Bernhard Rumpe

Agile Modeling with UML

Code Generation, Testing, Refactoring

Agile Modeling with UML

Bernhard Rumpe

Agile Modeling with UML

Code Generation, Testing, Refactoring

Springer

Bernhard Rumpe
Software Engineering
RWTH Aachen University
Aachen
Germany

ISBN 978-3-319-58861-2 ISBN 978-3-319-58862-9 (eBook)
DOI 10.1007/978-3-319-58862-9

Library of Congress Control Number: 2017939615

Based on a translation from the German language edition: Agile Modellierung mit UML – Codegenerierung, Testfälle, Refactoring © Springer Verlag Berlin Heidelberg 2005, 2012. All Rights Reserved.
© Springer International Publishing AG 2017
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Foreword to the First Edition

Today, software systems are generally complex products and the use of engineering techniques is essential if the systems are to be produced successfully. Over the last three decades, this finding, which is frequently quoted but is now more than 30 years old, has led to intensive work on languages, methods, and tools in the IT field of Software Engineering to support the software creation process. However, despite great advances, we must concede that in comparison with other much older engineering disciplines, many questions still remain unanswered and new questions are constantly arising.

For example, a superficial comparison with the field of construction quickly shows that in there, international standards have been set for creating models of buildings, analyzing the models, and then realizing the models in actual constructions. The distribution of roles and tasks is generally accepted and there are professional groups such as architects, structural engineers, as well as engineers for construction above and below ground.

This type of model-based approach is increasingly finding favor in software development. In recent years in particular, this has led to international attempts to define a generally accepted modeling language so that just like in construction, a model created by a software architect can be analyzed by a "software structural engineer" before it is implemented in executable programs by specialists responsible for the realization, i.e., programmers.

This standardized modeling language is the Unified Modeling Language and it is subject to continuous further development by an international consortium in a gradual process. Due to the wide range of interested parties in the standardization process, the current version 2.0 of UML has emerged as a language family with a great many open questions with regard to scope, semantic foundation, and methodological use.

Over the past few years, Professor Rumpe has dedicated himself to this problem in his scientific and practical work, the results of which are now

available to a wide audience in two books. In these books, Professor Rumpe focuses on the methodological process. In line with the current finding that lightweight, agile development processes offer great advantages particularly in smaller and medium-sized development projects, Professor Rumpe has developed techniques for an agile development process. On this basis, he has then defined a suitable modeling language by defining a language profile for UML. In this language profile, UML/P, Professor Rumpe has made UML leaner and rounded it off in some places to produce a manageable version of UML in particular for an agile development process.

Professor Rumpe has explained this language UML/P in detail in his previous book "Modeling with UML", which offers a significant basis for the current book (the content of the previous book is briefly summarized). The current book, "Agile Modeling with UML", is dedicated primarily to the methodological treatment of UML/P.

Professor Rumpe addresses three core topics of model-based software development. These are:

- Code generation, i.e., the automated transition from a model to an executable program
- Systematic testing of programs using a model-based, structured definition of test cases
- Further development of models using techniques for transformation and refactoring

Professor Rumpe initially examines all three core topics systematically and introduces the underlying concepts and techniques. For each topic, he then presents his approach based on the language UML/P. This division and clear separation between basic principles and applications make the presentation extremely easy to understand and also allows the reader to transfer this knowledge directly to other model-based approaches and languages.

Overall, this book is of great benefit to those who practice software development, for academic training in the field of Software Engineering, and for research in the area of model-based software development. Practitioners learn how to use modern model-based techniques to improve the production of code and thus significantly increase quality. Students are given both important scientific basics as well as direct applications of the basic techniques presented. And last but not least, the book gives scientists a comprehensive overview of the current status of development in the three core topics it covers.

The book therefore represents an important milestone in the development of concepts and techniques for a model-based and engineering-style software development and thus offers the basis for further work in the future. Practical experience of using the concepts will validate their stability. Scientific, conceptual work will provide further research on the topic of model transformation based on graph transformation in particular. It will also deepen the area of model analysis in the direction of structural model analysis.

This deeper understanding of the IT methods in model-based software development is a crucial prerequisite for a successful combination with other engineering-style methods, such as in the field of embedded systems or the area of intelligent, user-friendly products. The fact that the language UML/P is not specific to any domain also offers a lot of opportunities here.

Gregor Engels

Paderborn, September 2004

Preface to the Second Edition

As this is the second book on agile software development with UML, interested readers will probably be familiar with the first book [Rum16]. The preface in [Rum16] holds true for both books and here, therefore, I refer to the first book, in which the following aspects are discussed:

- Agile methods and model-based methods are both successful software development techniques.
- So far, the two approaches have not been harmonized or integrated.
- However, the basic idea of using models instead of programming languages provides the opportunity to do exactly that.
- This book contributes to this integration in the form of UML/P.
- In the second edition, UML/P has been updated and adapted to UML 2.3 and Java Version 6.

I hope you enjoy using this book and its contents.

Bernhard Rumpe

Aachen, Germany, March 2012

Preface to the English and 3rd Edition

Colleagues have asked when the English version of the two books would be published. The first one was finished in 2016 and now, here comes the second one. I wish all the readers, students, teachers, and developers fun and inspiration for their work.

I would like to thank all the people that helped me translating and quality checking this book, namely Tracey Duffy for the main translation, Sylvia Gunder and Gabi Heuschen for continuous support, Robert Eikermann for the Latex and continuous integration setup, Kai Adam (for reviewing Chapters 5,7 and 10), Vincent Bertram (8), Arvid Butting (3,8,10), Anabel Derlam (1), Katrina Engelbrecht (3,9,10), Robert Eikermann (3,8,9), Timo Greifenberg (6,7,11), Lars Hermerschmidt (11,), Steffen Hillemacher (3,7,11), Katrin Hölldobler (9,10), Oliver Kautz (3,8,10), Thomas Kurpick (2), Evgeny Kusmenko (2,5), Achim Lindt (1,2,9), Matthias Markthaler (7,9), Klaus Müller (4), Pedram Mir Seyed Nazari (1,5), Dimitri Plotnikov (1,4,5), Deni Raco (6,7,8), Alexander Roth (4), Christoph Schulze (6,8,11), Michael von Wenckstern (2,3,4,6), and Andreas Wortmann (1,11).

Bernhard Rumpe

Aachen, Germany, February 2017

Further material:

http://mbse.se-rwth.de

Contents

1 Introduction 1
1.1 The Goals and Content of Volume 1 2
1.2 Additional Goals of This Book 4
1.3 Overview 6
1.4 Notational Conventions 7

2 Agile and UML-Based Methodology 9
2.1 The Software Engineering Portfolio 11
2.2 Extreme Programming (XP) 13
2.3 Selected Development Practices 19
2.3.1 Pair Programming 19
2.3.2 Test-First Approach 20
2.3.3 Refactoring 23
2.4 Agile UML-Based Approach 24
2.5 Summary 30

3 Compact Overview of UML/P 33
3.1 Class Diagrams 34
3.1.1 Classes and Inheritance 34
3.1.2 Associations 35
3.1.3 Representation and Stereotypes 37
3.2 Object Constraint Language 39
3.2.1 OCL/P Overview 39
3.2.2 OCL Logic 42
3.2.3 Container Data Structures 42
3.2.4 Functions in OCL 49
3.3 Object Diagrams 51
3.3.1 Introduction to Object Diagrams 51
3.3.2 Compositions 54
3.3.3 The Meaning of an Object Diagram 54
3.3.4 The Logic of Object Diagrams 55

3.4 Statecharts ... 56
3.4.1 Properties of Statecharts ... 56
3.4.2 Representation of Statecharts ... 60
3.5 Sequence Diagrams ... 65

4 Principles of Code Generation ... 71
4.1 Concepts of Code Generation ... 74
4.1.1 Constructive Interpretation of Models ... 76
4.1.2 Tests versus Implementation ... 78
4.1.3 Tests and Implementation from the Same Model ... 81
4.2 Code Generation Techniques ... 82
4.2.1 Platform-Dependent Code Generation ... 82
4.2.2 Functionality and Flexibility ... 85
4.2.3 Controlling the Code Generation ... 88
4.3 Semantics of Code Generation ... 89
4.4 Flexible Parameterization of a Code Generator ... 91
4.4.1 Implementing Tools ... 92
4.4.2 Representation of Script Transformations ... 94

5 Transformations for Code Generation ... 99
5.1 Transformations for Class Diagrams ... 100
5.1.1 Attributes ... 100
5.1.2 Methods ... 103
5.1.3 Associations ... 106
5.1.4 Qualified Associations ... 110
5.1.5 Compositions ... 114
5.1.6 Classes ... 116
5.1.7 Object Instantiation ... 119
5.2 Transformations for Object Diagrams ... 123
5.2.1 Object Diagrams Used For Constructive Code ... 123
5.2.2 Example of a Constructive Code Generation ... 125
5.2.3 Object Diagram Used as Predicate ... 125
5.2.4 An Object Diagram Describes a Structure Modification 129
5.2.5 Object Diagrams and OCL ... 131
5.3 Code Generation from OCL ... 132
5.3.1 An OCL Expression as a Predicate ... 133
5.3.2 OCL Logic ... 135
5.3.3 OCL Types ... 137
5.3.4 A Type as an Extension ... 139
5.3.5 Navigation and Flattening ... 140
5.3.6 Quantifiers and Special Operators ... 141
5.3.7 Method Specifications ... 141
5.3.8 Inheritance of Method Specifications ... 145
5.4 Executing Statecharts ... 146
5.4.1 Method Statecharts ... 146

5.4.2 The Transformation of States ... 147
5.4.3 The Transformation of Transitions ... 152
5.5 Transformations for Sequence Diagrams ... 155
5.5.1 A Sequence Diagram as a Test Driver ... 155
5.5.2 A Sequence Diagram as a Predicate ... 157
5.6 Summary of Code Generation ... 158

6 Principles of Testing with Models ... 161
6.1 An Introduction to the Challenges of Testing ... 162
6.1.1 Terminology for Testing ... 163
6.1.2 The Goals of Testing Activities ... 165
6.1.3 Error Categories ... 167
6.1.4 Terminology Definitions for Test Procedures ... 168
6.1.5 Finding Suitable Test Data ... 169
6.1.6 Language-Specific Sources for Errors ... 169
6.1.7 UML/P as the Test and Implementation Language ... 171
6.1.8 A Notation for Defining Test Cases ... 174
6.2 Defining Test Cases ... 177
6.2.1 Implementing a Test Case Operatively ... 177
6.2.2 Comparing Test Results ... 178
6.2.3 The JUnit tool ... 181

7 Model-Based Tests ... 185
7.1 Test Data and Expected Results using Object Diagrams ... 186
7.2 Invariants as Code Instrumentations ... 189
7.3 Method Specifications ... 191
7.3.1 Method Specification for Code Instrumentation ... 191
7.3.2 Method Specifications for Determining Test Cases ... 191
7.3.3 Defining Test Cases using Method Specifications ... 194
7.4 Sequence Diagrams ... 195
7.4.1 Triggers ... 196
7.4.2 Completeness and Matching ... 198
7.4.3 Noncausal Sequence Diagrams ... 199
7.4.4 Multiple Sequence Diagrams in a Single Test ... 199
7.4.5 Multiple Triggers in a Sequence Diagram ... 200
7.4.6 Interaction Patterns ... 200
7.5 Statecharts ... 202
7.5.1 Executable Statecharts ... 202
7.5.2 Using Statecharts to Describe Sequences ... 205
7.5.3 Statecharts used in Testing ... 206
7.5.4 Coverage Metrics ... 208
7.5.5 Transition Tests instead of Test Sequences ... 211
7.5.6 Further Approaches ... 212
7.6 Summary and Open Issues Regarding Testing ... 212

8 **Design Patterns for Testing** 217
8.1 Dummies 219
8.1.1 Dummies for Layers of the Architecture 221
8.1.2 Dummies with a Memory 222
8.1.3 Using a Sequence Diagram instead of Memory 223
8.1.4 Catching Side Effects 224
8.2 Designing Testable Programs 224
8.2.1 Static Variables and Methods 225
8.2.2 Side Effects in Constructors 228
8.2.3 Object Instantiation 228
8.2.4 Predefined Frameworks and Components 230
8.3 Handling of Time 232
8.3.1 Simulating Time in a Dummy 233
8.3.2 A Variable Time Setting in a Sequence Diagram 234
8.3.3 Patterns for Simulating Time 236
8.3.4 Timers 237
8.4 Concurrency with Threads 237
8.4.1 Separate Scheduling 238
8.4.2 Sequence Diagrams as Scheduling Models 240
8.4.3 Handling Threads 241
8.4.4 A Pattern for Handling Threads 241
8.4.5 The Problems of Forcing Sequential Tests 243
8.5 Distribution and Communication 245
8.5.1 Simulating the Distribution 245
8.5.2 Simulating a Singleton 247
8.5.3 OCL Constraints across Several Locations 248
8.5.4 Communication Simulates Distributed Processes 249
8.5.5 Pattern for Distribution and Communication 251
8.6 Summary 253

9 **Refactoring as a Model Transformation** 255
9.1 Introductory Examples for Transformations 256
9.2 The Methodology of Refactoring 261
9.2.1 Technical and Methodological Prerequisites for Refactoring 261
9.2.2 The Quality of the Design 263
9.2.3 Refactoring, Evolution, and Reuse 264
9.3 Model Transformations 265
9.3.1 Forms of Model Transformations 265
9.3.2 The Semantics of a Model Transformation 266
9.3.3 The Concept of Observation 272
9.3.4 Transformation Rules 277
9.3.5 The Correctness of Transformation Rules 278
9.3.6 Transformational Software Development Approaches . 280
9.3.7 Transformation Languages 282

10 Refactoring of Models 285
10.1 Sources for UML/P Refactoring Rules 286
10.1.1 Defining and Representing Refactoring Rules 288
10.1.2 Refactoring in Java/P 289
10.1.3 Refactoring Class Diagrams 295
10.1.4 Refactoring in OCL 301
10.1.5 Introducing Test Patterns as Refactoring 302
10.2 A Superimposition Method for Changing Data Structures . . . 306
10.2.1 Approach for Changing the Data Structure 306
10.2.2 Example: Representing a Bag of Money 308
10.2.3 Example: Introducing the Chair in the Auction System 312
10.3 Summary of Refactoring Techniques 320

11 Summary, Further Reading and Outlook 323
11.1 Summary 324
11.2 Outlook 325
11.3 Agile Model Based Software Engineering 328
11.4 Generative Software Engineering 331
11.5 Unified Modeling Language (UML) 332
11.6 Domain Specific Languages (DSLs) 332
11.7 Software Language Engineering (SLE) 335
11.8 Modeling Software Architecture and the MontiArc Tool 339
11.9 Variability and Software Product Lines (SPL) 342
11.10 Semantics of Modeling Languages 344
11.11 Compositionality and Modularity of Models and Languages 348
11.12 Evolution and Transformation of Models 349
11.13 State Based Modeling (Automata) 351
11.14 Modelling Cyber-Physical Systems (CPS) 354
11.15 Applications in Cloud Computing and Data-Intensive Systems 355
11.16 Modelling for Energy Management 356
11.17 Modelling Robotics 358
11.18 Automotive Software 359
11.19 Autonomic Driving and Driver Intelligence 360

References 363

Index 385

1

Introduction

The real purpose of a book is
to trap the mind into
doing its own thinking.

Christopher Darlington Morley

Many projects today demonstrate quite spectacularly how expensive incorrect or faulty software can be.

In recent years we have seen a continuous increase in the complexity of software-based projects and products, both in the domains of operative or administrative information and web systems, as well as in cyber-physical systems such as cars, airplanes, production systems, e-health and mobile systems. To manage the arising complexities, an effective portfolio of concepts, techniques, and methods has been developed, allowing Software Engineering to become a fully-fledged engineering discipline.

The portfolio is in no way fully matured yet; it still has to become much more firmly established, above all in today's industrial software development processes. The capabilities of modern programming languages, class libraries, and existing software development tools allow us to use approaches today that seemed unfeasible just a short time ago.

Further material:

http://mbse.se-rwth.de

© Springer International Publishing AG 2017
B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_1

As part of this Software Engineering portfolio, this book examines a UML-based methodology which focuses primarily on techniques for using UML in practice. The most important techniques recognized and examined in this book are:

- Generating code from models
- Modeling test cases
- Refactoring of models to enable evolution

This book, Volume 2, is based heavily on the first volume, "Modeling with UML. Language, Concepts, and Methodology." [Rum16], which explains the language profile UML/P in detail. Therefore, when reading this volume, [Rum17], we recommend that you take the first volume [Rum16] at hand, even though parts of Volume 1 are repeated in a compact form in Chapter 3 of this second volume.

1.1 The Goals and Content of Volume 1

Joint mission statement of both volumes: One of the core goals of both volumes of this book is to provide basic techniques for model-based development (MBD) for the Software Engineering portfolio referred to above. Volume 1 presents a variant of UML that is particularly suitable for efficiently developing high-quality software and software-based systems. Building on this foundation, this second volume contains techniques for generating code and test cases and for refactoring UML/P models.

UML standard: The UML 2 standard has to satisfy a multitude of requirements from a variety of influences and is therefore inevitably overloaded with plentitude of different modelling diagrams with a broad and semi-clear semantics. Many elements of the standard are not suitable for our purposes, or at least not in their given form, and other language concepts are missing entirely. This book therefore presents an adapted language profile of UML referred to as UML/P. This adapted language profile UML/P is optimized for the proposed development techniques for design, implementation, and maintenance and can therefore be used more easily in agile development approaches.

Volume 1 concentrates primarily on defining the language profile and providing a general overview of the proposed methodology.

UML/P has arisen as the result of multiple basic research and application projects. In particular, the sample application presented in Appendix D, Volume 1 was developed using the principles described here. The auction system is also ideally suited for demonstrating the techniques developed in both volumes of this book because changes to the business model or the company environment occur particularly frequently in this application domain. Flexible but high-quality software development is essential here.

Object orientation and Java: New business applications today use primarily object technology. The existence of versatile class libraries and frameworks, the variety of tools available, and not least the largely successful language design substantiate the success of the programming language Java. The UML language profile UML/P and the development techniques that build on it are therefore tailored to Java.

Bridge between UML and agile methods: At the same time, the two volumes of this book form an elegant bridge between two approaches generally held to be incongruous: agile methods and the modeling language UML. Agile methods, and in particular Extreme Programming, have a number of interesting techniques and principles that enrich the Software Engineering portfolio for certain types of projects. These techniques typically involve no documentation being created; they concentrate on flexibility, optimize the time to market, and minimize the quantity of resources used. Nevertheless, these techniques still ensure the required quality. Agile methods are therefore very suitable as a basis for the goals of this book.

Agile approach based on UML/P: UML is used as notation for a range of activities, such as modeling business cases, analyzing the current and required form of a system as well as architecture and preliminary and detailed design at various levels of granularity. The artifacts of UML are therefore an important foundation for planning and controlling software development projects that are based on milestones. Accordingly, UML is used primarily in plan-based projects that involve a relatively high level of documentation and the inflexibility that this causes. However, compared to a normal programming language, UML is more compact, more semantically comprehensive, and more suitable for representing complex content. It therefore offers significant advantages for modeling test cases and for the transformational evolution of software systems. Based on a discussion of agile methods and the concepts they encompass, Volume 1 outlines an agile method that uses the UML/P language profile as the basis for many activities without bringing in the inflexibility of typical UML-based methods.

The goals described above were implemented in the following chapters in Volume 1:

1 Introduction

2 Class Diagrams

Introduces the form and use of class diagrams.

3 Object Constraint Language

Discusses a version of the textual description language OCL which has been adapted to Java, extended syntactically, and consolidated semantically.

4 Object Diagrams

Discusses the language and methodological use of object diagrams as well as their integration with the OCL logic to enable a “diagram logic”

in which undesirable situations, alternatives, and combinations can be described.

5 Statecharts
In addition to introducing Statecharts, this chapter contains a collection of transformations for simplifying Statecharts whilst simultaneously preserving their semantics.

6 Sequence Diagrams
Describes the form, meaning, and use of sequence diagrams.

A Language Representation with Syntax Class Diagrams
Offers a combination of Extended Backus-Naur Form (EBNF) and specialized class diagrams for representing the abstract syntax (metamodel) of UML/P.

B Java
Describes the abstract syntax of that part of Java used in the book.

C The Syntax of UML/P
Describes the abstract syntax of UML/P.

D Sample Application: Internet-Based Auction System
Outlines the background information for the auction system example used in both volumes of the book.

1.2 Additional Goals of This Book

To increase efficiency in a project, developers need effective notations, techniques, and methods. Because the primary goal of any software development is to produce the executable and correctly implemented product system, UML should not be used solely for documenting designs: automated transformations of UML models into code using *code generators*, the definition of *test cases* with UML/P for quality management, and the evolution of UML models with *refactoring* techniques are essential aspects.

The combination of code generation, test case modeling, and refactoring offers significant synergy effects which contribute, for example, to quality management, to increasing reuse, and to improving the capability for evolution.

Code generation: Generating code from abstract models is essential for creating a system efficiently. The form of code generation discussed in this book allows us to develop models for specific domains and applications compactly and largely independent of a concrete technology. Technology-dependent aspects such as the database connection, communication, or the GUI representation are only added when the code is generated. This means that we can use UML/P as a programming language and there is no conceptual break between the modeling language and the programming language. However, it is important to differentiate between executable and abstract models explicitly in the software development process and to use each type of model appropriately.

Modeling automatable tests: Developing and executing tests systematically and efficiently is a key factor in ensuring the quality of a system. The goal is that once created, the tests can run automatically. Code generation is therefore used not only for developing the product system, but in particular also for test cases in order to check the consistency between the specification and the implementation. Using UML/P to model test cases is therefore a significant part of an agile methodology. In particular, object diagrams, OCL, and sequence diagrams are used to model test cases. Fig. 1.1 illustrates the first two goals of this book.

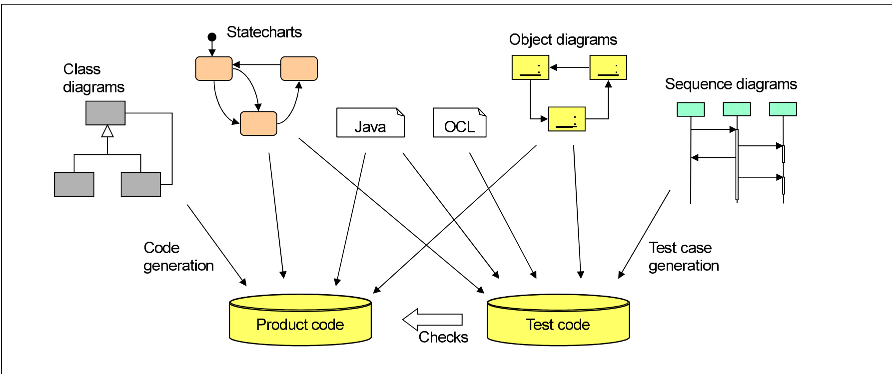


Fig. 1.1. Notations in UML/P

Evolution with refactoring: The flexibility discussed in this book for a quick reaction to changes in the requirements or the technology requires a technique for adapting the existing model or the implementation systematically. Ideally, we use refactoring techniques for system evolution reacting to new requirements or a new form of use, as well to eliminate structural deficits in the software architecture. The book therefore also focuses on establishing and embedding refactoring techniques as part of the more general approach for model transformation. It also discusses which types of refactoring rules can be developed for UML/P or adopted from other approaches. In this context, special attention is given to class diagrams, Statecharts, and OCL.

With regard to both, test case modeling and refactoring techniques, the book presents findings from the basic theories and applies them to UML/P. The goal of the book is to explain these concepts using numerous practical examples and to adapt them to UML diagrams in the form of test patterns and refactoring techniques.

Model-driven architecture (MDA) is a development method initiated by the industry consortium OMG[1]. Its primary goal is to intensify the use of UML tools in software development. MDA, for example, for code generation, offers similar techniques to the approach discussed in this book. The refactoring of models presented in this book extends MDA by to "horizontal" transformations.

Limitation: With the techniques discussed, this book concentrates mainly on supporting design, implementation, and maintenance activities but does not cover these activities in their entirety. Important aspects that are also not covered here are techniques and notations for eliciting and managing requirements, for project planning and execution, for control, and for version and change management. Instead, there are references at appropriate points to relevant further literature.

1.3 Overview

Fig. 1.2 provides a good overview of the matrix-type structure of further parts of both volumes. While this second volume deals more intensively with methodological issues, Volume 1 explains UML/P.

	General Information	Class diagrams	OCL	Object diagrams	Statecharts	Sequence diagrams
Introduction and discussion	Chapter 2	*Chapter 2 (Volume 1)*	*Chapter 3 (Volume 1)*	*Chapter 4 (Volume 1)*	*Chapter 5 (Volume 1)*	*Chapter 6 (Volume 1)*
Syntax	*Appendix A & B & C.1 (Volume 1)*	*Appendix C.2 (Volume 1)*	*Appendix C.3 (Volume 1)*	*Appendix C.4 (Volume 1)*	*Appendix C.5 (Volume 1)*	*Appendix C.6 (Volume 1)*
Code generation	Chapter 4	Chapter 5.1	Chapter 5.3	Chapter 5.2	Chapter 5.4	Chapter 5.5
Test case modeling	Chapter 6	(Chapter 8)	Sections 7.2 & 7.3	Section 7.1	Section 7.5	Section 7.4
Refactoring	Chapter 9	Sections 10.1 & 10.2	Section 10.1	(Section 10.2)	*Section 5.6 (Volume 1)*	(Section 10.2)

Fig. 1.2. Overview of the content of both volumes

Chapter 2 outlines an agile UML-based approach. This approach uses UML/P as the primary development language for creating executable models, generating code from the models, designing test cases, and planning architecture adjustments (refactoring).

Chapter 3 gives a brief and compact summary of Volume 1, [Rum16]. In particular, it presents the language profile of UML/P introduced in Volume 1 very compactly and therefore incompletely.

[1] The Object Management Group (OMG) is responsible for the definition of UML in the form of a "Technical Recommendation".

Chapters 4 and 5 discuss principal and technical problems with regard to code generation. This discussion addresses the architecture of a code generator and methods for controlling it, as well as the suitability of UML/P notations for test or product code. Building on this discussion, these two chapters introduce a mechanism for describing code generation. Furthermore, using selected parts of the various UML/P notations, these chapters show how test and product code can be generated from the models created in those parts of UML/P. In this context, the chapters discuss alternative results of the generation and demonstrate the effects of combining transformations.

Chapters 6 and 7 discuss the concepts for test approaches which are familiar from literature. These chapters also discuss the special features which arise due to the use of UML/P as the test and implementation language and which have to be taken into consideration during testing. The chapters describe the appearance of an architecture for automated test execution and discuss how we can use UML/P diagrams to define test cases.

Chapter 8 uses test patterns to show how we can use UML/P diagrams to define test cases. These test patterns contain practical information for defining programs that can be tested, in particular for functional tests for distributed and concurrent software systems.

Chapters 9 and 10 discuss techniques for transforming models and code and thus provide a solid foundation for refactoring as a type of transformation that preserves semantics. For refactoring, the chapters introduce an explicit notation of observation that can be used in practice and discuss how we can apply existing refactoring techniques to UML/P. Finally, the chapters propose an *additive approach* that supports refactoring on a larger scale using OCL invariants.

1.4 Notational Conventions

Various types of diagrams and textual notations are used in this book. To enable easy identification of the type of diagram or textual notation presented in each case, the top right-hand corner contains a label in one of the forms illustrated in Fig. 1.3. This is an approach which deviates from UML 2. These labels are also suitable for labeling textual parts of diagrams and are more flexible than the UML 2 labeling. On the one hand, a label is used as an orientation aid; on the other hand, it is used as part of UML/P, as the name of the diagram and certain diagram properties can be added to the label in the form of stereotypes. In individual cases, special types of labels are used and these are generally self-explanatory.

The textual notations such as Java code, OCL descriptions, and textual parts in diagrams are based exclusively on the ASCII character set. For a better readability of these textual notations, individual keywords are highlighted or underlined.

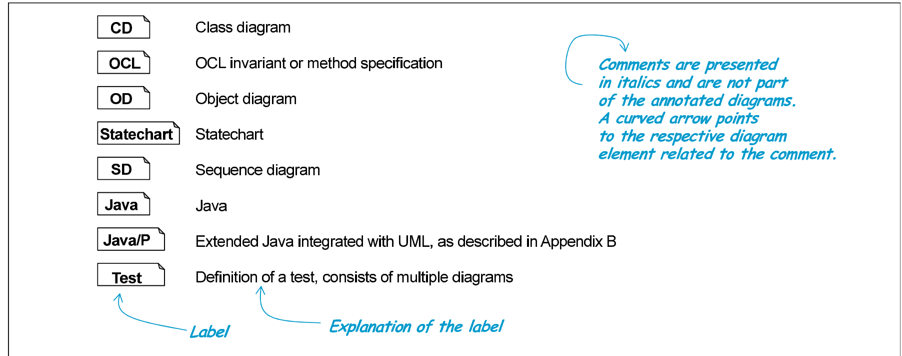


Fig. 1.3. Labels for diagrams and text parts

The following special characters are used in this book:

- The representation indicators "..." and "©" are a formal part of UML/P and describe whether the representation presented in a diagram is complete.
- Stereotypes are specified in the form ≪stereotypeName≫. Tags have the form {`tagName=value`} or {`tagName`}.

2

Agile and UML-Based Methodology

The most valuable insights are the methods.
Friedrich Wilhelm Nietzsche

As useful modeling language must be embedded in a methodology. This chapter presents characteristics of agile methods, in particular those of the process of *Extreme Programming (XP)* [Bec04, Rum01]. Using these characteristics and further elements, the chapter introduces a proposal for an agile methodology based on UML.

2.1 The Software Engineering Portfolio **11**
2.2 Extreme Programming (XP) **13**
2.3 Selected Development Practices **19**
2.4 Agile UML-Based Approach **24**
2.5 Summary **30**

© Springer International Publishing AG 2017

B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_2

For many years now, the improvements in Software Engineering have been responsible for the continual increases in efficiency in software development projects. These days, software has to be created with a high level of quality with increasingly fewer personnel resources in increasingly shorter timeframes. Processes such as the Rational Unified Process (RUP) [Kru03] or the V-Modell XT [HH08] are more suitable for large projects with a lot of team members. For smaller projects, however, these processes are overloaded with activities that are not entirely essential for the end result. Consequently, the processes are too inflexible to be able to respond to the rapidly changing environment (technology, requirements, competing products) of a software development project.

A relatively new group of processes has arisen under the common label of "agile processes". These new processes are characterized primarily by flexibility and quick feedback. They concentrate on the main work results and focus on the people involved in the project—in particular, customers.

The Standish Report [Gro15] describes how significant causes of project failure can be found in poor project management: there is insufficient communication; too much, too little, or incorrect documentation; risks are not counteracted in time; and feedback is requested from users too late.

It is the human element in particular—the communication within the project and the cooperation amongst the developers and between the developers and the users—that is seen as the main cause of failure for software development projects. [Coc06] also states that projects rarely fail for technical reasons. On the one hand, this indicates that the developers have a good command of even innovative technologies; but on the other hand, this claim must be questioned to some extent at least. If the technology used causes difficulties, these problems often disrupt the communication between team members. As the project continues, these problems in communication come to the fore for emotional reasons and are ultimately remembered as "perceived" reasons for the failure of the project.

Because use only a reduced set of appropriate languages and tools is used this generation of processes can be regarded as lightweight. The more compact the language and the better the analysis and generation tools, the less redundancy is necessary. In addition, the efficiency of the developers is improved by reducing additional efforts and expenses for management and documentation.

Today, Software Engineering offers an extensive *portfolio* of approaches, principles, development practices, tools, and notations that we can use to develop software systems in various forms, sizes, and levels of quality. These elements of the portfolio complement each other to some extent, but can also be used as alternatives to one another, which means that there is a wide range of selection options available for managing, controlling, and executing a project.

This chapter looks firstly at the current status of the Software Engineering portfolio. Section 2.2 discusses "Extreme Programming" (XP) and Section 2.3

presents three essential practices from XP. Section 2.4 contains a sketch for a method that is suitable as a reference for the use of UML/P and the techniques discussed in detail in this book.

Chapter 3 provides a compact overview of the Unified Modeling Language profile UML/P. This profile can be used for the method proposed here and is supported, e.g., in [Sch12] with a suitable tool. Like UML itself, UML/P is largely *not method-specific*. This means that it is possible and even helpful to use UML/P in other methods. However, as UML/P focuses on the ability to generate code and tests, it is particularly suitable for agile methods.

2.1 The Software Engineering Portfolio

[AMB+04] and [BDA+99] attempt to consolidate the knowledge about Software Engineering that has built up over more than 40 years into a "Software Engineering Body of Knowledge" (SWEBOK). In the SWEBOK, concept formations are standardized, the main core elements of Software Engineering are presented as an engineering discipline. The goal is to establish a generally accepted consensus about the content and concepts of Software Engineering.

Some of the terminology used for software development processes which is significant for our deliberations is shown in Fig. 2.1.

Experience gained from the execution of software development projects in recent years clearly shows that there cannot be *one, unique* process for software development: Projects strongly differ in importance, size, area of application, and project environment. Instead, efforts are being made to compile a collection of concepts, best practices, and tools that allow project-specific requirements to be taken into account in an individual process. The level of detail and the precision of the documents, milestones, and results to be delivered are defined dependent on the size of the project and the desired quality of the results in each case. Existing process descriptions, which can be viewed as templates, can be helpful. However, project-specific adjustments are considered necessary in most cases. Therefore, it is useful for those involved in a project to be familiar with as many approaches from the current portfolio as possible.

The 1990s saw a strong trend towards complete and therefore rather bureaucratic software development processes. The agile methods of the 2000s have broken away from this trend. Two factors enabled this change of direction: firstly, the significantly increased understanding of the tasks involved in developing complex software systems; and secondly, the availability of improved programming languages, compilers, and a number of other development tools. Today, it is almost as efficient to implement a GUI immediately as it is to specify the GUI. The specification can therefore be replaced by a prototype which the user can try out and which can be reused in the realization of the final product. The trend towards reducing the level of required

Development method: A development method (synonym *process model*) describes the procedure "for executing software creation in a project" [Pae00]. We can differentiate between a technical, a social, and an organizational development method.

Software development process: This term is occasionally used as a synonym for "development method" but is often understood as a more detailed form. Thus, [Som10] defines a process as a set of activities and results used to create a software product. In most cases, the chronological sequence or the dependencies of the activities are also defined.

Development task: A process is divided into a series of development tasks. Each task delivers certain results in the form of *artifacts*. The team members participating in the project perform *activities* in order to complete these tasks.

Principle: Principles are fundamentals on which action is based. Principles are generally valid, abstract, and as general as possible in nature. They form a theoretical basis. Principles are derived from experience and findings (see [Bal00]).

Best practices: This term describes successfully tested *development practices* in development processes. A development practice can be perceived as a specific, operationalized process pattern that implements a general principle (see RUP [Kru03] or XP [Bec04]).

Artifact: Development results are represented with a specific form of notation—for example, natural language, UML, or a programming language. The documents of these languages are called *artifacts* and examples include requirements analyses, models, code, review results, or a glossary. An artifact can have a hierarchical structure.

Transformation: Developing a new artifact and improving a version of an existing artifact can both be understood as transformations. The development or improvement can be automated or manual. Ultimately, almost all activities can be seen as transformations of the set of given artifacts in a project.

Fig. 2.1. Terminology definitions for the software development process

developer capacities is intensified by the fact that having fewer people involved in a project also reduces the level of organizational overhead, which in turn can further reduce the workload.

When we use agile methods, the increased emphasis on the individual capabilities and needs of the developers and customers involved in a project allows us to reduce project bureaucracy even further in favor of greater individual responsibility. This focus on the team can also be observed in other areas of economic life—for example, where flat management hierarchies are used. It is based on the assumption that mature and motivated project participants will demonstrate responsibility and courage by taking the initiative when the project environment gives them the opportunity to do so.

2.2 Extreme Programming (XP)

Extreme Programming (XP) is an "agile" software development method. The main elements of this method are described in [Bec04]. Although XP as an approach was defined and refined in, for example, software development projects at a Swiss bank, its name already indicates a strong influence by the software development culture of North America, which is defined by pragmatism. Despite its given name, XP is no hacker technology; rather, it has some very detailed, elaborate methodological aspects which have to be applied rigorously. These aspects allow its supporters to postulate that XP can be used to create high-quality software with relatively low effort, within budget, and to the satisfaction of the customers. Statistically meaningful studies of XP projects are more than mere anecdotes [DD08, RS02, RS01].

XP is rather popular in practice. Many books have already been written about XP [Bec04, JAH00, BF00, LRW02] discussing different aspects of XP in detail, as well as the current levels of knowledge about this methodology or illustrate case studies of projects that have already been conducted [NM01]. [Wak02] and [AM01] contain particular practical aids for implementing XP, [Woy08] looks at cultural aspects, and [BF00] discusses planning in XP projects.

[EH00a] and [EH01] offer a critical description of XP with an explicit discussion of its weaknesses. Amongst other things, these works criticize the lack of use of modeling techniques such as UML and draw a critical comparison with Catalysis [DW98]. A dialectic discussion of the advantages and disadvantages of Extreme Programming can be found in [KW02]. It highlights, for example, the necessity of a disciplined approach, as well as the strong and, compared to classic approaches, significantly modified demands placed on the team leader and the coach in particular.

Important elements of XP are presented and discussed below in accordance with the introductions given in [Bec04] and [Rum01]. Further topics in literature now treat XP in parallel with other methods [Han10, Leh07, Ste10, HRS09] and thus support a portfolio of agile techniques. Alternatively, they adapt agile methods for distributed teams [Eck09] or cover the migration of companies to agile methods [Eck11].

Overview of XP

XP is a lightweight method of software development. It dispenses with the need for a number of elements from classic software development in order to allow faster and more efficient coding. The potential deficits this causes for quality management are compensated for by a stronger weighting for other concepts (in particular, the test process). XP consists of a larger number of concepts. Within the scope of this overview, we will cover only the most important of these.

XP tries explicitly not to use new methodological concepts or methodological concepts that have not yet been tested to any great extent. Instead, it integrates proven techniques into a process model which focuses on the essentials and dispenses with the need for organizational ballast as far as possible. Since the ultimate goal of software development is source code, this is what XP focuses on from the very beginning. Any additional documentation is considered to be ballast that should be avoided. Creating documentation takes a lot of effort, and the documentation is often much more erroneous than the code itself because it cannot usually be analyzed and tested automatically to a satisfactory extent. In practice, customers frequently present new or modified requirements; documentation reduces the flexibility of the evolution and adaptation of the system as a quick response to these new or modified requirements. Therefore, almost no documentation is created in XP projects (with the exception of the code and the tests). To compensate for this, good comments in the source code based on coding standards and an extensive test suite are very important.

The primary goal of XP is the efficient development of high-quality software on time and within budget. The mechanisms used to do this are illustrated by the *values*, the *principles*, the basic *activities*, and the *development practices* implemented in the activities shown in the pyramid in Fig. 2.2.

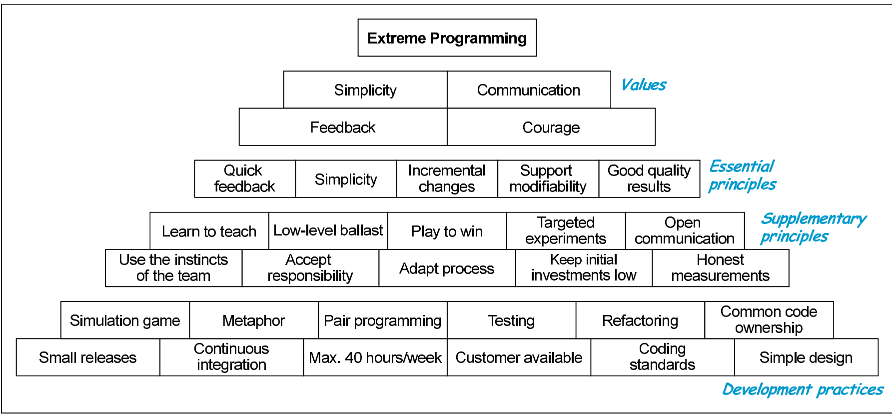


Fig. 2.2. Structure of Extreme Programming

Success Factors of XP

Now that XP has been in use for some years, we can clearly identify some of the major factors for the success of XP projects:

- The team is motivated and the working environment is suitable for XP. This means, for example, that working places are set up for pair program-

ming and the developers sit in close proximity to one another and to the customer.
- The customer is actively involved in the project and is available to answer questions. The study [RS02] showed that this has to be rated as one of the most critical success factors for XP projects.
- The importance of tests at all levels becomes clear as soon as there are any changes, new developers join the team, or the system reaches a certain size which means that manual testing can no longer be performed.
- The result of the drive for simplicity, which is a topic that is being discussed in all domains, means that documentation is omitted. Equally, the design is as simple as possible. These factors allow a significant reduction in the workload.
- The lack of system specifications and the presence of a customer who can be included in negotiations about functionality during the course of the project means that the customer is integrated in the project more intensively. This has two effects: on the one hand, it allows a fast response to changing customer wishes. On the other hand, it also allows the project to influence customer wishes. The project success thus also becomes a social agreement between the customer and the team of developers and not solely an objectively tested achievement of objectives based on documents.

This last aspect in particular corresponds to the XP philosophy of less control and a greater demand for individual responsibility and commitment. In a world in which requirements are constantly changing, this could lead to a more satisfactory result for everyone involved in the project than is possible with fixed system specifications.

Limits to the Applicability of XP

As far as project documentation and the integration of customers are concerned, XP is indeed revolutionary. Accordingly, there are a number of constraints and requirements that apply to the project size and project environment. These are discussed in various works, including [Bec04], [TFR02], and [Boe02]. XP is particularly suitable for projects with up to ten team members [Bec04] but it is evidently a problem to scale XP for large projects, as discussed in [JR01], for example. XP is simply one more approach in the Software Engineering portfolio—just like many other techniques, it can only be used under certain premises.

The basic assumptions, techniques, and concepts of XP polarize opinions. On the one hand, some programmers believe that XP elevates hacking to the status of an approach; on the other hand, XP is not taken entirely seriously because it ignores a lot of development processes that have been compiled over earlier decades. In fact, both arguments are only correct to a limited extent. On the one hand, it is true that hackers are more attracted to an XP-type

approach than to an approach according to RUP. On the other hand, when they look more closely, many software developers will recognize development practices that are already established. Furthermore, the XP approach is very strict and requires discipline for implementation.

It is certainly correct that XP is a lightweight software development method which is positioned explicitly as a counterweight to heavyweight methods such as RUP [Kru03] or V-Modell XT [HH08]. Significant differences in XP include firstly, the fact that it concentrates solely on code as a result, and secondly, the integration of the needs of the project participants. However, the most interesting difference is the increased capability of XP to respond to changes in the project environment or the user requirements flexibly. This is why XP belongs to the group of "agile methods".

The Costs of Fixing Bugs in XP Projects

One of the fundamental assumptions in XP questions important findings in Software Engineering. Previously, the assumption was that the costs for fixing errors or for implementing modifications increase exponentially over time, as described in [Boe81]. However, the assumption for XP is that these costs flatten out over the course of the project. Fig. 2.3 shows these two cost curves.

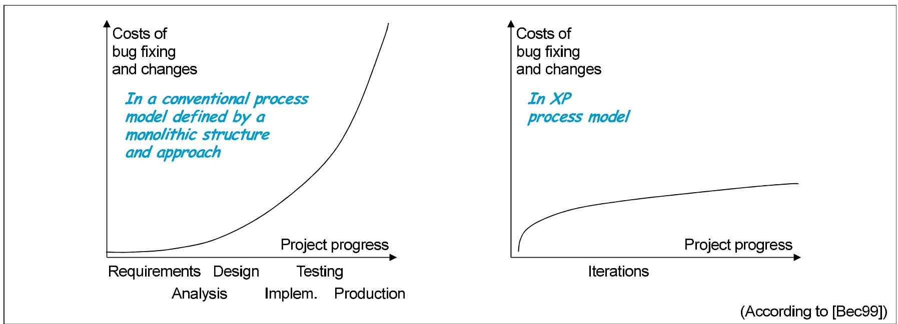


Fig. 2.3. Bug-fixing costs over the course of a project

In XP, the assumption is that the costs of modifications no longer increase dramatically over time [Bec04, Chapter 5]. There is no real empirical evidence for this assumption; however, it does have significant implications for the applicability of XP. If we assume that the cost curve can be flattened out with XP, then it is actually no longer essential to develop an initial architecture which is largely correct and which can be extended for all future developments. The entire profitability of the XP approach is therefore based on this assumption.

However, there are indicators that support at least a certain amount of flattening out of the cost curve in XP. Defects can be eliminated more quickly and more extensively by considering the following aspects, all of which are reflected in coding standards: using better languages such as Java, using better web and database technologies, and improving development practices. The use of better tools and development environments also helps in eliminating defects more effectively. Due to the common code ownership, even defects that are not localized in one artefact can be eliminated without the need for a series of planning and discussion meetings. The waiving of documentation removes the necessity of keeping any documents created consistent. On the contrary, with XP we have the effort and expense of updating automated tests. However, the fact that the tests are automated offers the significant advantage that tests which are no longer correct can be recognized efficiently—compared to the expensive and time-consuming proofreading of written documentation.

One of the main indications of a reduced cost curve, however, is an approach that uses small iterations. Errors and defects that can be localized in one iteration only have a local effect and remain within the iteration. In the subsequent iterations the effect is limited, meaning that only a slow increase in bug-fixing costs can then be expected there. The iterative approach, possibly coupled with a decomposition of the system into subsystems, may therefore produce the cost curve presented in Fig. 2.4. This is the cost curve that we saw in the auction project, for example.[1]

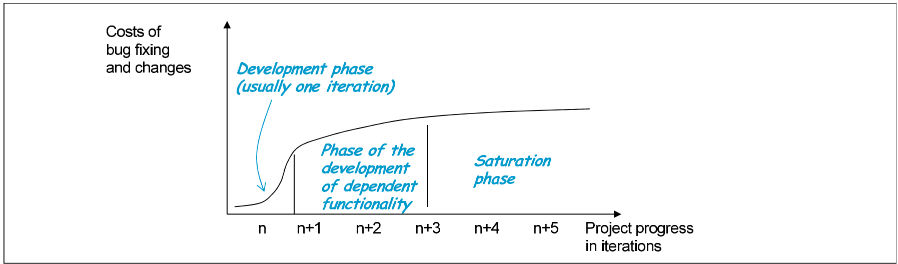


Fig. 2.4. Bug-fixing costs in iterative projects

Depending on whether the error can be localized within one part of the system, a certain increase in bug-fixing costs can arise in the causal and immediately subsequent iterations. In later iterations, it is only the costs of identifying the defect and its source that should increase. For defects in the architecture, however, which by their nature affect many parts of the system, the saturation occurs very late and at a high level. It is therefore generally worth investing a certain amount of initial expense in modeling the architecture.

[1] However, there is no statistically valid numerical data for this.

Hence, even though certain arguments support the achievable cost reduction at least to some extent, this statement must first be proven with numerical data obtained by examining a sufficient number of XP projects.

However, we can say that one of the advantages of XP is that, as a result of the development of automated tests, continuous integration, pair programming, short iteration cycles, and permanent feedback with the customer, the probability of finding errors at an early stage has increased.

Relationship between XP and CMM

In the article [Pau01], one of the authors of the Capability Maturity Model (CMM) asserts that, rather than being incompatible contradictions, XP and the software CMM [PWC+95] actually complement each other. Accordingly, XP has good development practices that satisfy core requirements of CMM. For many of the "key process areas" (KPA) demanded by CMM for the five CMM levels, XP offers practices which, although to a certain extent unconventional, are well-suited to the project area covered by the XP approach. These practices address the goals of CMM. The article [Pau01] breaks down the support provided by XP for the KPAs individually in a table. Of the total number of eighteen KPAs, seven are rated as negative and eleven are rated as positive to very positive. Of the KPAs rated as negative, however, the training program can also be rated as more positive due to the pair programming performed by experts with beginners, and the management of subcontracts can be ignored. In agreement, [Gla01] describes how the XP approach adheres to CMM Level 2 without any additional effort and indicates that the project-based part of CMM Level 3 can also be achieved with little additional effort. However, full CMM Level 3 requires cross-project measures across the company which are not addressed by XP.

In summary, [Pau01] comes to the understandable conclusion that XP has good techniques that companies could consider, but that very critical systems should not be realized exclusively with XP techniques.

Findings from Experiences with XP

XP provides a lightweight process comprised of coherent techniques. As XP focuses on the code, the process can be designed much more efficiently than RUP, for example. This efficiency means that fewer resources are required and the process is more flexible. The increased flexibility mitigates one of the basic problems of software development, namely handling user requirements that change over the duration of the project.

Ensuring the quality of the product being developed calls for pair programming and rigorous automated tests without, however, relying to much on the test theory which has been around for a long time. The constant demand for simplicity increases the quality and efficiency further.

Based on this analysis of XP, which stems partly from literature and partly from own project experiences, including the auction project described in Appendix D, Volume 1, we can categorize XP as an approach suitable for small projects with approximately 3–15 people. To a limited extent, using suitable measures, such as the hierarchical decomposition of larger projects based on components [JR01, Hes01] and the accompanying additional activities, XP can be scaled to larger tasks.

Alternatively, it may also be possible to reduce the size of the task using innovative technologies, including reusing and adapting an existing system where this is possible.

In particular, reducing the size of the task involves the improvement of used languages, tools, and class libraries. Many parts of a program that have technical code, such as the output, storage, or communication of data, have similar structures. If these program parts can be generated and, using an abstract representation of the application, composed to form executable code, this increases the efficiency of the developers even further. This is true for the product code but even more so for the development of tests which, in abstract modeling, are also made more understandable with diagrams.

Accordingly, the goal of using an executable sublanguage of UML as a high-level programming language must be to increase the efficiency of the process of developing models and transforming them into code, thereby accelerating the software development process further. Ideally, the design and implementation part of a project is reduced to such an extent that a project consists primarily of eliciting requirements that can be implemented efficiently and directly.

2.3 Selected Development Practices

Three of the development practices of XP are being investigated further: pair programming, the test-first approach, and the evolution of code. This is because they also play a significant role in agile modeling with UML.

2.3.1 Pair Programming

Pair programming existed before XP and was described in [Con95], for example. Although it was initially a stand-alone technique, today it is integrated in XP (as well as other techniques) because it is a good basis for common code ownership. It means that for all parts of the system, there are at least two people who are familiar with it.

The main idea behind pair programming is also referred to as the “principle of dual control”: two developers work on one task which they solve together. They need only one computer to do so, and while one developer

types in what they have developed, the partner performs a constructive review at the same time. However, the keyboard and the control over the constructive work quickly alternate between the two parties involved. Originally, this principle was intended for use by developers with the same level of expertise; however, it is also suitable for a combination of a system expert and a project beginner. It allows the beginners to familiarize themselves with existing software structures and new techniques efficiently. In purely mathematical terms, however, pair programming initially means double the personnel expense.

In tests conducted at universities, pair programming has been studied in more detail and analysis schemes have been created [SSSH01]. The studies [WKCJ00] and [CW01] show that, for pair programming, after a relatively short time for familiarization with the new programming style, the total expense compared to programming by individuals had increased but there was a significant reduction in the duration of the project and in particular, a significant increase in the quality of the software.[2] However, it is also evident that the technique of programming in pairs has to be learned and that pair programming does not suit everyone.

Therefore, in practice rigorously enforced pair programming would not be productive. Instead, cooperatively encouraging pair programming should lead to optimal results, because among others a project also involves activities that do not need to be performed in pairs. These activities include planning activities, tool installation and maintenance, as well as (in some circumstances) discussions with customers to elicit requirements. Unfortunately, flexible working hours for developers and unfavorable room allocations are further obstacles to applying pair programming consistently.

2.3.2 Test-First Approach

Tests are performed at different points in time and discussed and used with differing intensity in different methodologies. For example: the V-Modell XT explicitly separates tests for methods, classes, subsystems, and the overall system. In the Rational Unified Process according to [Kru03], however, there is no differentiation between the test levels and no discussion of metrics for test coverage. In XP, testing is one of the four core activities—it plays a significant role and is therefore discussed in more detail here.

[Bec01] describes the advantages of the test-first approach propagated in XP for software development very clearly. [LF02] elaborates on this approach for describing unit tests and discusses the advantages and disadvantages as well as the methodological use in a pragmatic form. [PP02] compares the test-first approach with the traditional creation of tests after implementation in a

[2] Statistically meaningful example figures are given in [WKCJ00] and [CW01], showing that when pair programming was used, development costs rose by 15% but the resulting code had 15% fewer defects. This allows a reduction in costs for bug fixing. The total cost saving is estimated at between 15% and 60%.

general context. [Wak02, p. 8] contains a description of a micro development cycle based on tests and coding.

The main idea of the test-first approach is to think about test cases before developing the actual functionality (in particular, individual methods). These test cases must be suitable for checking that the functionality to be realized is correct in order to describe this functionality in the form of an example. The test cases are documented in tests that run automatically. According to [Bec01] and [LF02], this has a number of advantages:

- Defining test cases before the actual implementation allows an explicit demarcation of the functionality to be realized, meaning that the test design equates de facto to a specification.
- The test justifies the necessity of the code and describes, amongst other things, which parameters are required for the function that is to be realized. This means that the code is designed such that it *can be tested*.
- Once the functionality has been realized, the existing test cases can be used for immediate verification; this increases the confidence in the code developed enormously. Although there is no guarantee that the functionality is free of errors, practical application, including in the auction project, shows that this confidence is justified.
- It is easier to separate the logical design from the implementation. When test cases are defined—in this case before the implementation—the first step is to determine the signature of the new functionality. This signature contains the name, parameters, and the classes, which contain methods. The methods are not implemented until the next step, that is, after the definition of the tests.
- The amount of work involved in formulating test cases should generally not be forgotten, especially if complex test data is required. According to [Bec01], this leads to functions being defined in such a way that they are provided with only the data necessary in each case. This results in classes being decoupled, making the designs better and more simple.
  This argument may be true in individual cases but it may not always be correct. It would be more correct to state that classes are decoupled as a result of the early detection of the possibility for decoupling or due to retrospective refactoring. This is also demonstrated by typical examples of the test-first approach [LF02, Bec01].
- Early definition of sets of test data and signatures is also helpful in pair programming as it allows developers to discuss the desired functionality more explicitly.

One advantage of test cases is that other developers can recognize the desired functionality based on the test case descriptions. This is e.g. necessary if the code is unclear and does not have sufficient comments and there is no explicit specification of the functionality. The test cases themselves therefore represent a model for the system.

Experience shows, however, that the possibility of developing the functionality based on the tests is limited. This is because tests are usually defined less thoroughly than the actual code. Also, tests written in a programming language do not represent the actual test data very compactly or clearly.

Although the test-first approach offers a number of advantages, in practice we must still assume that defining tests in advance does not provide sufficient coverage for implementation. However, the coverage metrics that we know from the testing theory are not explicit part of XP. A very informal concept for test coverage based in particular on the intuition of the developers is generally seems satisfactory. On the one hand, this is somehow unsatisfactory for the controlling in a project, but on the other hand, it is successful in practice. But there are also tools which automatically measure the extent to which the tests detect local modifications in the code and therefore satisfy certain coverage criteria. Some are even developed or adapted for the XP approach [Moo01]. These tools include mutation tests [Voa95, KCM00, Moo01] that through simple mutation of the test object check whether a test detects the modification (defect).

If the desired functionality is implemented, once the initial test suite has been completed, the development of further tests should achieve a better coverage. These further tests include, for example, the treatment of borderline cases and the investigation of conditional expressions and loops that can be necessary in part due to the technique or framework used and therefore were not anticipated in the test cases developed in advance.

The test-first approach is generally seen as an activity which combines analysis, design, and implementation in "microcycles". However, [Bec01] also refers to the fact that test methods which are used to define test cases systematically and which measure the coverage of the code according to different criteria are generally ignored. This is consciously accepted with the argument that these test methods involve a lot more effort but the results are not (if at all) significantly better. [LF02, p. 59] at least points out that tests are created not only before implementation but also after completion of a task in order to achieve "sufficient" coverage, but does not explain precisely when the coverage is sufficient.

Depending on the type and size of the project, the strict test-first approach can be an interesting element of the software development process which can typically be used after at least an initial architecture has been modeled for the system and the system has been broken down into subsystems. By using the executable sublanguage UML/P, this approach can be elevated to the modeling level more or less unchanged. For this purpose, in a first step, sample data and sample sequences can be modeled as test cases using object diagrams and sequence diagrams respectively. Based on these test cases, the functionality to be realized can be modeled using Statecharts and class diagrams.

2.3.3 Refactoring

The desire for techniques which incrementally improve and modify source code using sets of rules is only a slightly more recent phenomenon than the creation of the first programming languages [BBB+85, Dij76]. The goal of transformational software development is to break the software development process down into small, systematically applicable steps which are manageable due to their effects being limited to the local environment. Refactoring was first discussed in [Opd92] for class diagrams. [Fow99] is highly recommended. It describes an extensive collection of transformation techniques for the programming language Java. The refactoring techniques allow us to migrate code in line with a class hierarchy. They also allow us to break down or divide classes, shift attributes, expand or outsource parts of code into separate methods, and much more. The strength of the refactoring techniques is based on how easy it is to manage the individual transformation steps (referred to as "mechanisms") and the ability to combine them flexibly, which leads to large, goal-oriented improvements in the software structure.

The goal of refactoring is to transform an existing program but preserve the semantics. Refactoring is used to improve the quality of the design whilst retaining the functionality rather than to extend the functionality. It therefore supplements the normal programming activity.

Refactoring and the evolution of functionality are complementary activities that can quickly alternate in the development process. The course of a project can thus be outlined as shown in Fig. 2.5. However, there is no objective criterion for measuring the "quality of the design". Initial approaches, for example, measure the conformance to coding standards, but are insufficient for evaluating the maintainability and testability of the architecture and implementation.

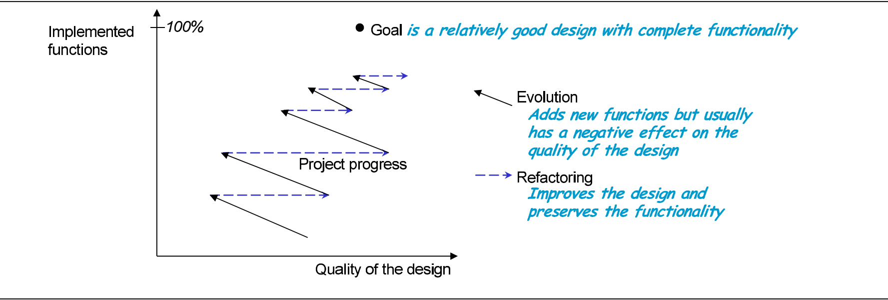


Fig. 2.5. Refactoring and evolution are complementary

No verification techniques are used to ensure that transformations which preserve semantics are correct; instead, the existing test suite is used. If we can assume that the existing test suite has a high level of quality, there is

a high probability that faulty modifications, that is, modifications which change the behavior of the system, will be detected. Refactoring often modifies internal signatures of a subsystem if, for example, a parameter is added to a method. Therefore, certain tests have to be adapted together with the code. In the sense of the test-first approach, [Pip02] even proposes refactoring first the tests and then the code.

Refactoring techniques are aimed at various levels of the system. Some refactoring rules have a small effect; others are suitable for modifying an entire system architecture. The possibility of improving a system architecture that has already been realized in code means that it has become less necessary to define a correct and stable system architecture a priori. Modifications to the system architecture of course involve high costs. In XP, however, the assumption is that maintaining system functions which are not used is more cost-intensive over the long term. In accordance with the principle of *simplicity*, the XP approach prefers to keep the system architecture simple and to only modify or extend it as required using suitable refactoring steps.

While many refactoring techniques for Java based on [Fow99] are currently under ongoing development, at present only a few similar techniques exist for UML diagrams [SPTJ01].

2.4 Agile UML-Based Approach

Due to the diversity of software development projects in terms of size, application domain, criticality, context etc., we can conclude that a standardized approach does not and will exist in the diversified project landscape.

Instead, as proposed in Crystal [Coc06], a suitable approach must be selected (and adapted) where necessary from a collection of approaches based on the following criteria: the size and type of project, the criticality of the application, as well as the experience and knowledge of the people involved in the project.

Corresponding to this diversity, this book outlines a proposal for a lightweight, agile method using UML. This proposal concentrates in particular on the technical part of an approach and does not claim to be suitable for all types of projects.

Definition of "Agility"

The characterization of the agility of a method is outlined in Fig. 2.6.

Efficiency can be improved by cleverly omitting unnecessary work (documentation, functionality not required, etc.) and also through an efficient implementation.

It is not necessarily the case that all of the techniques of an agile method focus on the *quality* of the product developed. However, certain elements of an agile method, such as concentrating on the simplicity of the design and

An approach is deemed to be "agile" if it emphasizes the following criteria:

- The *efficiency* of the overall process and the individual steps is as ideal as possible.
- The *reactivity*, that is, the speed at which the approach reacts to changing requirements or a different project environment, is high. Planning therefore tends to be *short-term and adaptable*.
- The approach itself can also be *adapted flexibly* so that it can adapt itself dynamically to internal project circumstances, as well as to external project circumstances which are determined by the environment and which therefore can only be partially controlled.
- *Simplicity* and practical implementation of the development approach and its techniques lead to simple design and implementation.
- The approach is *customer-oriented* and demands active integration of the customer during the project.
- The *capabilities, knowledge, and needs of the project participants* are accounted for in the project.

Fig. 2.6. Terminology: agility of a method

the extensive development of automated tests, do support the improvement in the quality. Other practices enforced by some agile methods, such as the lack of detailed specifications and reviews, discourage using the method for highly critical systems. In this case, an alternative process or a suitable extension must be selected.

Improved Support for Agility

The agility of a project can be improved not only by modifying the activities but also in particular by increasing the *efficiency* of the developers. It is of interest to improve the efficiency of creating models, the implementation, and the tests. This is particularly effective if the implementation and the tests can be derived as efficiently as possible or even completely automatically from models. This type of automatic generation is interesting if the source language used for the models allows a more compact representation than the implementation itself could provide.

The premise for using models in this way is that they can be created more quickly because they are more compact but still allow a complete description of the system. In the form offered in the UML standard [OMG10], UML is not sufficient for this purpose. Therefore, UML/P has been extended to include a complete programming language.

Use Cases for Models

Models that are used to generate code require a high level of detail. But then the usual coding activity, which is very time-consuming, is no longer necessary. The detailed modeling and the implementation merge into one single

activity. Code generation is therefore an important tool for using models successfully. This allows us to achieve a goal similar to XP, in which design and modeling activities are generally executed directly in code.

However, we can use models for other goals besides code generation. *Abstract* or relatively *informal* models suffice for the communication between developers. *Abstraction* means that details that are not necessary for describing the communicated content can be omitted. *Informality* means that the language correctness of the model represented does not have to be adhered to precisely. For example, diagrams on paper can use intuitive notational elements that do not belong to the language if the developers have a common understanding of their meaning as far as necessary. Furthermore, where *informal* diagrams are used, they do not have to be completely consistent with the content modeled or with other diagrams.

We can also use models to document the system. Documentation is a written form of communication intended for long-term use. A higher level of detail, greater formality, and/or consistency are therefore useful depending on the goal of the documentation. A short document that gives an overview of the architecture of a system and an introduction to important decisions that led to this architecture will have a low level of detail, but the formality and consistency within the model and with the implementation are important. If complete documentation is required, the ideal way to ensure that the documentation is consistent with the system implemented is to generate the implementation and, as far as possible, tests, from the models of the documentation.

In a project that uses UML/P for modeling and for implementation, we at least differentiate between:

- *Implementation models*
- *Test models*
- *Models for documentation*
- *Models for communication*

We can use UML/P for all of these purposes even though the models each have different characteristics. However, there is a significant advantage in the fact that there are no notational breaches between specification, implementation, and test cases. UML/P can be used for detailed as well as abstract and incomplete modeling.

Modifying Models

If a model is used exclusively for communication, then it normally exists only as an informal drawing. A model created by a tool has a formal representation of the syntax[3] and can therefore be used for further processing

[3] For example, *XML Metadata Interchange Format* (XMI) is a standard for representing UML models and therefore this type of formal representation of the syntax.

supported by tools. The syntax of UML/P is therefore precisely defined in Appendices A, Volume 1, B, Volume 1, and C, Volume 1.

We have already identified *code generation* as one of the important techniques for using models. There are two main variants of code generation: product code generation and test code generation.

The necessity of adapting software based on changing requirements also means that techniques must be available for *transforming* or *refactoring* models. Systematically adapting models to new and changed functionalities in a process validated at each stage by automated tests and invariants allows this dynamic adjustment which is already familiar from XP. As a model created according to given requirements is more compact and therefore easier to understand than the code, this also improves adaptability of models. For example, we can adapt the structure within one class diagram, which is spread across a number of files in the code. The necessity of developing a fixed architecture at an early stage of a project decreases very similar to the observations made in the XP approach. At the same time, the flexibility for integrating new functionality and therefore the agility of the project continues to increase.

We can therefore identify the following important techniques in handling models [Rum02]:

- Generating product code
- Generating test code
- Refactoring models

Furthermore, various techniques for analyzing models are interesting, such as the reachability of states in Statecharts. Another factor that is helpful in creating tests efficiently is deriving many test cases from universal specifications, such as OCL constraints or Statecharts.

Special techniques, for example, for generating database tables, generating a simple graphical interface from data models, or for migrating data sets conforming to the old model into a new model are also important and must be supported by a code generator with suitable parameters. However, this book does not cover such techniques in any further detail.

Further Useful Principles and Practices

To complete an approach, further principles and practices have to be selected on a project-specific basis. For small projects, for example, we can identify the following principles and practices, many of them adaptable from XP:

- Many small iterations and releases
- Simplicity
- Quick feedback
- Permanent dialog with a customer who is constantly available
- Short internal, daily meetings for coordination
- Review after each iteration for the purpose of process optimization

- Development of tests before implementation ("test-first")
- Pair programming as a technique for learning and review
- Common ownership for models
- Continuous integration
- Modeling standards as a requirement for the form and commenting of the models similar to coding standards

We can very easily apply the development of tests before an implementation in accordance with the test-first approach discussed in Section 2.3.2 to UML/P. To do so, we initially model sequence diagrams, which are an essential component of tests, as descriptions for use cases and we can then transform them into test cases with an acceptable level of effort. A similar situation applies for object diagrams, which we can create as examples of data sets during the recording of requirements.

Standard modeling guidelines applicable across the project are necessary to ensure the usability of the models; the project participants must be able to adapt and extend these models as necessary.

Sometimes developers still need to learn the language UML/P for modeling to become productive. Any necessary learning phase can be supported by pair programming, which will now generally be referred to as "pair modeling" in this book.

Quality, Resources, and Project Goals

The effects of the approach outlined in this section can be explained using the value system discussed in Section 2.2 and the four criteria (time consumption, costs, quality, and goal orientation) also used in XP, for example.

Communication is easier to accomplish based on UML/P models than based on implemented code if we can assume that the communication partners are familiar with UML.

Simplicity of the software is better supported for two reasons: on the one hand, as the software can be changed easily, it is even less important to establish structures for future functionality that might potentially be required and thus integrate potentially unused complexity at an early stage. On the other hand, it is even easier to remove elements that are no longer necessary from the model using refactoring steps. However, the developers themselves must still be able to recognize unnecessary complexity and superfluous functionality.[4]

Feedback is ensured by the greater efficiency in development and the resulting even shorter iterations.

Individual responsibility and courage can and must remain with the developers, as in XP.

[4] Analysis tools can only provide limited support here by recognizing which methods, parameters, or classes are not required.

The four main variables of project management are also addressed:

Time consumption: Time savings and, in particular, early availability of first usable versions ("time to market") are essential not only for Internet applications. The increased development efficiency and the higher level of reusability of technical parts allow a reduction of the time necessary.

Costs: Costs also decline due to increased efficiency and the resulting decrease in time and personnel expense. This reduction in the personnel expense means that some management efforts can also be omitted, meaning that the process used is more lightweight and thus enables further savings.

Quality: The quality of the product is influenced positively by the following factors: the more compact and therefore clearer representation of the system; the easier modeling of test cases; and through the fact that there is no longer a breach between the modeling and implementation language. Concentrating on further aspects, such as the level of test coverage, additional model reviews, and actively integrating the customer determines whether the quality of the resulting system meets the demands.

Goal orientation: In order to ensure that the system is implemented in the form desired by the user, it is important to integrate the user actively. Using UML/P does not influence this aspect in any direction, as we can assume that it is unlikely that a user would be presented with UML diagrams for discussion.

The Problems of Agile, UML-Based Software Development

In addition to the advantages discussed above, the approach outlined has some disadvantages that should be considered:

- The advantage that almost the same notation can be used for the abstract modeling and implementation can prove to be a blowback. In practice, previous approaches for an abstract modeling of essential properties, such as SDL [IT07b, IT07a] or algebraic specifications [EM85, BFG+93], have been used often as high-level programming languages as soon as their executability has been available through a tool. The same is now proposed for a sublanguage of UML.[5] If UML is used for specification, this can lead to unnecessary details being filled out for the specification because a later use of the specification as an implementation has been anticipated to early. This phenomenon is also referred to as "overengineering".
- The teaching effort for using UML/P must be assumed as high. With regard to syntax, UML/P is significantly more complex than Java, meaning

[5] In algebraic specifications, this has led at least in part to a greater focus on transformability into efficient code rather than on the abstract modeling of properties and thus led to strange implementation-oriented specifications.

that an incremental learning approach is recommended here. As an initial step, the use of UML/P for describing structures with class and object diagrams can be taught. This can be followed by sequence diagrams for modeling test cases and, building on that, OCL for defining conditions, invariants, and method specifications. Using Statecharts for modeling behavior usually requires the most practice.
However, similarly to Java, it is not only the syntax that needs to be mastered but in particular the use of modeling standards and design patterns, which must also be learned.

- At present, there is not enough tool support that covers all aspects of the desired code generation completely. In particular, efficient code generation is essential for creating the system quickly, which in turn enables efficient execution of tests and the resulting feedback.
  However, a number of tool developers are working on realizing this vision of complete tool support and are already in a position to demonstrate results.

2.5 Summary

Agile methods represent a relatively new approach which, through several characteristics, distinguishes itself explicitly from methods previously used in software development. The tools, techniques, and the understanding for the problems of software development have improved significantly. Therefore, these methods can offer more efficient and more flexible approaches for a subdomain of software development projects through, for example, strengthening the focus on the primary result, the executable system, and a reduction in the secondary activities. At the same time, the motivation and the commitment of the project participants for rigorous execution of their activities come to the fore and short iterations enable flexible, situation-dependent control of the software development.

We have identified important factors for determining the size of a project: the *size of the problem* and the *efficiency of the developers*. We can use these factors to derive the required *method size*, which consists mainly of the methodological elements to be executed, the formality required for the documents, and the additional effort for communication and management. Due to additional communication and management overheads, the *efficiency of the developers* is disproportionate in the determination of the *project size*, i.e., the number of developers required and the runtime of the project. Therefore, improving developer efficiency is a significant lever for reducing development costs.

As a modeling language for *architecture modeling*, *design*, and *implementation*, UML/P provides a standardized language framework that allows us to model the executable system and the tests completely. The compact size of the representation leads to greater efficiency and results in scaling effects for

the project size. The standardized framework prevents an otherwise often observed notational breach between the modeling, implementation, and test languages.

3

Compact Overview of UML/P

The limits of my language
are the limits of my world.

Ludwig Wittgenstein

This chapter contains a compact summary of the language profile UML/P which is introduced in Volume 1 [Rum16]. It describes some but not all of the special features and deviations of the UML/P profile compared to the UML 2 standard. For a more detailed description, see Volume 1. Readers already familiar with UML can use this chapter mainly for reference purposes when required. The examples used in this chapter to introduce the language refer primarily to the auction system application described in Volume 1.

3.1 Class Diagrams **34**
3.2 Object Constraint Language **39**
3.3 Object Diagrams **51**
3.4 Statecharts **56**
3.5 Sequence Diagrams **65**

© Springer International Publishing AG 2017
B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_3

3.1 Class Diagrams

Class diagrams are the architectural backbone of many system developments. Accordingly, class diagrams and, in particular the classes within those diagrams, have a number of roles to fulfill (Fig. 3.1): class diagrams describe the structure or rather the architecture of a system and are thus the basis for almost all other description techniques. The *class* concept is used universally in modeling and programming; it therefore offers a backbone that enables us to trace requirements and errors through the different activities of a project. Class diagrams form the skeleton for almost all other notations and types of diagrams as these are based respectively on the classes and methods defined in class diagrams. Thus, for analysis purposes, we use class diagrams to structure concepts of the real world, whereas in design and implementation, we use class diagrams primarily to represent a structural view of a software system.

The features and roles of a class are:

- Encapsulation of attributes and methods into a conceptual unit
- Representation of instances as objects
- Typing for objects
- Description of the implementation
- Bytecode (the compiled, executable form of the description of the implementation)
- Extension (set of all objects existing at a point in time)
- Characterization of the possible structures of a system

Fig. 3.1. The variety of features and tasks of a class

3.1.1 Classes and Inheritance

Fig. 3.2 presents the most important terminology for class diagrams.

Fig. 3.3 shows a simple class diagram consisting of one class. Depending on the level of detail required, attributes and methods of the classes may be omitted or only partially specified in a class diagram. The types of attributes and methods, visibilities, and other modifiers are also optional.

We use the inheritance relationship to structure classes in manageable hierarchies. Fig. 3.4 demonstrates inheritance and interface implementation using the common features of multiple output types of the auction system described in Chapter D, Volume 1. The diagram does not show interface extension which would look like inheritance between classes.

Class: A class consists of a collection of attributes and methods which define the state and behavior of the *instances* (*objects*) of the class. Classes are connected to one another by associations and inheritance relationships. A *class name* identifies the class.

Attribute: The state of a class is defined through its attributes. An attribute is defined by its *name* and *type*.

Method: The functionality of a class is organized in methods, which consist of a *signature* and a *body*. The body describes the implementation. An *abstract* method has no body.

Modifier: The modifiers `public`, `protected`, `private`, `readonly`, `abstract`, `static`, and `final` can be applied to classes, methods, and attributes to define the visibility, instantiability, and changeability of the modified element. For the first four of these modifiers, UML/P contains the graphical variants "`+`", "`#`", "`-`", and "`?`".

Constants: Constants are defined as special attributes with the modifiers `static` and `final`.

Inheritance: If two classes are in an inheritance relationship with one another, the *superclass* passes its attributes and methods to the *subclass*. The subclass can add further attributes and methods and can *redefine* methods—provided the modifiers allow this. The subclass forms a *subtype* of the superclass; in accordance with the *substitution principle*, instances of the subclass may be used where instances of the superclass are required.

Interface: An interface describes the signatures of a collection of methods. In contrast to a class, no attributes (only constants) and no method bodies are specified. Interfaces are related to abstract classes and can also be in an inheritance relationship with one another.

Type: A type is a primitive data type (such as `int`), a class, or an interface.

Interface implementation: An interface implementation is a relationship, similar to inheritance, between an interface and a class. A class can implement any number of interfaces.

Association: An association is a binary relationship between classes, which we use to realize structural information. An association has an *association name*, a *role name* for each end, a *multiplicity*, and a *navigation direction*.

Multiplicity: The multiplicity is specified for each association end. It is of the form "`0..1`", "`1`", or "`*`" and describes whether an association in the specified direction is optional, unique, or permits multiple links.

Fig. 3.2. Terminology definitions for class diagrams

3.1.2 Associations

An association places objects of two classes in a relationship with one another. We can use associations to portray complex data structures and to call methods of neighboring objects. Fig. 3.5 shows an excerpt from the auction system with multiple associations in different forms.

An association usually has an *association name*, one *association role* respectively for each of the two ends, a *multiplicity* specification, and a description of the possible *navigation directions*. Individual details can also be omitted

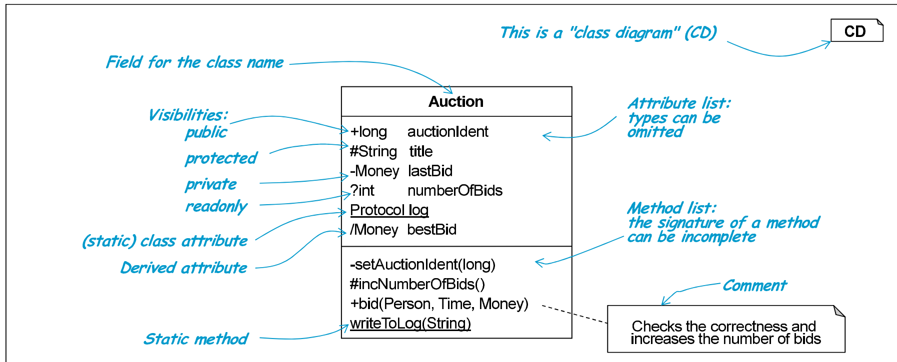


Fig. 3.3. A class in a class diagram

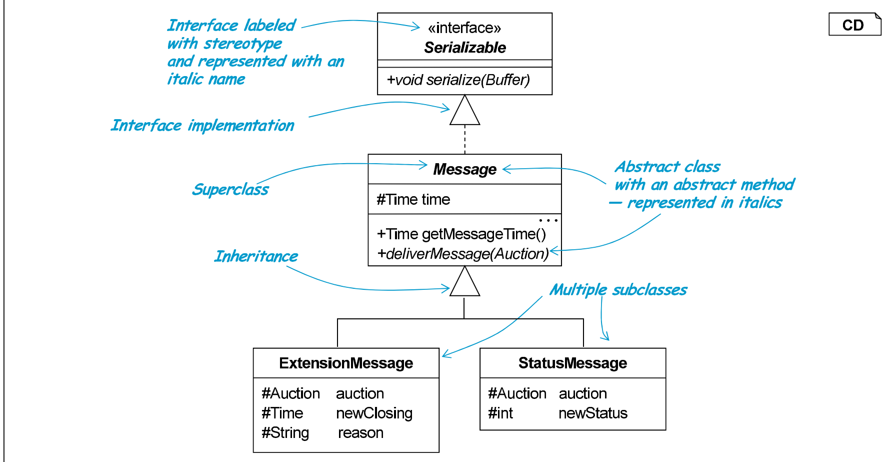


Fig. 3.4. Inheritance and interface implementation

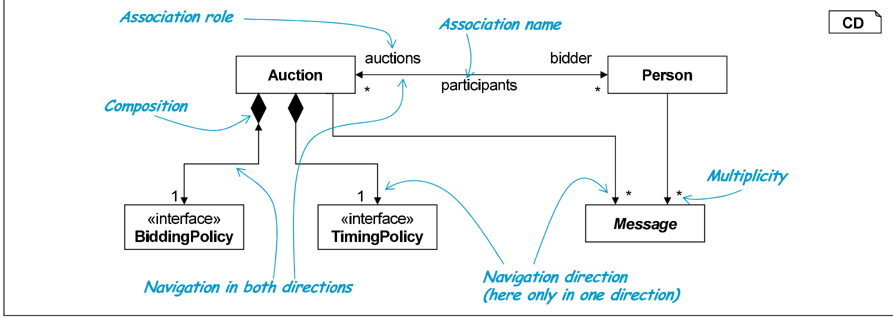


Fig. 3.5. A class diagram with associations

from the model if they are not important for representing the desired facts and unambiguity is not lost by omitting them.

Associations can be unidirectional or bidirectional. If no explicit arrow direction is specified, the association is assumed to be bidirectional. From a formal perspective, in this situation the navigation options are considered as unspecified and thus not constrained.

A *composition* is a special form of an association. It is represented by a solid diamond at one association end. In a composition, the *parts* are heavily dependent on the *whole*. The life cycles of all *parts* are also bound to the *whole* (i.e., the composition). Often the life cycles are identical.

The UML standard offers a number of additional tags for associations. These tags allow us to regulate the association properties more precisely. Fig. 3.6 shows some frequently used tags, such as `{ordered}`, which permits an ordered access using an index. The tag `{frozen}` specifies that once an auction object has been initialized, the two associations to the policy objects cannot be changed. The tag `{addOnly}` models the fact that objects may be added to the association but cannot be removed.

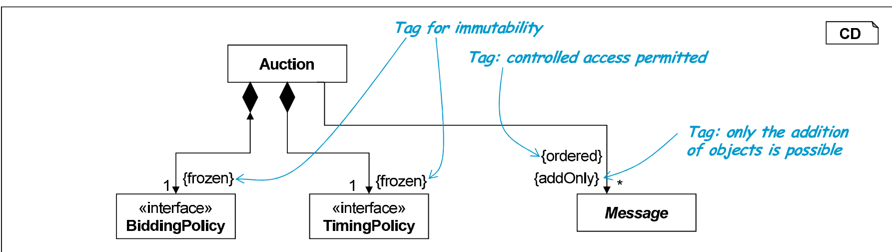


Fig. 3.6. Tags for associations

Qualified associations allow us to use a *qualifier* to select an individual object from a set of objects. Fig. 3.7 shows multiple associations that are qualified in different ways.

In explicitly qualified associations, we can select the type of qualifier; whereas in ordered associations, integer intervals beginning with `0` are used.

3.1.3 Representation and Stereotypes

The goal of class diagrams is often to describe the data structure required for a certain task, including the relationships within the data structure. For the purposes of an overview, a complete list of all methods and attributes is thus more of a hindrance. A class diagram therefore usually represents an incomplete *view* of an entire system; individual classes or associations may be missing. Within the classes, attributes and methods may be omitted or may be represented incompletely.

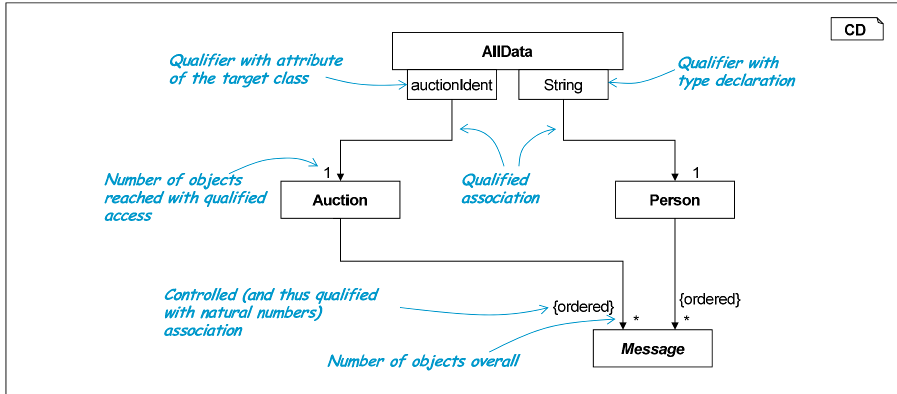


Fig. 3.7. Qualified associations

To enable developers to indicate whether the information contained in a UML diagram is complete, UML/P offers representation indicators: "..." indicates that information is incomplete and "©" indicates complete information (Fig. 3.8).

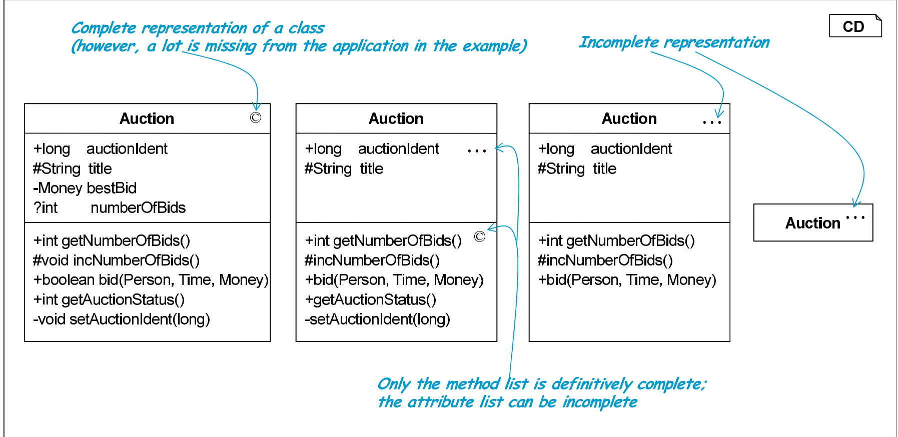


Fig. 3.8. Forms of representation of a class

The indicators "©" and "..." have no effect on the class itself, only on the representation of the class within a class diagram. A "©" in the class name states that both the attribute list and the method list are complete. In contrast, the incompleteness indicator "..." indicates that the representation *may* be incomplete.

Both representation indicators are merely a special form of tag with their own syntactical representation. In very general terms, UML offers the pos-

sibility of labeling model elements using stereotypes and tags (Fig. 3.9). The syntax of such a label is of the form «stereotype», {`tag`}, or {`tag=value`}.

Stereotype: A stereotype classifies model elements such as classes or attributes. It refines the meaning of the model element, allowing the element to be handled more specifically during code generation. A stereotype can have a number of tags.
Tag: A tag describes a property of a model element. It is denoted as a pair consisting of a *keyword* and a *value*. Multiple values can be grouped in a list.
Model elements: Model elements are the (main) components of UML diagrams. For example, the class diagram has the following model elements: classes, interfaces, attributes, methods, inheritance relationships, and associations.
Tags and stereotypes can be applied to model elements but are not model elements themselves.

Fig. 3.9. Terminology definitions: tag and stereotype

3.2 Object Constraint Language

The Object Constraint Language (OCL) is a property-oriented modeling language used for modeling invariants as well as preconditions and postconditions of methods. The OCL/P contained in UML/P is an extended variant of the OCL standard that has been adapted to Java from a syntax perspective.

3.2.1 OCL/P Overview

Fig. 3.10 explains the most important terms of OCL.

One of the outstanding features of OCL constraints is that they are embedded in a context which consists of UML models. The class diagram in Fig. 3.11 shows such a context in which, for instance, the following expression can be formulated:

OCL

```
context Auction a inv Bidders2:
  a.activeParticipants == { p in a.bidder | p.isActive }.size
```

Expression `Bidders2` is quantified over all objects `a` of the class `Auction`. It contains navigation expressions, such as `a.bidder`, which allow navigation via associations. The *set comprehension* {`p in ... | ...`} selects all active persons of an auction. Set comprehension is a comfortable extension compared to the OCL standard. It will be discussed later in more detail in multiple forms.

If only one class is given the context, the name `this` can be used instead of an explicit name. In this case, attributes without a qualification can also be accessed. *Closed conditions* have an empty context:

Condition: A condition is a Boolean expression applicable to a system. It describes a property that a system or a result should have. The interpretation of a condition always results in one of the logical values `true` or `false`.

Context: A condition is embedded in a context over which it defines a statement. The context is defined by a set of *names* and their signatures which can be used in the condition. The names include class names, method names, and attribute names of the model.

Interpretation: A condition is interpreted with respects to a concrete object structure. As part of the interpretation, the variables defined in the context are assigned values or objects of this object structure.

Invariant: An invariant describes a property that should be valid at any (observable) point in time during a system execution. The observation times can be restricted in order to permit time-restricted violations, for example, during the execution of a method.

Precondition: A precondition of a method characterizes the properties that must be valid for this method to deliver a defined and useful result. If the precondition is not satisfied, there is no information about the result given.

Postcondition: A postcondition of a method describes the properties that are valid after the execution of the method. In the postcondition objects can also be accessed in the state that was valid immediately before the method call (at the "time" of the interpretation of the precondition). Postconditions are thus logically interpreted using two object structures which represent the situations before and after the method call.

Method specification: A method specification is a pair consisting of a precondition and a postcondition.

Query: A query is a method offered by the implementation. Calling a query does not cause a change in the system state. Although, new objects may be created as a result of the call, they must not be connected to the system state by means of links. Queries therefore have no side effects and can be used in OCL constraints.

Fig. 3.10. Terminology definitions for OCL

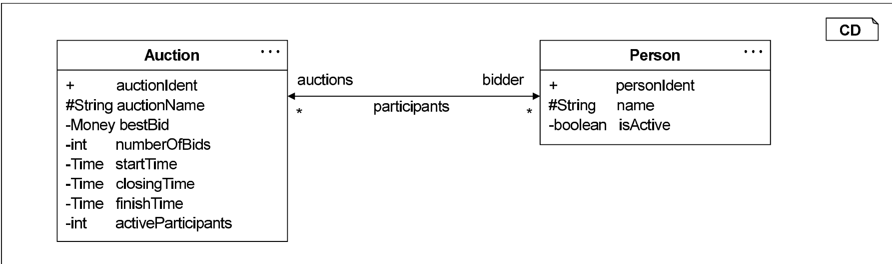


Fig. 3.11. Excerpt from the auction system

OCL
```
inv:
  forall a in Auction:
    a.startTime >= Time.now() implies
      a.numberOfBids == 0
```

We can use the **let** construct to assign intermediate results to an auxiliary variable, which can be used in the body of the construct in all places where needed:

OCL
```
context  inv SomeArithmeticTruth:
  let middle = (a+b)/2
  in
    a<=b  implies  a<=middle  and  middle<=b
```

The **let** construct can also be used to define auxiliary functions:

OCL
```
context Auction a inv Time2:
  let min(Time x, Time y) = (x<=y) ? x : y
  in
    min(a.startTime, min(a.closingTime,a.finishTime))
          == a.startTime
```

We have already seen the conditional expression `.?.:.`, which also can be used in form `if-then-else`. In contrast to an imperative conditional expression, the `then` and `else` branches must always be specified.

A special form of the conditional expression allows us to handle casts as they occasionally occur in subtype hierarchies. For this purpose, OCL/P offers a type-safe construction which represents a combination of a cast and a query regarding the convertibility of the cast:

OCL
```
context Message m inv:
  let Person p = (m instanceof BidMessage ? m.person : null)
  in ...
```

In addition to a normal conditional expression, in the `then` branch of the conditional expression, the type of the variable `m` is set to the subtype and thereby allows us to select `m.bidder`. The type safety is retained despite the conversion.

The primitive data types are the types familiar from Java: `boolean`, `char`, `int`, `long`, `float`, `byte`, `short`, and `double` and the operators that work on those types without any side effects. The increment operators `++` and `--` and all forms of assignment are thus excluded. New functions that are not known from Java are the Boolean operators `implies` and `<=>`, which are used to represent implications and equivalences, and the postfix operators `@pre` and `**`.

Just like in Java, the data type `String` is understood not as a primitive data type but as a class that is always available.

<table>
<tr><th>Priority</th><th>Operator</th><th>Associativity</th><th>Operand(s), meaning</th></tr>
<tr><td>14</td><td>@pre</td><td>left</td><td>Value of the expression in the precondition</td></tr>
<tr><td></td><td>**</td><td>left</td><td>Transitive closure of an association</td></tr>
<tr><td>13</td><td>+, -, ~</td><td>right</td><td>Numbers</td></tr>
<tr><td></td><td>!</td><td>right</td><td>Boolean: negation</td></tr>
<tr><td></td><td>(type)</td><td>right</td><td>Cast</td></tr>
<tr><td>12</td><td>*, /, %</td><td>left</td><td>Numbers</td></tr>
<tr><td>11</td><td>+, -</td><td>left</td><td>Numbers, String (+)</td></tr>
<tr><td>10</td><td><<, >>, >>></td><td>left</td><td>Shifts</td></tr>
<tr><td>9</td><td><, <=, >, >=</td><td>left</td><td>Comparisons</td></tr>
<tr><td></td><td>instanceof</td><td>left</td><td>Type comparison</td></tr>
<tr><td></td><td>in</td><td>left</td><td>Element of</td></tr>
<tr><td>8</td><td>==, !=</td><td>left</td><td>Comparisons</td></tr>
<tr><td>7</td><td>&</td><td>left</td><td>Numbers, Boolean: strict and</td></tr>
<tr><td>6</td><td>^</td><td>left</td><td>Numbers, Boolean: xor</td></tr>
<tr><td>5</td><td>|</td><td>left</td><td>Numbers, Boolean: strict or</td></tr>
<tr><td>4</td><td>&&</td><td>left</td><td>Boolean logic: and</td></tr>
<tr><td>3</td><td>||</td><td>left</td><td>Boolean logic: or</td></tr>
<tr><td>2.7</td><td>implies</td><td>left</td><td>Boolean logic: implies</td></tr>
<tr><td>2.3</td><td><=></td><td>left</td><td>Boolean logic: equivalent</td></tr>
<tr><td>2</td><td>? :</td><td>right</td><td>Expression selection (if-then-else)</td></tr>
</table>

Table 3.12. Priorities of the OCL operators

3.2.2 OCL Logic

The definition of the underlying logic for a constraint language needs to be appropriate for the language to be used in practice. Therefore, in OCL/P, we choose a two-valued logic and a special handling of the problematic "undefined" value. A detailed discussion of these aspects can be found in Volume 1.

Mapping the undefined value to the logical value `false` is the most elegant solution because this gives rise to a two-valued logic. This simplifies both specifications and reasoning, because there is no third case to be dealt with. Unfortunately, these semantics, which are so convenient for the specification, cannot be implemented completely because we would then have to determine whether a calculation does not halt and output the value `false`.[1] Nevertheless, the selected solution is feasible (see Volume 1).

Fig. 3.13 shows the truth tables of the OCL operators with this implicit mapping of `undef` to `false`.

3.2.3 Container Data Structures

Containers are the basis for the navigation via associations. Starting with one single object, a navigation expression allows us to describe a *set* or a *list* of

[1] However, the halting problem is undecidable.

!a	
!true	false
!false	true
!undef	true

a ^ b	true	false	undef
true	false	true	true
false	true	false	false
undef	true	false	false

a && b	true	false	undef
true	true	false	false
false	false	false	false
undef	false	false	false

a || b	true	false	undef
true	true	true	true
false	true	false	false
undef	true	false	false

a implies b	true	false	undef
true	true	false	false
false	true	true	true
undef	true	true	true

a <=> b	true	false	undef
true	true	false	false
false	false	true	true
undef	false	true	true

Fig. 3.13. Interpretations of the Boolean operators

reachable objects and to define certain properties for these objects. Like Java generics, OCL/P offers three *type constructors* for containers (Fig. 3.14).

Set<X> describes a set over the data type X. The usual operators such as disjunction or addition are available for sets. We can use any primitive data type, any class, and any container type for the type X. For comparisons of sets, we compare elements by value for primitive data types and by object identity for classes. However, objects from certain classes such as String redefine the comparison equals offering a value comparison.

List<X> describes ordered lists and the operations useful for such lists. List<X> allows us to manage the objects of this list, starting with the index 0.

Collection<X> is a supertype for the two named types Set<X> and List<X>. It provides the common functionality of these two types.

Fig. 3.14. OCL type constructors

A comparison of containers requires a binary operation that tests the equality of the elements. If the elements are primitive data types or also containers, we use the comparison == for the elements. However, if the elements are objects, we use the comparison equals. This corresponds to a comparison by value for primitive data types and containers and (in most cases) a comparison of the identities for objects. However, for special object types, such as String, the method equals is redefined, providing also a comparison by value.

Like Java generics, the subtype relationship of Set<X> and List<X> to Collection<X> allows us to use values of the types Set<X> or List<X> instead of values of the type Collection<X> for any type X.

The notation of sets and lists is based primarily on the extension of classes, i.e. the sets of their objects, set comprehensions, and navigation expressions. Some examples:

```
Auction.size < 1700;
Set{};
Set{"text",""} == {"text",""};
List{8,5,8};
List{'a'..'c'} == List{'a','b','c'};
```

OCL

Set and List Comprehensions

Compared to the OCL standard, OCL/P offers an extensive collection of options commonly provided by functional programming languages for describing sets and lists in enumerations as well as comprehensions based on properties.

The general syntax of a list comprehension is of the following form:

```
List{ expr | characterization }
```

OCL

Here, in the *expression characterization* (right), we define new variables which we can use in the expression (left). The characterization can consist of multiple *generators*, *filters*, and *local variable definitions* separated by commas.

A *generator* `v in list` allows a new variable `v` to iterate over all elements of the `list`. This allows us, for example, to define square numbers as follows:

```
inv:
  List{ x*x | x in List{1..5} } == List{1,4,9,16,25}
```

OCL

A *filter* describes a restriction on a list of elements. It evaluates to a logical value which decides whether an element is included in a list. In combination with a generator, we can therefore describe filters for partial lists:

OCL

```
inv:
  List{ x*x | x in List{1..8}, !even(x) } == List{1,9,25,49}
```

To further improve convenience, we can calculate intermediate results and assign them to local variables.

```
inv:
  List{ y | x in List{1..8}, int y = x*x, !even(y) } ==
      List{1,9,25,49}
```

OCL

Container Operations

Sets, lists, and containers provide the operations listed in Fig. 3.15, 3.16, and 3.17. Their signatures represent a combination of the functionality familiar from Java sets and the functionality offered by the OCL standard. In OCL, we can write the operations `size`, `isEmpty`, and `asList` as attributes without

```
Set<X>  add(X o);                                    Signature
Set<X>  addAll(Collection<X> c);
boolean contains(X o);
boolean containsAll(Collection<X> c);
int     count(X o);
boolean isEmpty;
Set<X>  remove(X o);
Set<X>  removeAll(Collection<X> c);
Set<X>  retainAll(Collection<X> c);
Set<X>  symmetricDifference(Set<X> s);
int     size;
X       flatten;                  // when X is a collection type
List<X> asList;
```

Fig. 3.15. Signature of sets of the type `Set<X>`

brackets, because a query without any arguments can generally be treated like an attribute. We can also write the operations as a query with brackets due to the compatibility with Java.

In contrast to a Java implementation, the concept of exceptions is not available for OCL expressions. Instead, all operators of OCL are *robust*, which means that the interpretation of these operators always produces meaningful results.

Since containers do not have an object identity in OCL, both equality operators == and `equals` are identical on containers:

```
context Set<X> sa, Set<X> sb inv:                          OCL
    sa.equals(sb) <=> sa==sb
```

As already discussed, the comparison of sets uses the comparison of the elements `equals` for objects and == for primitive data types. In OCL, == for containers of objects is dependent on the freely definable equality `equals` on the elements and differs from the comparison in Java.

Like in Java indexing of list elements begins with 0:

```
List{0,1,2}.add(3)              == List{0,1,2,3};            OCL
List{'a','b','c'}.add(1,'d')    == List{'a','d','b','c'};
List{0,1,2}.prepend(3)          == List{3,0,1,2};
List{0,1,2}.set(1,3)            == List{0,3,2};
List{0,1,2}.get(1)              == 1;
List{0,1,2}.first               == 0;
List{0,1,2}.last                == 2;
List{0,1,2}.rest                == List{1,2};
List{0,1,2,1}.remove(1)         == List{0,2};
List{0,1,2,3}.removeAtIndex(1)  == List{0,2,3};
List{0,1,2,3,2,1}.removeAll(List{1,2}) == List{0,3};
List{0..4}.subList(1,3)         == List{1,2};
```

```
List<X> add(X o);                                              Signature
List<X> add(int index, X o);
List<X> prepend(X o);
List<X> addAll(Collection<X> c);
List<X> addAll(int index, Collection<X> c);
boolean contains(X o);
boolean containsAll(Collection<X> c);
X       get(int index);
X       first;
X       last;
List<X> rest;
int     indexOf(X o);
int     lastIndexOf(X o);
boolean isEmpty;
int     count(X o);
List<X> remove(X o);
List<X> removeAtIndex(int index);
List<X> removeAll(Collection<X> c);
List<X> retainAll(Collection<X> c);
List<X> set(int index, X o);
int     size;
List<X> subList(int fromIndex, int toIndex);
List<Y> flatten;                // X is a collection type Collection<Y>
Set<X>  asSet;
```

Fig. 3.16. Signature of lists of the type `List<X>`

```
Collection<X> add(X o);                                        Signature
Collection<X> addAll(Collection<X> c);
boolean       contains(X o);
boolean       containsAll(Collection<X> c);
boolean       isEmpty;
int           count(X o);
Collection<X> remove(X o);
Collection<X> removeAll(Collection<X> c);
Collection<X> retainAll(Collection<X> c);
int           size;
Collection<Y> flatten;   // X has the form Collection<Y> or Set<Y>
List<Y>       flatten;   // X has the form List<Y>
Set<X>        asSet;
List<X>       asList;
```

Fig. 3.17. Signature of containers of the type `Collection<X>`

Deeply nested container structures contains some structural information about the nesting which is sometimes helpful for specifying systems. We can,

for instance use the type Set<Set<Person>> to describe a grouping of sets of persons.

```
let Set<Set<Person>> ssp = { a.bidder | a in Auction }
in ...
```

OCL

Using the operator flatten allows us to reduce the nesting depth of container structures.

The flatten operator merges the two "upper" levels of containers without dealing with the internal structure of the embedded element type X contained in these two levels. This is also called *shallow* flattening. A precise description can be found in Volume 1.

In navigation chains, the flatten operator is used by default and implicitly such that the result of a navigation chain never represents a container structure that is nested more deeply than the source structure. The only exceptions are navigation chains that start from a single object, which, depending on the form of the association, can lead to a set or a sequence.

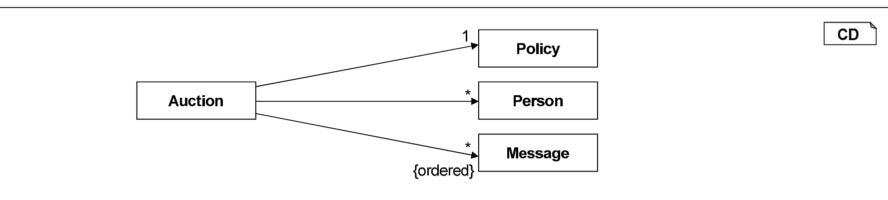


Fig. 3.18. Abstract model for explaining navigation results

Starting with a simple object, the result of a navigation chain depends on the multiplicity of the association as shown in Fig. 3.18:

```
let Auction         a = ...;
    Policy         po = a.policy;
    Set<Person>   spe = a.person;
    List<Message> lm  = a.message
in ...
```

OCL

Quantifiers and Special Operations

We can use the two quantifiers **forall** and **exists** to describe properties that must be satisfied for either all elements of a group or for at least one of the group's elements. Some of the previous examples have already shown how we can use quantifiers to formulate expressions over the elements of a container. We can combine quantifiers over multiple variables:

```
inv Message1:
  forall a in Auction, p in Person, m in a.message:
    p in a.bidder implies m in p.message
```

OCL

The third quantifier in this example shows that the source set that is being quantified can be the extension of a class, but also any set-valued or list-valued expression.

The existential quantifier is dual to the universal quantifier:

```
inv:
  (exists var in setExpr: expr)      <=>
      !(forall var in setExpr: !expr)
```

Both quantifiers are usually only applied to finite containers or sets of class objects. This has the advantage of computability (at least in principle), as the quantified variable ranges only over existing objects. For the quantified set, it is essential to use the extension—in the form of all currently existing objects—of a class, such as `Person`. The extension is unrestricted but finite at any point in time.

In addition to finite sets of objects, OCL/P allows the use of quantifiers across infinite primitive data types, such as `int`, which is naturally not computable. Quantification over containers also ranges over all potentially possible sets and lists, and not over the `Set` and `List` objects that currently exist in the system. Consider the following expression:

```
inv SetQuantor:
  forall Set<Person> lp: lp.size != 5
```

This expression interprets the variable `lp` across all *potential* sets over the `Person` objects that actually exist at a point in time. Thus, a quantification via `Set<Person>` is a combination of the interpretation of an infinite quantifier on a primitive data type and a finite quantifier on the reference type contained in the primitive data type. Accordingly, quantifiers on lists are infinite and we can therefore use them only for specification. As the power set of a finite set is also finite, the above quantification via `Set<Person>` is therefore finite and moreover it can be computed.

We can use the special operator `any` to select an element from a container structure. This operator is not defined uniquely for sets, as the following defining equations show:

```
(any listExpr) == listExpr.get(0);
(any setExpr) == any setExpr.asList;
(any var in collection: expr) == any { var in collection | expr }
```

To process sets and lists element by element, we can also use the `iterate` operator besides the already known form of comprehension. This operator simulates a loop with a state memory known from functional and imperative programming.

```
iterate { elementVar in setExpr;
          Type accumulatorVar = initExpr :
          accumulatorVar = expr
        }
```

We can use the `iterate` operator, for example, to compute the total of a set of numbers:

```
inv Sum:
  let int total =
     iterate { elem in Auction;
               int acc = 0 :
               acc = acc+elem.numberOfBids
             }
  in ...
```

OCL

The `defined` operator allows us to handle undefined values. This operator returns `true` exactly when its argument is defined. For undefined arguments, this operator evaluates to `false`:

```
context Auction a inv:
  let Message mess = a.person.message[0]
  in
    defined(mess) implies ...
```

OCL

A problem specific to OCL is the definition of transitive closures via navigation paths: OCL is a first-order logic and therefore not capable of specifying a transitive closure. OCL/P therefore has a special operator `**` which, when applied to an association, retains that association's signature. In combination with inheritance we have the four cases specified in Fig. 3.19.

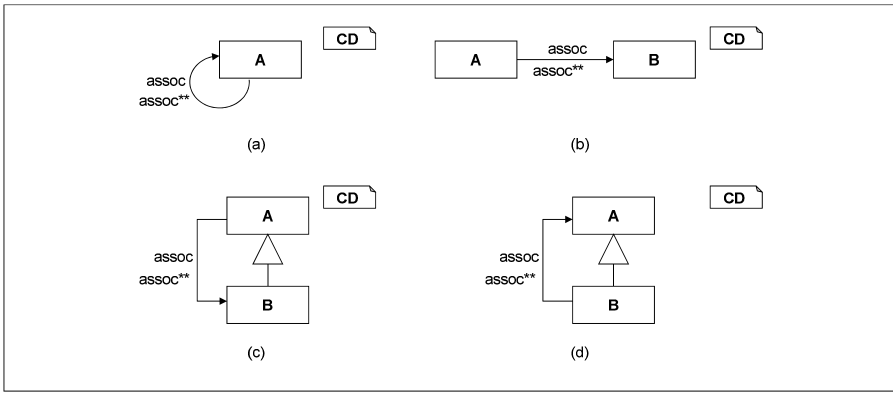


Fig. 3.19. Typing of the transitive closure

3.2.4 Functions in OCL

We can specify the effects of methods of the underlying model using pairs of preconditions and postconditions. Furthermore, the model can provide

methods for specifying OCL constraints. These methods are called *queries* and thus labeled with the stereotype ≪query≫. Queries have no side effects as described in Volume 1.

When modeling complex properties, it is often useful or even necessary to define additional queries which are not present in an implementation, however. We label these *specification queries* with the stereotype ≪OCL≫.

Method Specifications

To describe individual methods, we use a *method specification* formulated in the style of a precondition/postcondition.

The context of a method specification is defined by a method, the class that contains the method, and the parameters. Two conditions, the *precondition* and the *postcondition*, which can both be given names, characterize the effect of the method.

The precondition describes the properties under which a method can be called to compute a defined and robustly implemented reaction. The postcondition characterizes the result of the method call and the changes of the object state.

In the postcondition, we assess the result of the method using the special variable `result`. We use ≪static≫ to label static methods. Like in Java, no `this` may be used in the bodies of static methods.

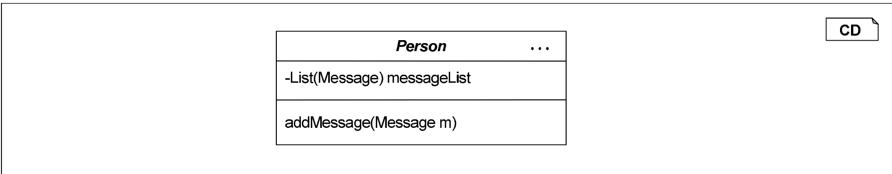


Fig. 3.20. Excerpt of the person class

The postcondition of the method specification can use the state before the method call explicitly. We can achieve this by using the postfix `@pre`. The method `addMessage` from Fig. 3.20 is used to add a further message for a person:

```
context Person.addMessage(Message m)
pre:  m.time >= messageList.last.time
post: messageList == messageList@pre.add(m)
```

OCL

The operator `@pre` may only be applied to individual attributes or navigation elements. The expression `a@pre` evaluates to the value of the attribute `a` at the time of the method call and thus allows the comparison between the old and the new value.

If the precondition is not satisfied, the specification gives no statement about the behavior of the method. We can therefore use multiple specifications to describe a complex method. However, if preconditions overlap, both postconditions must be satisfied. This composition technique allows method specifications to be passed to and specialized in subclasses. For details, see Volume 1.

For complex method specifications, there is an extended form of the **let** construct that allows us to define variables that are assigned in both conditions.

```
context Class.method(parameters)
let var = expression
pre:  ... var ...
post: ... var ...
```

OCL

Recursive definitions of functions are permitted but as discussed in Volume 1, may only be used in restricted form.

3.3 Object Diagrams

Object diagrams model actual structures in exemplary form. They are therefore particularly suitable for representing static, unchanging structures in dynamic, object-oriented systems or special situations used as preconditions or postconditions for tests. The integration with OCL leads to an "object diagram logic" that permits a methodologically enhanced use of object diagrams.

3.3.1 Introduction to Object Diagrams

A short description of the most important terms for object diagrams is shown in Fig. 3.21. These terms are explained in more detail below.

Fig. 3.22 shows a simple object diagram from the auction system that consists of only one object. This object describes an auction for electrical power.

In principle, object diagrams *conform* to the structure specified by class diagrams. They are very similar to class diagrams but differ in some substantial points and are therefore an independent notation. Objects contain attributes that are usually specified with specific values. Multiple objects of the same class may occur, which is why each object has its own (or if applicable, anonymous) name.

No inheritance relationships are shown in an object diagram. Instead, we can list the attributes inherited from superclasses explicitly in the object of the subclass. The inheritance structure between classes is therefore "expanded" in an object diagram. While class diagrams have a third field for representing methods, methods are not detailed in object diagrams. We therefore do not see a method list for objects in an object diagram.

Object: An object is an instance of a *class* and contains the *attributes* defined by the class. These attributes are initialized with a value (or, where applicable, are still uninitialized). In an object diagram, *prototypical objects* are used to illustrate example situations. There is normally no 1:1 relationship between the *prototypical objects* visible in the diagram and the real objects in the system (see also the discussion in Section 4.2, Volume 1).

Object name: The object name allows an object to be named uniquely in the object diagram. A descriptive name is used as the object name, which is this name not usually found in the real system, because there system-specific object identifiers are used.

Attribute: An attribute describes a part of the state of an object. In an object diagram, an attribute is characterized by the attribute name, the type, and a specific value. Further characteristics, such as visibility, can be added to the attribute. In *abstract object diagrams*, we can use variable names or expressions whose content remains "unspecified" in the diagram instead of specific values. Attribute types or values may also be omitted.

Link: A link is an instance of an association between two objects whose classes each participate in the association. The navigation direction, association name, and role name can also be represented on the link.

Composition link: A composition link is a special form of a link in which, in addition to the pure connection, there are further dependencies of the part on the whole. A composition link is an instance of a composition.

Fig. 3.21. Terminology definitions for object diagrams

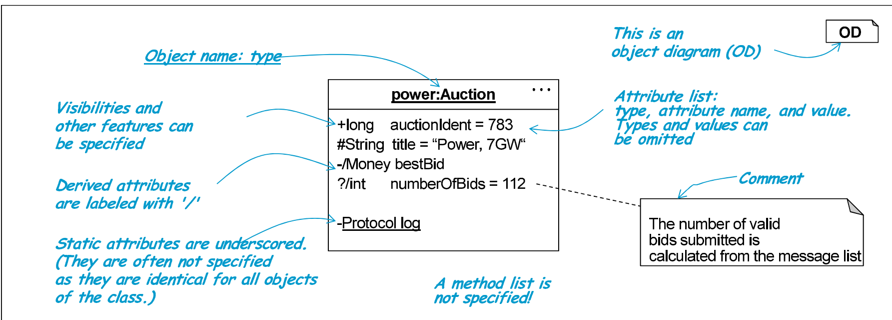


Fig. 3.22. Individual auction object

A link connects two objects. Just like an object is an instance of a class, a link is an instance of an association. Fig. 3.24 shows an object structure for an auction which contains at least the three persons specified. One of these persons is permitted only as an observer.

If an association has a *qualifier* at one end, in an object diagram a concrete qualifier value can be specified at every link of the association. Fig. 3.25 shows such an object diagram.

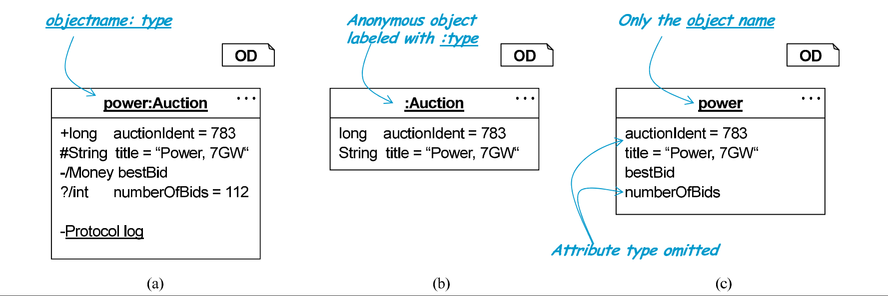


Fig. 3.23. Forms of representation for the auction object

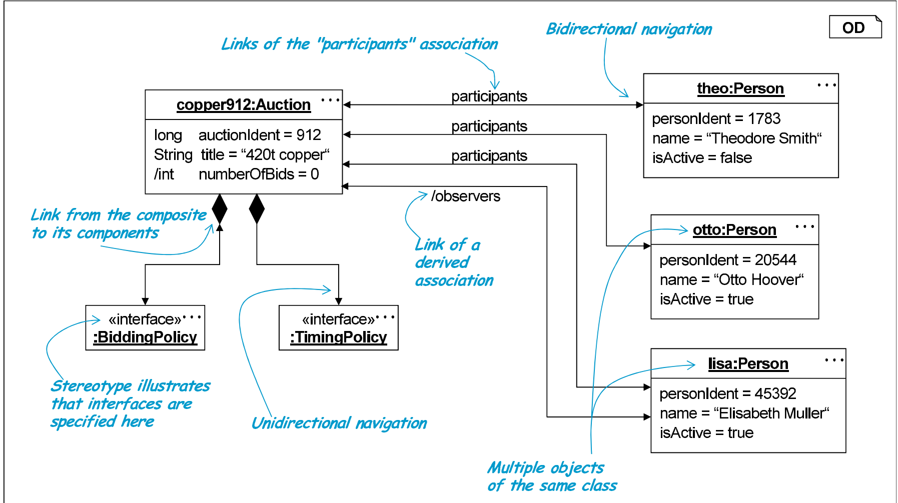


Fig. 3.24. Object structure for an auction

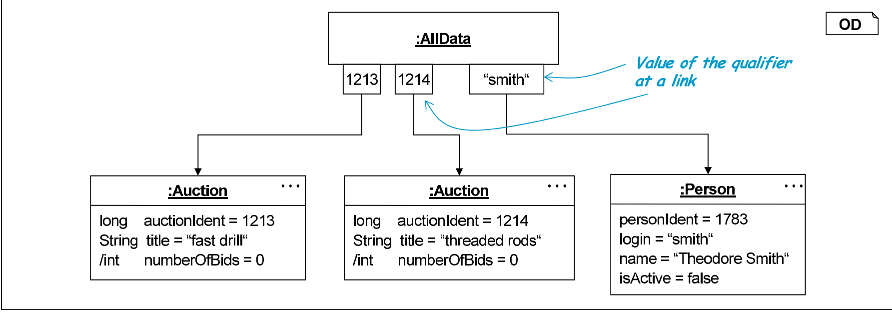


Fig. 3.25. Qualified links

3.3.2 Compositions

A *composition* is represented in an object diagram by a solid diamond at one association end. While a class may appear in multiple compositions in class diagrams, in an object diagram linking one object as part to multiple compositions is not permitted. To highlight the dependency of parts on the whole and the composition character more strongly, we can use the alternative representation on the right in Fig. 3.26.

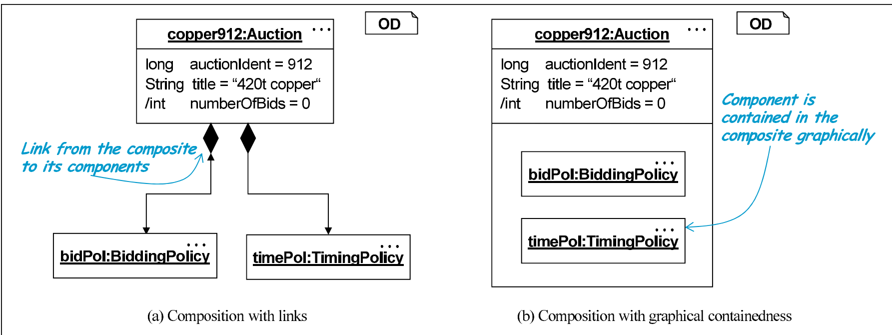


Fig. 3.26. Alternative representations for a composition

3.3.3 The Meaning of an Object Diagram

Due to the dynamics of object systems, the number of objects in a system varies constantly. Furthermore, object structures can change their links dynamically. This means that an unlimited number of variations of object structures can arise, and it is usually not possible to represent these variations completely. An object diagram therefore models a system situation valid for a limited time, and in extreme cases, a *snapshot* valid for a single point in time.

As discussed in Volume 1, structures modeled with object diagrams have a *prototypical*, *pattern-like* character. They show an example situation without this situation necessarily taking on a normative character. The situation represented in the object diagram does not have to occur in the actual execution of the system. However, it can occur more than once, in chronological order or even at the same time. Different or (sometimes) the same objects can be involved in each of the situations that occur. Object diagrams can therefore represent overlapping structures in the system. There must be a clear distinction between the representation of an object in the object diagram and the real objects of a system. The objects represented in the object diagrams are referred to as *prototypical* objects.

The ability to instantiate prototypical objects is a basis for integrating object diagrams in the OCL logic.

3.3.4 The Logic of Object Diagrams

A core element of integrating the exemplary object diagrams in OCL is the possibility of understanding an object diagram in simple terms as an expression about the objects described. As described in Volume 1 through a transformation of object diagrams to OCL, the objects names act as free variables of such expressions, whereas anonymous objects are existentially quantified. Therefore, in the embedding OCL constraint, we must define all names used in the object diagram.

The name of an object diagram represents the expression declared in the diagram and can be used freely in OCL. The following expression can therefore be used to stipulate that in any auction that has already begun, a welcome message has been sent. The two object diagrams from Fig. 3.27 are used here, with `Welcome1A` representing the prerequisite for the situation in `Welcome1B`.

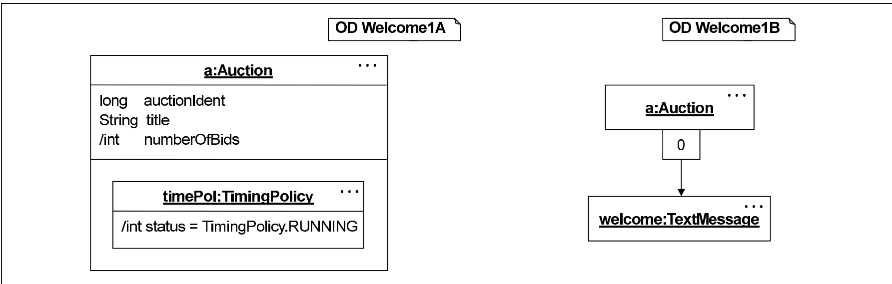


Fig. 3.27. Object diagrams for the welcome message

```
inv Welcome1:                                          OCL
 forall Auction a, TimingPolicy timePol:
   OD.Welcome1A implies
     exists Message welcome: OD.Welcome1B
```

The expression is as follows: "If objects `a` and `timePol` exist and satisfy object diagram `Welcome1A`, then the object `welcome` exists and all properties formulated in object diagram `Welcome1B` are valid."

By replacing specific values with variables or OCL expressions, we can create *abstract object diagrams*. We can import the abstract values into an OCL constraint and use them there to specify properties. For example, to arrange a test auction with 100 persons, we can use the object diagram `NPersons` from Fig. 3.28.

The integration of OCL and object diagrams is significantly more comfortable for descriptions than OCL alone. However, the expressiveness of the descriptions corresponds to that of the original OCL. But object diagrams have become more expressive, allowing us to describe:

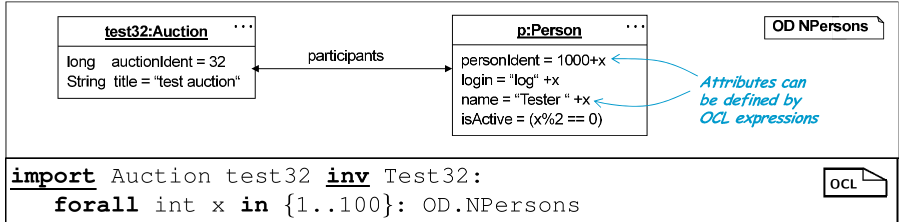


Fig. 3.28. Auction with parameterized persons object

- Alternatives
- Undesired situations
- Composed object structures
- Generalizations (patterns)

It is primarily the logic operators disjunction, negation, conjunction, and quantifiers that are used here. This increases the power of expression of object diagrams, elevating them from example expressions to generally valid expressions.

3.4 Statecharts

Statecharts are an evolution of the automata applied to describe object behavior. Each somewhat complex system has controlling parts that can be modeled with Statecharts. The variant of Statecharts presented here uses OCL as constraint language and Java statements as actions.

3.4.1 Properties of Statecharts

Automata occur in various forms. They can be executable, used to recognize sequences of letters or messages, to describe the state space of an object, or to specify response behavior to a stimulus. The overview article [vdB94] discusses a number of syntactical variants and semantic interpretation options for Statecharts and shows that these have to be adapted to the respective application domains. With suitable stereotypes for control, we can use the UML/P Statecharts for modeling, generating code, or describing test cases. Fig. 3.29 shows a Statechart that represents a simplifying abstraction of the state system for an auction. Fig. 3.30 contains an overlapping list of tasks that a Statechart can take over.

A compact overview of the Statechart constructs is summarized in Fig. 3.31, 3.32, and 3.33.

A Statechart describes the response behavior of an object that occurs when a stimulus is applied to this object. A stimulus can be a method call, an asynchronous message, or a timeout. The stimulus is processed atomically.

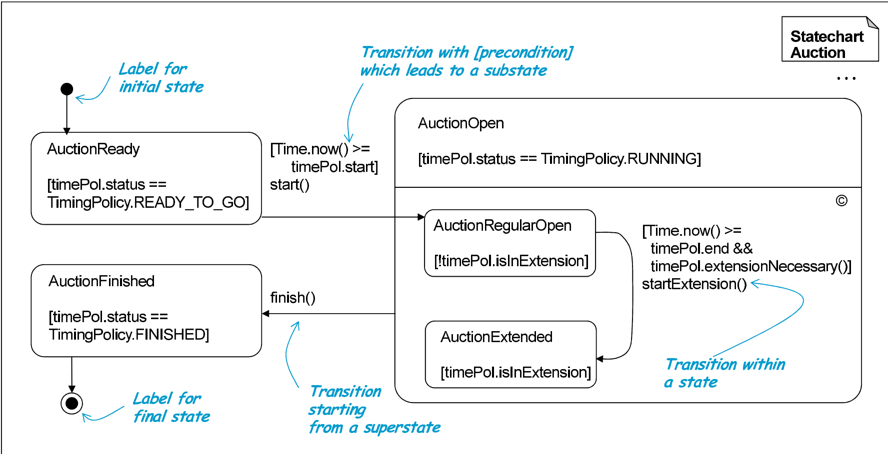


Fig. 3.29. Statechart

The tasks of a Statechart can be:

- Representation of the life cycle of an object
- Description of the implementation of a method
- Description of the implementation of the behavior of an object
- Abstract requirement description for the state space of an object
- Representation of the order of the permitted occurrence of stimuli (order of calls)
- Characterization of the possible or permitted behavior of an object
- Link between the description of the state and the description of the behavior

Fig. 3.30. The variety of tasks of a Statechart

State: A *state* (synonym: *diagram state*) represents a subset of the possible *object states*. A diagram state is modeled with a state name and optionally also a *state invariant*, *entry action*, *exit action*, or *do activity*.

Initial state: Objects start their life cycle in an initial state. Multiple initial states can be used to represent multiple forms of object instantiation. The meaning of an initial state as part of another state is described in Section 5.4.2, Volume 1.

Final state: A final state describes that in this state, the object has satisfied its obligation and is no longer needed. However, objects can exit their final states again. The meaning of a final state as part of another state is described in Section 5.4.2, Volume 1.

Substate: States can be nested hierarchically. A *superstate* contains multiple substates.

State invariant: A state invariant is an OCL constraint which, for a diagram state, characterizes which object states are belonging to this diagram state. State invariants of different states may normally overlap.

Fig. 3.31. Terminology definitions for Statecharts, part 1: states

Stimulus: A stimulus is caused by other objects or the runtime environment and leads to the *execution of a transition*. Examples of stimuli include a method call, a remote procedure call, the receipt of a message sent asynchronously, or the notification of a timeout.
Transition: A transition leads from a *source state* to a *destination state* and contains a description of the *stimulus* as well as the response to the stimulus in the form of an *action*. Additional OCL constraints restrict the enabledness and specify the response of a transition in more detail.
Enabledness: A transition is *enabled* exactly when the object is in the source state of the transition, the stimulus is correct, and the precondition for the transition is valid. If multiple transitions are enabled in the same situation, the Statechart is *nondeterministic* and it is open which transition is chosen.
Precondition of the transition: Besides by the source state and the stimulus, we can also restrict the enabledness of a transition with an OCL constraint which must be satisfied for the attribute values and the stimulus.
Postcondition of the transition (also: *action condition*)**:** In addition to an operational description of the reaction to a stimulus, an OCL constraint can also specify the possible reaction in a property-oriented form.

Fig. 3.32. Terminology definitions for Statecharts, part 2: transitions

Action: An *action* is a modification of the state of an object and its environment described through operational code (e.g., with Java) or through an OCL specification. A transition usually contains one action.
Entry action: A state can contain an *entry action* that is executed when the object enters the state. If actions are described operationally, the entry action is executed after the transition action. If there is a property-oriented description, the conjunction of both descriptions applies.
Exit action: Analog to an *entry action*, a state can contain an *exit action*. In an operational form, it is executed before the transition action; in the property-oriented form, the conjunction also applies.
Do activity: A state can contain a permanently enduring activity called the *do activity*. It can be implemented via various mechanisms for the creation or simulation of parallelism, such as separate threads, timers, etc.
Nondeterminism: If there are multiple alternative transitions that are enabled in a situation, this is referred to as *nondeterminism of the Statechart*. The behavior of the object is thus *underspecified*. There are several methodologically useful options for using and dissolving underspecification during the software process.

Fig. 3.33. Terminology definitions for Statecharts, part 3: actions

Therefore, the processing cannot be interrupted and is not parallel to other transitions of the same Statechart.

While an object typically has an infinite state space, a Statechart consists of a finite, typically even small set of diagram states. The diagram states therefore represent an abstraction of the object state space. We can define the relationship between diagram states and object states precisely by adding

OCL constraints. The same applies for the preconditions and effects of transitions. Depending on the level of detail and the form of representation of these conditions, a Statechart can therefore be seen as an executable implementation or as an abstract requirement description. We can therefore use Statecharts beginning from requirements analysis until implementation.

Fig. 3.34 illustrates how we relate the finite number of diagram states and transitions of a Statechart with an underlying infinite transition system that describes the behavior of an object.

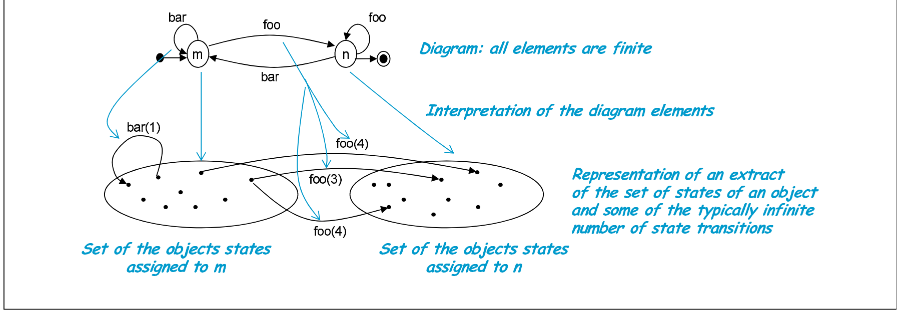


Fig. 3.34. Interpretation of a diagram

The interpretation of an element of the diagram through a set of states or transitions has certain effects and there is a detailed examination of these effects in Volume 1. Important properties are summarized here:

- Statecharts are based on Mealy automata which process stimuli, allow output to be emitted (for example, in the form of method calls), and allow changes to the object state.
- There is no parallelism within the object described by the Statechart. The UML/P Statecharts therefore have no parallel states (elsewhere called "and"-states).
- Statecharts generally permit nondeterminism. This can lead to a real nondeterminism of the implementation, but can also be interpreted as an underspecification of the model and can be eliminated later in the process by more detailed, deterministic specifications.
- Spontaneous (ε-) transitions model observations in which the triggering stimulus remains invisible (for example, an internal method call or a timer). This means that spontaneous transitions merely provide a special form of underspecification.
- A Statechart can be incomplete in that it does not provide any transitions for specific combinations of states and stimuli. In this case, there is no statement made about the implementation and therefore any robust reaction is possible. In particular, there is no need that the implementation should ignore the stimulus.

- If the Statechart is incomplete, we can define a specific behavior by applying a suitable stereotype that adds implicit transitions, e.g., into an error state.

The two primary tasks of Statecharts are the representation of life cycles of objects and the representation of the behavior of individual methods.

3.4.2 Representation of Statecharts

Fig. 3.35 shows an individual state that the class *Auction* can take. In addition to the name `AuctionOpen`, the state contains a *state invariant*, one *entry action*, and one *exit action*, as well as one *or do activity*.

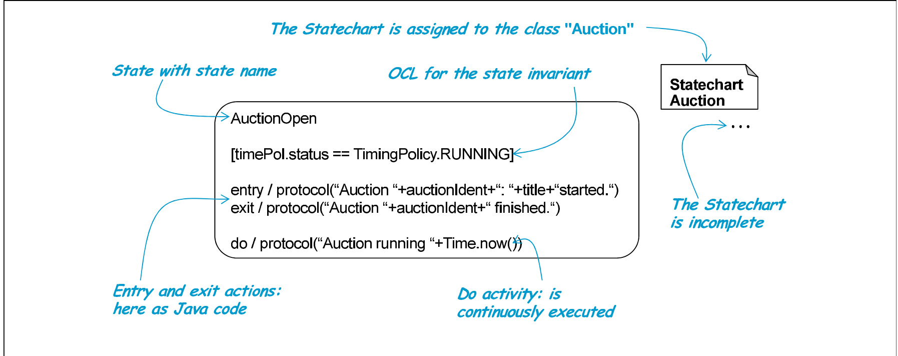


Fig. 3.35. A state of the class Auction with an invariant and actions

As a diagram state corresponds to a set of object states, we can use the *state invariant* described in OCL to establish this relationship. State invariants do not have to be disjoint. If they are disjoint, they are referred to as *data states*, otherwise they are *control states*. The *data state* of an object is determined by the attributes of the object's own and any dependent objects. In a real system, the *control state* in addition manifests itself through the program counter and the call stack, but a state invariant can only provide information about the data state.

We can use a hierarchy for states to prevent a state explosion. Just like any other state, a hierarchically divided state has a name and can contain a state invariant, an entry and an exit action, and a do activity. The flat and hierarchical representation of states in a Statechart is, as illustrated in Fig. 3.36, equivalent if the state invariants are taken into account accordingly.

Fig. 3.29 shows the label for the initial and final states at the highest hierarchy level. We can also label initial and final states within a hierarchically decomposed state. However, they then have a somewhat different meaning (see Volume 1).

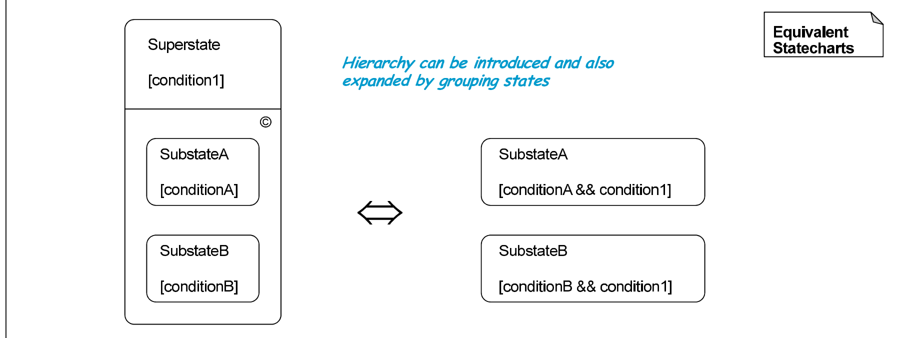


Fig. 3.36. Introduction and expansion of a hierarchy

If the object is in the source state of a transition and satisfies the triggering condition, the transition can be executed. Accordingly, an action is executed and the target state of the transition is entered.

Stimuli

There are five different categories of stimuli that lead to the triggering of a transition:

- A *message* is received
- A *method call* takes place
- The result of a *return statement* is returned in response to a call sent out earlier
- An *exception* is caught
- The transition occurs *spontaneously*

For the receiving object, there is no difference between whether a method call is transmitted asynchronously or as a normal method call. Therefore, in the Statechart, there is also no differentiation between these two forms of stimuli. This results in the types of stimuli for transitions shown in Fig. 3.37.

Firing Rules

The *enabledness* of a transition can be characterized as follows:

1. The object must be in an object state that corresponds to the source state of the transition.
2. Either the transition is spontaneous and therefore does not require any stimulus, or the stimulus required for the transition has occurred.
3. The values specified in the stimulus description (for example, method parameters) match the actual values of the stimulus received. If variables are specified in the stimulus description, the actual values are assigned to these variables.

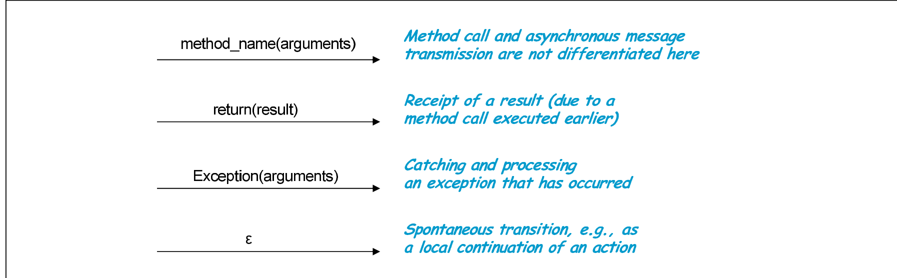


Fig. 3.37. Types of stimuli for transitions

4. The precondition evaluated on the object state and the parameters of the stimulus received evaluates to true.

It is possible that a precondition cannot be satisfied in any situation. In this case the transition is useless, as it will never be executed. As a result of nondeterminism (underspecification), it is also possible for multiple transitions to be enabled simultaneously. This also means that an enabled transition does not necessarily have to be executed. *Nondeterminism* in a Statechart does not necessarily mean, however, that the implementation has to be nondeterministic, as the programmer can reduce the underspecification by selecting the more suitable realization himself. Fig. 3.38 shows two permitted situations for overlapping firing conditions.

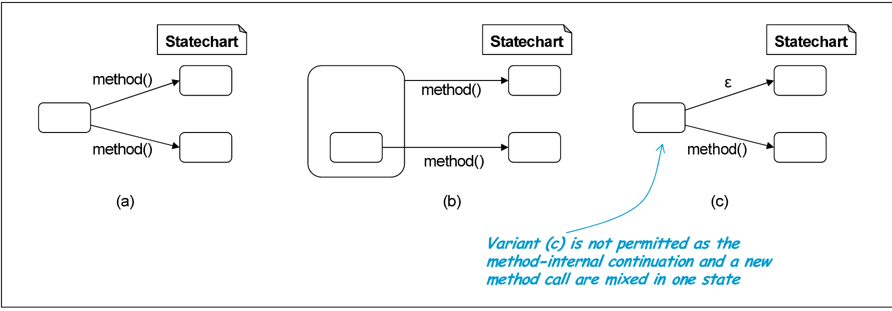


Fig. 3.38. Situations for overlapping firing conditions

In both cases (a) and (b) both alternatives are possible. Case (a) can be resolved with explicit priorities. Furthermore, for nondeterminism at different hierarchy levels, we can use the stereotypes ≪prio:inner≫ or ≪prio:outer≫ to generally define whether inner or outer transitions are given priority.

If a Statechart is incomplete, we can also use a stereotype to specify the behavior more precisely.

1. «completion:ignore» indicates that the stimulus is ignored.
2. An error state is taken if there is a state labeled «error». For processing exceptions, a further state can be labeled with «exception».
3. The stereotype «completion:chaos» permits arbitrary underspecification in the incomplete part of the Statechart.

As discussed in Section 5.2.6, Volume 1, this results in differences in the interpretation of the life cycle of an object. In the first two interpretations, the life cycle is understood as the *maximum possible*; in the last, it is understood as the *minimum ensured*.

The use of «completion:chaos» is particularly interesting in the specification phase, when refinements are performed to specify the behavior in more detail but not to adapt it.

Actions

We can use *actions* to describe the reaction to the receipt of a stimulus in a certain state by adding them to the state as entry or exit actions or adding them to the transition as a reaction. UML/P provides two types of actions: a *procedural form* allows the use of assignments and control structures; a *descriptive action form* allows the effect of an action to be characterized without specifying how this action is actually realized.

Procedural actions are realized with Java and descriptive actions ("action conditions") are specified with OCL. Fig. 3.39 contains an excerpt from a Statechart for the class `Auction`. This excerpt shows a transition with a procedural action and a postcondition.

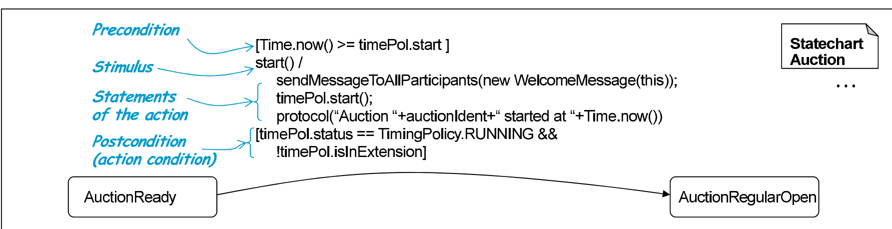


Fig. 3.39. Transition with a procedural and a descriptive action

On the one hand, we can formulate action conditions as a redundant addendum to action statements, so that they can be used in tests, for example. On the other hand, action conditions can supplement the statements. Thus, we can define part of the behavior procedurally and characterize part of it descriptively using an OCL constraint.

The combination of *entry* and *exit actions* from states and transition actions depends on the form of the specified transition. Fig. 3.40 shows the transfer

of procedural actions to the transitions and demonstrates the order in which entry and exit actions are executed in hierarchical states. Procedural actions are thus composed sequentially.

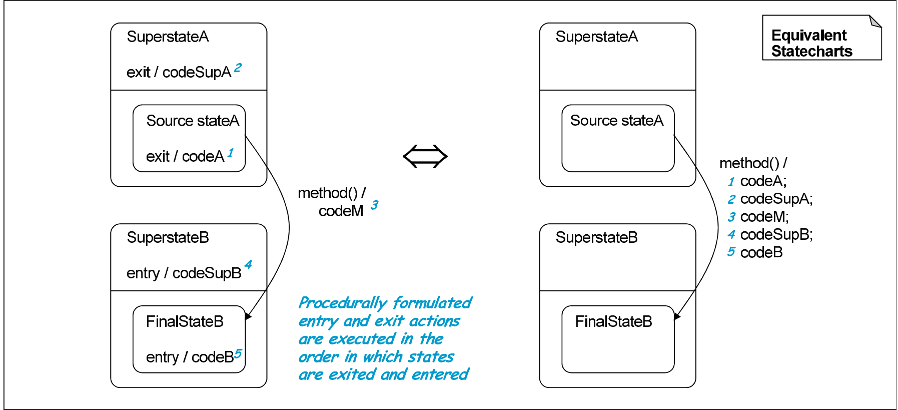


Fig. 3.40. Entry and exit actions in hierarchical states

If the actions of a Statechart are specified by OCL action conditions, as shown in Fig. 3.41, we use the logical conjunction instead of the sequential composition.

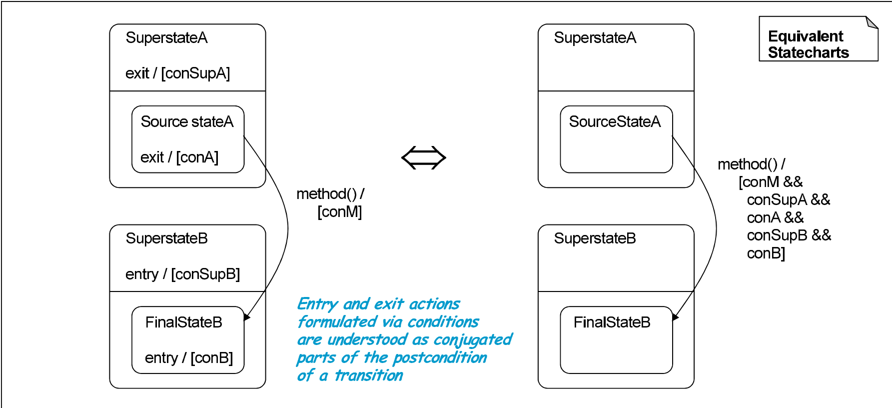


Fig. 3.41. Conditions as entry and exit actions in states

An alternative to the logical composition is explained in Volume 1 and characterizes a section-by-section validity of the conditions which is necessary, for example, with transition loops.

Advanced Concepts

State-internal transitions are independent transitions for which the entry or exit action of the containing state is not executed. Fig. 3.42 illustrates a state-internal transition.

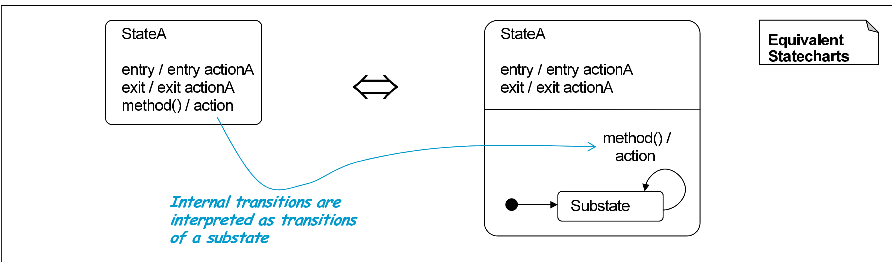


Fig. 3.42. State-internal transitions

If a state represents a situation of the object in which an activity prevails—for example, a warning message is flashing—we can describe it with a *do activity*. Fig. 3.43 characterizes the interpretation of the activity by using a timer.

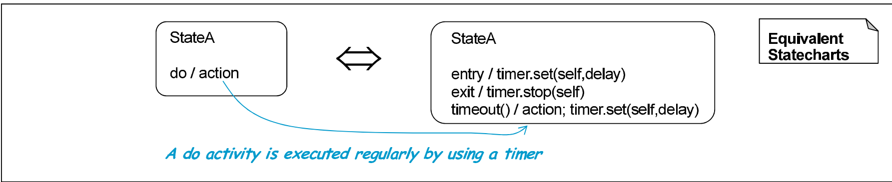


Fig. 3.43. do activity as a time-controlled repetition of an action

The concepts introduced in Fig. 3.42 and 3.43 are explained by transformation to existing concepts of the Statechart. Volume 1 specifies a transformation system consisting of 19 rules that reduces Statecharts completely to flat Mealy automata and therefore simplifies further processing. Statecharts therefore have a transformation calculus that is suitable for a refinement that preserves semantics.

3.5 Sequence Diagrams

We use sequence diagrams to model interactions between objects. A sequence diagram represents an exemplary excerpt from the interactions occurring in a software system execution. It models the interactions and method

calls that occur in this system run. We can extend the sequence diagram by adding property descriptions in OCL.

A sequence diagram describes the order in which method calls are started and finished. Similarly to Statecharts, it therefore models behavioral aspects. However, there are some significant differences:

- A sequence diagram focuses on the interaction between objects. In contrast, the inner state of an object is not represented.
- A sequence diagram is always an *example*. Therefore, in the same way as for the object diagram, the information represented can occur any number of times during the execution of a system, multiple times in parallel, or does not have to occur at all.
- Due to their exemplary nature, sequence diagrams are not suitable for modeling behavior completely and are used primarily during the requirements specification and, as shown in this volume, for modeling test cases.

Fig. 3.44 shows a typical sequence diagram. The main terms that occur in the sequence diagram are explained briefly in Fig. 3.45.

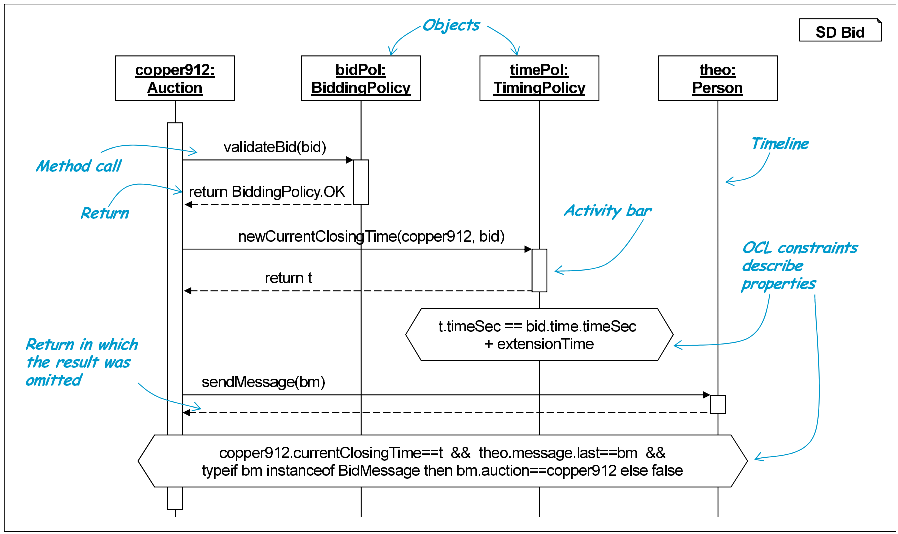


Fig. 3.44. Sequence diagram for accepting a bid

In a sequence diagram, *objects* are represented in a row next to one another and have a *timeline* that points downwards. We simplify UML sequence diagrams by assuming that a simultaneous or overlapping sending of messages does not occur. This simplification is motivated from the use of sequence diagrams for test purposes, where concurrency is explicitly controlled in order to obtain deterministic test results. This simplifies the modeling with sequence diagrams significantly but under some circumstances,

Object: In a sequence diagram, an object has the same meaning as in an object diagram (see Fig. 3.21) but is represented only with the name and type. Multiple objects of the same type are allowed. Name or type are optional.
Timeline: In a sequence diagram, chronologically sequential events are represented from the top to the bottom. Each object has a timeline that represents the progression of time for this object. The time is not true to scale and also does not have to be identical in all objects. Thus, the timeline is used only to represent chronological sequences of interactions.
Interaction: An interaction between two objects can be triggered by one of multiple forms of stimuli. These include: method calls, returns, exceptions, and asynchronous messages. In a sequence diagram, specifying the parameters of interactions is optional.
Activity bar: To allow a method call to be processed, an object is active for a certain time. The activity bar is used to represent this activity. For a recursion, the activity bar can also occur nested.
Constraint: OCL constraints can be used for a detailed description of properties holding within a system execution.

Fig. 3.45. Terminology definitions for sequence diagrams

certain situations that can occur in an implementation cannot be represented with sequence diagrams.

The types of interaction possible are shown in Fig. 3.46. Each interaction, with the exception of the recursive call, is represented by a horizontal arrow. This arrow symbolizes that the respective time consumed is disregarded. Method calls, returns, or exceptions are specified on the arrows optionally with arguments in Java.

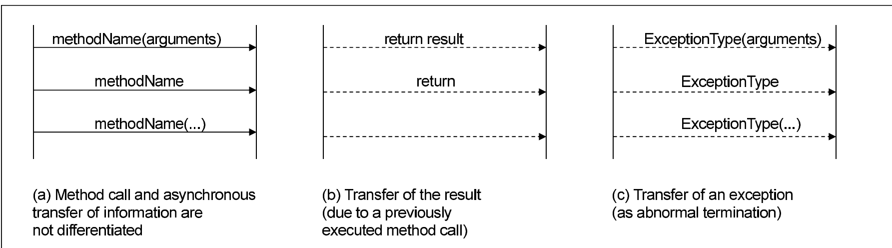


Fig. 3.46. Forms of interaction in a sequence diagram

If a method is static, it is underscored, like in the class diagram. In this case, the arrow ends at a class which is thus used analog to an object.

If an object is created during the interaction, this is represented by the timeline of the object beginning later. Fig. 3.47 contains two typical forms of object instantiation.

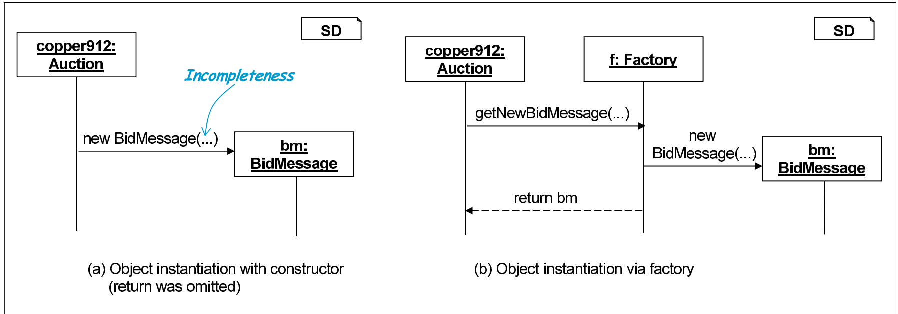


Fig. 3.47. Creation of a new object

As Fig. 3.44 shows, in a sequence diagram, we can specify additional conditions valid at specific times of the execution. In these OCL constraints, the objects and values that appear in a sequence diagram are constrained. To do this, we can access named objects as well as variables used as arguments in method calls.

An OCL constraint can describe the objects and parameters that occur and the effect of a message on an object more precisely, whereby the context of the constraint (i.e. the usable variables) results from the sequence diagram. Access to attributes is also possible through the qualification with the object. However, if an OCL expression is exclusively attached to one timeline, it belongs to this object and the attributes of this object can be accessed directly.

An OCL constraint has to be valid immediately after the last occurring interaction or the last occurring activity bar. This means that the OCL constraint only has to be adhered to immediately after this interaction. If an earlier value of an attribute is to be accessed in the constraint, this must be stored explicitly in an auxiliary variable. As Fig. 3.48 shows, we use a `let` construct based on OCL here, introducing new variables that are only accessible within the sequence diagram.

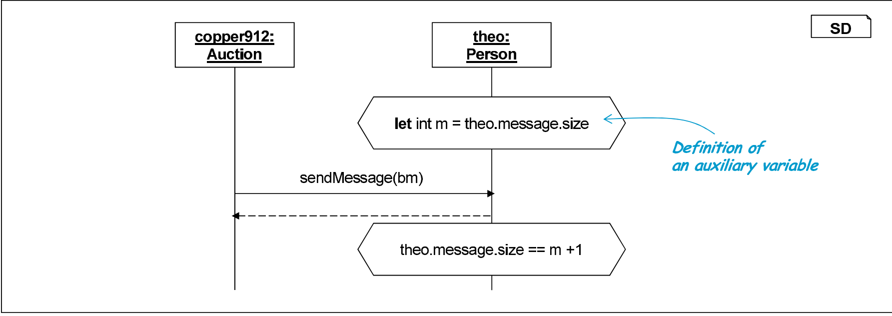


Fig. 3.48. Auxiliary variables store values

Semantics of a Sequence Diagram

We have already stated that sequence diagrams are exemplary in nature. To use sequence diagrams methodically, we have to differentiate between multiple forms of exemplariness. Analog to the object diagram, the exemplariness of a sequence diagram is based on the description of a set of objects that can occur in a system in this form in any frequency or even not at all. Moreover, the sequence of interactions is highly exemplary, as it can itself appear in this form any number of times consecutively or nested, or not appear at all.

In addition to these forms of *exemplariness*, a sequence diagram represents an *abstraction* of a sequence as it cannot contain all objects of a system or all interactions that occur. Some interactions can also be missing from a sequence diagram.

If a method call in a system takes place multiple times during the observation period, this also produces an ambiguous situation regarding which actual method call corresponds to the interaction represented in the diagram.

However, we can define the form of observation more precisely using suitable stereotypes. For objects in the sequence diagram, the representation indicator "©" shows the completeness of the interactions depicted. The alternative representation indicator "..." for incompleteness applies per default.

For an object that is observed completely in this sense, this inevitably implies that for each call, we have to specify a suitable return. In doing so, we have to specify all objects that interact directly with the object being observed. This can lead to significantly more detailed diagrams than actually desired. Therefore, Volume 1 introduces further stereotypes that allow relaxed variants of the definition of semantics.

For example, the stereotype ≪match:visible≫ is suitable for defining sequence diagrams for tests: ≪match:visible≫ prohibits the omission of interactions with other objects specified in the sequence diagram but does allow us to omit interactions with objects not specified in the diagram. The observation of this object is therefore complete with respects to all objects visible in the diagram.

4

Principles of Code Generation

Words enough have been exchanged,
let me at last see some action.

Faust, Johann Wolfgang von Goethe

Code generation is a significant success factor for the use of models in the software development process. We can generate code efficiently for the product system or test drivers from many models, thus improving the consistency between the model and its implementation as well as saving resources. This chapter describes basic concepts, techniques, and problems of code generation. It also outlines a form of representation for rules for code generation using transformation rules.

4.1	**Concepts of Code Generation**	**74**
4.2	**Code Generation Techniques**	**82**
4.3	**Semantics of Code Generation**	**89**
4.4	**Flexible Parameterization of a Code Generator**	**91**

© Springer International Publishing AG 2017
B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_4

The ability to generate executable code from a model offers interesting perspectives for software development and, to some extent, is even an essential prerequisite for some the following goals:

- Increasing the efficiency of the developers [SVEH07]
- Separating application modeling and technical code, which simplifies maintenance and evolution of the functionality as well as porting the functionality to new hardware and new versions of operating systems [SD00]
- Rapid prototyping using models that allow a more compact description of the system than it would be possible with a general purpose programming language such as Java
- Fast feedback via demonstrations and test runs
- Generating automated tests, which is a significant aspect of quality management

One of the strengths of code generation is that it allows us to create frequently recurring similar code fragments (or "aspects", [LOO01, KLM$^+$97]) quickly. With regard to integrating technical aspects, such as the GUI, persistence, or communication between distributed system parts, these aspects often have identical structures and we can easily derive them from abstract models. This drastically reduces the size and complexity of artifacts that we have to create manually. In turn, this reduces the number of programming errors, the generated code has an even better conformity to coding standards[1], and we can create the system more quickly and adapt it more flexibly based on models.

Problems with Current Tools

At present generating executable code from a model is justifiably one of the main challenges faced by the vendors and developers of modeling tools. This holds not only for UML-based tools, but also for tools for similar languages, such as Autofocus [HSSS96, Sch04], SDL (used in telecommunications) [IT07b, IT07a], or the Statecharts implemented in Statemate and Rhapsody [HN96]. Due to the various ongoing work, we can assume that the situation for code generation will continue to improve in the coming years.

Many of the tools that exist today already allow us to generate code or code frames from parts of UML[2]. However, there are also a number of fundamental problems that require design improvements.

1. Generating code fragments from class diagrams is now state of the art. In this process, fragments are generated for the classes. As a minimum, these fragments contain attribute definitions and access functions. The

[1] This is important, for example, for system certification.
[2] An overview of generators can be found at umltools.net

bodies of generated methods have to be entered manually. Because detailed models in a project are usually subject to a high rate of change, the code bodies inserted into the generated code manually have to be updated after each generation or otherwise they are lost. "Roundtrip engineering" [SK04] is therefore used as a workaround.

2. Roundtrip engineering allows the mutual transformation of code into class diagrams and vice versa. It is important to ensure that both views can be changed manually without changes in the other view being lost. In particular, method bodies are retained in the code view even when they are not visible in the class diagram. However, if the goal is to achieve a most compact form of representation of the system, then this is a dead end. Then an integrated, redundancy free presentation of graphical class diagrams and textual code bodies is more useful. The first main obstacle here is the fact that developers do not currently have enough confidence in the generated code, which means that manual intervention is still desired. It has also been not satisfactorily clarified, how and where code bodies should be stored to allow developers to process them efficiently. However, a similar situation occurred with the first compilers. Assembler source code was generated, which, in theory, should allow developers to adapt the code manually. We can assume that with the increasing maturity of the technology for code generation, the less important the level of readable source code will become and we will be able to create bytecode directly. It will then also no longer be necessary for generated code to satisfy *coding guidelines* and to be easily readable.
3. If, as in roundtrip engineering, code bodies are inserted directly in the generated code, in these code bodies, it will become increasingly difficult to abstract from the concrete realization of attributes, associations, etc. Instead, the developer will have to know the form of implementation of these elements and the resulting access functions. However, if the code bodies are also generated (rather than being inserted), then attribute accesses can be replaced by corresponding `get` and `set` methods, for example. In that case, the instrumentation of the code for tests will also be easier.
4. Unfortunately, the documented or underlying semantics (in the sense of meaning, [HR04]) for the analysis and refactoring techniques on models and the behavior arising from the code generation do not always match. This is a general problem which requires that code generation, analyses, refactoring techniques, and the documented semantics should be defined very carefully otherwise a refactoring that is correct in terms of the defined semantics may still modify the system behavior.
5. Simple tools often generate a rigid and closed form of code without addressing the specific needs of the project. A parameterization of the code generation is desirable and also necessary at many points in the code in order, for example, to integrate platform-specific adaptations, allow the

use of frameworks and storage and communication techniques, or to enable optimization of the transformation of model elements.
There are many diverse possible forms of code generation and we cannot anticipate them directly. Therefore, on the one hand, we need a flexible template or script language potentially assisted by a wizard, and on the other hand, we must ensure that we do not loose the "essential semantics" of the diagrams as a result of the technology-specific code generation.

This chapter discusses basic concepts for code generation, with explanations based on the UML/P notation. For a more detailed study of generation techniques, see [CE00]. Section 4.1 examines concepts of code generators which also cover the use of UML for modeling test cases as discussed in Chapter 7. Section 4.2 discusses requirements such as flexibility, platform-independence, and the ability to control a code generator. Section 4.3 examines the relationship between code generators and the definition of the semantics of the language. Section 4.4 describes the possible forms of a flexible code generator. A generator for UML/P that can be adapted flexibly is available with [Sch12].

4.1 Concepts of Code Generation

In anticipation of the conceptual basics discussed below, Fig. 4.1 introduces and defines the main terminology for code generation.

Using a code generator has advantages compared to conventional programming. The modeling or programming language used becomes more *understandable* due to the fact that graphical elements make the language more compact and/or clearer. The *efficiency of the software development* is also increased. The fact that less code has to be written, checked, and tested manually means that developers can become more efficient. Additional aspects, such as the better *reusability* of abstract models from a model library increase the efficiency of the developers even further. This reduces the overall effort and workload necessary for a software development. As a consequence, less project organization is necessary, which also allows further increases in efficiency.

Reusability is possible at multiple levels. We can reuse a model in an adapted form in a similar project or product line [BKPS04]. Ideally, repeated improvement gives rise to a *model framework* that we can reuse directly for similar projects and that even has a code generator specially suited to the purpose. The technical knowledge embedded in the code generator regarding, for example, the creation of interfaces or safe and efficient transfer mechanisms, can be reused independently of the model or model framework. We can also reuse models within a project. For example, we can use an object diagram both as a predicate and constructively to create an object structure.

Constructive model: A specification of the system that is used for code generation with the help of an automatic generator. Modifying a constructive model modifies the product directly. The modeling language is also seen as a high-level programming language, as it can generally be executed.

Descriptive model: A specification which is used to describe the system without actually being used constructively in the implementation of the system. The descriptions are typically abstract and incomplete. These models are used as requirements for manual implementation or are only produced as documentation after the system has been developed.

Test model: A specification which is suitable for deriving tests manually or automatically. The test model is compiled into executable code which is used to set up test data records, as a test driver, or as an expected test result.

Constructive test model: A test model that is transformed into executable tests by the code generator. In contrast, descriptive test models are transformed into tests manually.

Code generation: The activity of generating code from constructive models.

Code generator: A program that transforms a constructive model of a higher level programming language into an implementation (according to [CE00]). The generated code can be part of the product system or the test code.

Script: Contains the constructive control for the code generation. Scripts parameterize the code generator and thus allow platform-specific and task-specific code generation.

Template: A special form of a script. It describes code patterns in which specific elements of the model are inserted during code generation. Typically, a macro replacement mechanism is used.

Fig. 4.1. Terminology definitions for code generation

We can also use both forms in the product system and when defining test cases. As discussed in Section 5.2, the code generated here is very diverse and creating it manually therefore involves a lot more effort.

Some code generators today can also create *more efficient code* than would be possible with an acceptable effort through manual optimization. Of course, this applies in particular to mature compilers of normal programming languages, which use a number of optimization techniques. For executable modeling languages such as UML/P, we must assume that the increase in the efficiency of the developers is currently at the cost of a less efficient implementation. Accordingly, the generators for modeling languages are more useful for individual software and not so much for embedded system software operated en masse in devices that are as cost-effective as possible.

Ultimately, the flexible generation of code from specialist models allows us to handle technical and, to some extent, specialist variabilities in the sense of [HP02]. Open technical aspects of the functional model are filled out appropriately by a generator that is adapted to the respective specific technology.

4.1.1 Constructive Interpretation of Models

As already described in Volume 1, in essence, a *model* is a reduced or abstracted representation of the original system in terms of scale, level of detail, or functionality [Sta73]. We normally use models where the actual system is so complex that it helps to analyze certain properties of the system using the model first or to explain them to the customer using the model. This applies to architectural models of buildings as well as technical models of complex machines or models of social and economic relationships. Some models, such as construction plans or wiring diagrams, focus on describing the structure (architecture); for other models, being able to simulate functionality and other behavior-oriented properties is more important.[3]

In general, however, these models and construction drawings are aids that help us to subsequently create the respective artifact. If the model is created for the purpose of *subsequently* creating the actual artifact, it has a *prescriptive* effect. In contrast, a model is used *descriptively* if the original exists before the model. Examples of this are a model railway or photographs [Lud02].

As software is intangible, models in software development exhibit a *constructive* effect as well as a descriptive effect. If all we need to do to generate executable software from an intangible model stored in a computer is to press a button, then the model acts as a constructive specification. Due to the automated transformation, the source code of a programming language can be seen as equivalent to the object code created. Strictly speaking, the source code and object code are also models of the system. However, for practical concerns, they are identified with the system itself for the sake of simplicity, and justifiably so. We can also assumpe this for executable UML/P models.

Using models constructively has some effects that do not occur otherwise. For example, adding or omitting elements of the model changes the system modeled immediately. One example is the use of multiple Statecharts to model the behavior of a class at different levels of abstraction. A generator that can do this simulates these different levels in parallel and thus realizes a multidimensional state concept[4] for a class.[5] If a further Statechart is added as a model, and this Statechart only represents existing information in a more abstract form, the state concept is inflated even further. This may not change the overall functional behavior but it does change the internal structure and the timing behavior of the system.

Thus, we use models in various roles in software development, including the automatic generation of product code and tests. We can also transform

[3] For a more detailed discussion of the concept of the model, see, for example, [Sta73] or [Lud02].

[4] Usually cross-products.

[5] The assumption in this case is that tools are not automatically capable of recognizing that one Statechart is an abstraction of another.

a model manually as well as create models once the artifact exists. Fig. 4.2 characterizes the three dimensions for differentiating the use of models.

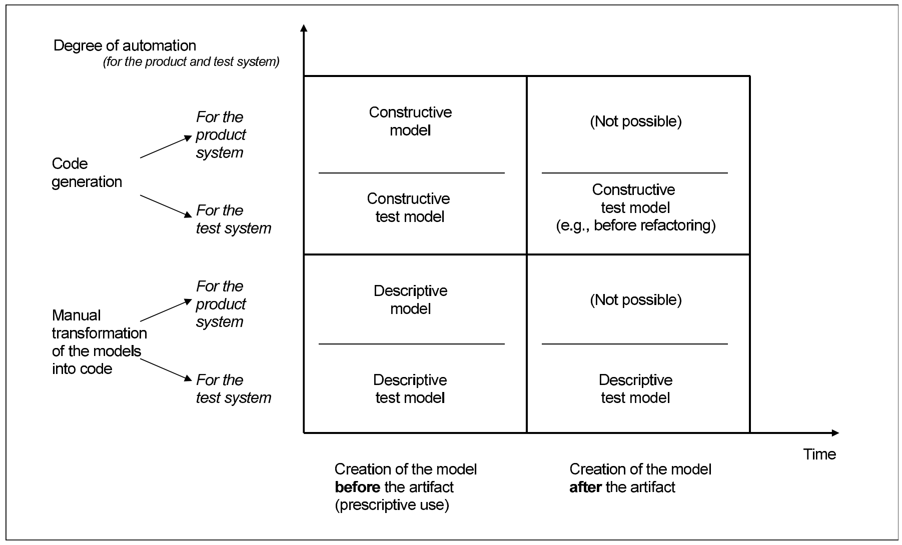


Fig. 4.2. Variants of the use of models

Strictly speaking, it is possible to create a constructively used model after the artifact and this is useful, for example, in reverse engineering. However, in this case, the model is used not to create the existing original, but rather for the next version.

The difference between the constructive and descriptive interpretation of models is related to a similar phenomenon that was discussed in detail in the domain of algebraic specifications. We should give a model used descriptively a *loose semantics* [BFG+93], thus allowing to describe various implementations that satisfy the model. Many of these implementations contain, for example, further parts of the state, functionality, or interfaces that are not mentioned explicitly in this incomplete model. A class diagram thus describes an excerpt of a system that can have further, undisclosed classes. A descriptive model can therefore be incomplete. For example, Volume 1 [Rum16] described loose semantics for sequence diagrams.

In contrast, a model used constructively represents a complete description of the software system, because the entire executable system can be gen-

erated from the model alone. For algebraic specifications, this corresponds to *initial semantics*, where a model has exactly one implementation.[6,7]

4.1.2 Tests versus Implementation

UML/P allows us to create models suitable for generating both tests and the product system. These models include object diagrams which, as discussed in Section 4.4, Volume 1, are used (1) constructively as preconditions to establish the initial situation for a test, and (2) descriptively as postconditions to specify which situation has to be satisfied after the application of the function for the test result to be considered successful.

Certain parts of a model can be used to generate only test code but not constructive code. For example, OCL constraints are generally executable. As described in Section 3.3.10, Volume 1, this generally also applies for the use of quantifiers: with the exception of quantifiers over primitive data types such as `int` and set-valued or list-valued types of a higher grade, all quantifiers are finite and can therefore be evaluated.

Nevertheless, there is a significant difference between whether a postcondition formulated in OCL can only be tested or whether it can be enforced constructively. Almost all OCL constraints that are interesting from a practical perspective belong to the first category. Indeed, the category of constructive OCL constraints is significantly smaller. This is demonstrated by the two examples below.

Sorting

The task of the method `sort` is to sort an array of numbers (`int`). A description of this task in the form of a precondition/postcondition that is often found is the following:

```
context int[] sort(int a[])                                    OCL
pre:  true
post: forall int i in {1..result.length-1}:
        result[i-1] <= result[i]
```

This specification can be tested very easily and in linear time. However, it is not suitable as a constructive description, because a generator cannot use it to generate a sorting algorithm. Furthermore, it is incomplete with regard to important properties as it does not ensure that the initial elements of the array `a[]` have to reappear in the results array `result[]`. In fact, an implementation in the form `result=new int[0]` would also be correct.

[6] In the theory of algebraic specifications, models are referred to as “specifications” and elements of the semantic domain or implementations are referred to as “models”.

[7] A specifications can only have an initial semantics, when the specifications obeys certain constraints. See also [EM85].

It is possible to describe a sorting algorithm constructively in principle but this is just as complex as a direct implementation. The significant advantage of descriptive descriptions comes to the fore in particular for complex algorithms, as they do not anticipate any specific form of implementation. Specifications are therefore much easier to understand, in particular compared to optimized implementations.

Equations as Assignments

The only task of a `set` method is to set the potentially encapsulated attribute:

```
context void setAttr(Type val)
pre:  true
post: attr==val
```

OCL

This specification is suitable both for tests of the method `setAttr` and for a constructive transformation into an implementation. If, namely, we replace the equality operator == with the Java assignment operator =, we can use the postcondition as the implementation. However, this implementation is only really correct if there are no further invariants that necessitate an additional modification of other attributes.

Unfortunately, postconditions can only be transformed constructively under certain, very narrowly defined usage conditions. Typically, a postcondition may only consist of a conjunction of assignments to local variables and later assignments must not invalidate earlier assignments. For example, `val==attr` is equivalent to the above postcondition but cannot be transformed directly into code in this form.[8] The following condition is also unsuitable for code generation as it contains cyclical dependencies:

```
context void method(Type val)
pre:  true
post: a==b+1 && b==2*a-val
```

OCL

Transforming this condition constructively requires that the linear set of equations is solved, with the result being the following constructive formulation:

```
context void method(Type val)
pre:  true
post: a==val-1 && b==val-2
```

OCL

There are a number of sophisticated techniques for interpreting conditions that were developed for various high-level languages constructively. Of these, the Horn clause logic from Prolog [Llo87], the evaluation of algebraic specifications formulated in the logic of equations [EM85], and the addition

[8] Strictly speaking, `val=attr` may be permitted in Java but then has a different effect to that desired.

of conditionals to these specifications are worthy of particular mention. For example, the following specification can also be transformed constructively:

```
context int abs(int val)
pre:  true
post: if (val>=0) then result==val else result==-val
```

OCL

Types of Generation

As the UML/P notations are used for modeling exemplary and complete structures and behavior, they are suitable for different forms of generating code. Fig. 4.3 shows the types of diagrams used for system and test generation. However, not all concepts of UML/P documents are suitable for code generation. UML/P allows the abstraction of details in principle—for example, via the omission of type information for attributes or via underspecification for methods and transitions—which means that the ability to generate code and tests from a UML/P artifact depends, amongst other things, on the completeness of the artifact. Intelligent generation algorithms can of course also use incomplete artifacts to generate code. They do so by filling out open aspects with defaults or guessing them intelligently. Thus, for example, a standard error behavior can be added to an incomplete Statechart and, for attributes for which there is no type information, generators can attempt to use type inference to compute the required type at the points where the attribute is used.

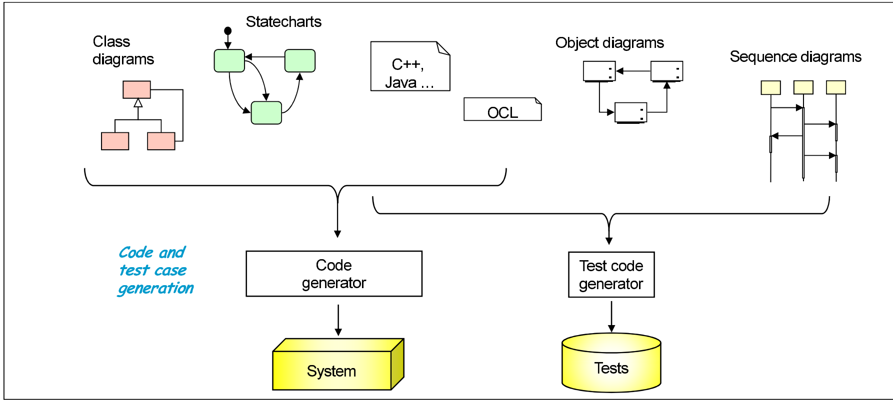


Fig. 4.3. Generation of code and tests from UML/P

Depending on the intended purpose of the generated code, we can use instrumentation to add additional test code to this generated product code. For test purposes, therefore, we can integrate inspection methods, interactive stop points, functions for accessing private attributes, or a check of invariants

in product code even though these are omitted when we generate the product code for use as a finished product. This form of instrumentation would be problematic if the optional code contains side effects that change the behavior of the instrumented product code. It is important, therefore, that this type of instrumentation is performed not manually but by code generators so that side effects that modify behavior can be excluded. In concurrent systems, the modification in the time behavior that arises as a result of the instrumentation must be examined in more detail.

4.1.3 Tests and Implementation from the Same Model

Models can be used on the one hand to generate tests, and on the other hand to generate constructive code. However, generating both types of code from the same model cannot create any additional confidence in the accuracy of the resulting system: if we generate incorrect implementation code from an incorrect model and also use the model to derive the tests for this implementation, the tests are equally incorrect. This is demonstrated by the following simple example intended to compute the absolute value of a number:

OCL
```
context int abs(int val)
pre:  true
post: result==-val
```

The code generation can therefore create the following Java code:

Java
```
int abs(int val) {
  return -val;
}
```

A typical collection of tests requires multiple input values for testing. Collections of numbers in the form $-n, -2, -1, 0, 1, 2, n$ for some large n have proven to be good standard values for the data type `int`.[9] Normally, these representatives and limit values are specified by the developer. The expected results do not have to be computed separately as the postcondition provides an opportunity for checking whether the result is correct. It would be possible to generate the following test code for our example:

Java/P
```
int val[] = new int[] {-1234567,-2,-1,0,1,2,3675675};
for(int i = 0; i<val.length; i++) {
  int result = abs(val);
  ocl result==-val;
}
```

Since the test code would be just as incorrect as the implementation, the error would not be recognized. In this type of situation, it is not the implemented code that is tested; rather, the test determines only whether the code

[9] The selection of the numbers is based on a simple classification for the values `int` and the use of individual representatives and limit values from the classes formed.

generator is working correctly. If an error were reported in this situation, it would only indicate an inconsistency between the generated code and the (also generated) test driver. This technique is interesting mainly when the parameterization of the generator shall be tested.

A consequence of this observation is that we have to model the constructive model used for code generation separately to the test model. However, we can represent fragments of the test model and the constructive model in the same diagrams. Nevertheless, there must be a clear distinction between the concepts used for different purposes. For example, Statecharts are used often constructively, but the state invariants can be used in the Statecharts for checking in tests.

4.2 Code Generation Techniques

4.2.1 Platform-Dependent Code Generation

Although the fact that UML/P has been defined based on the programming language Java means that a significant design decision has already been taken, the form of the code generated is not fixed. There are multiple dimensions of variations that have to be taken into account for code generation, including the platform dependency discussed here, which is particularly important for embedded systems or the cloud.

Depending on the target platform, different mechanisms are available—for example, for handling communication in a distributed system with the controlled systems, neighboring systems, the cloud, or users, as well as for storage and troubleshooting, or ensuring security, data authenticity, and data integrity.

These mechanisms can be dependent on the hardware in which the software is embedded or may have to be adapted to the available class libraries or APIs. The pieces of code to be inserted into generated code cannot be predicted by the code generator because new platforms, new control devices, or new versions of class libraries can lead to constant and fast changes, for example. It is therefore essential that the code generation can be adapted flexibly to the respective usage conditions. There are two main approaches here:

- Generation of abstract interfaces, as illustrated in Fig. 4.4
- Parameterization of the code generation, as illustrated in Fig. 4.5

With regard to separating platform-specific and hardware-independent code, the creation of an abstract interface and thus separating the layers is an ideal approach that improves the portability of software. Many of the Java APIs have been defined precisely for this purpose and have become the standard. [SD00] examines this strict separation of code into application

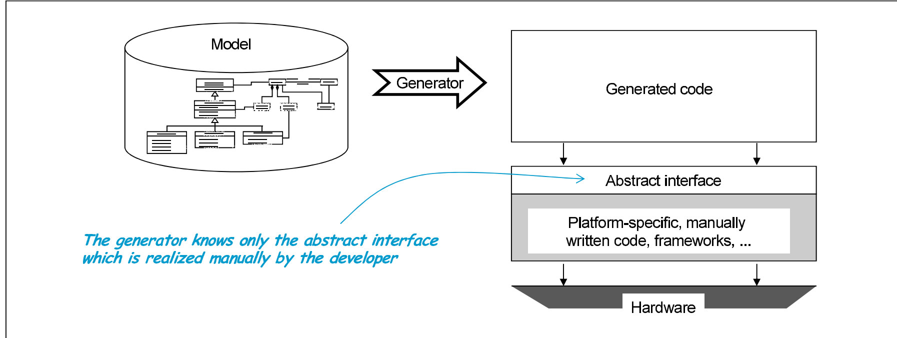


Fig. 4.4. Generation of code for an abstract interface

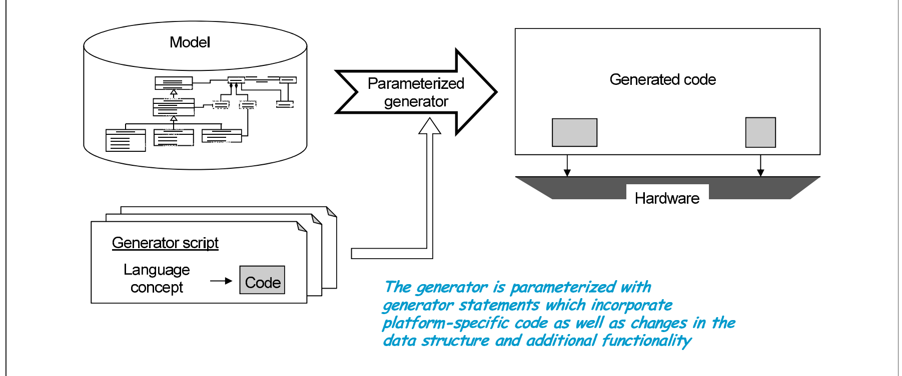


Fig. 4.5. Parameterized code generation

code ("A-code") and platform-specific, technical code ("T-code") in more detail and identifies necessary mixed forms. One of the results of these investigations backed up by practice is that a standardized "T-architecture"—that is, the technical code for saving, display, error processing, and similar standardizable technical functionalities—has a high potential for reuse. In code generation, we can exploit this potential by flexibly combining T-architecture parts with application (A) models provided by the developer.

As always the strict separation of the two code types can lead to inefficiencies when we introduce layers and adapters. For example, our auction system is based on the asynchronous communication of messages. However, if the abstract interface provides only an RPC mechanism, the buffering of the messages (amongst other things) must also be coded. At the lowest level, however, communication is again asynchronous via the Internet, where buffer mechanisms are already integrated. We can increase efficiency significantly, for example, by giving up the conceptual layer formation and

by "interweaving" higher and lower layers.[10] Alternatively, we can create the abstract interface with a broad scope and in the example, offer both synchronous RPC and asynchronous communication. However, this leads to significantly increased efforts for the realization and evolution and pays off only if the interface is reused often enough in other projects.

If we also assume that the generation of the target code is correct and waive manual postprocessing or inspection, adherence to architectural guidelines such as layer formation in the generated code is not very relevant. Instead, during generation, we can focus more on efficiency and, similarly to optimization techniques for the compiler, we can interweave platform-independent and platform-specific code. According to [SD00], this gives rise to an AT-code that is very difficult to maintain; it contains both application knowledge and technical knowledge. This is another reason why it is essential that only the initial models, separated by A and T aspects, and the generator scripts are changed manually, but not the generated code.

In practice, we can assume that a mixed form made up of both generation mechanisms will lead to the best results. Furthermore, a system will have a further component that provides a runtime environment for certain functionalities that cannot be mapped into either Java class libraries or Java language concepts. These additional components include extended functionality for handling the sets and lists available in OCL and for processing state models stored explicitly in the code. Fig. 4.6 therefore shows the principle structure of a code generator.

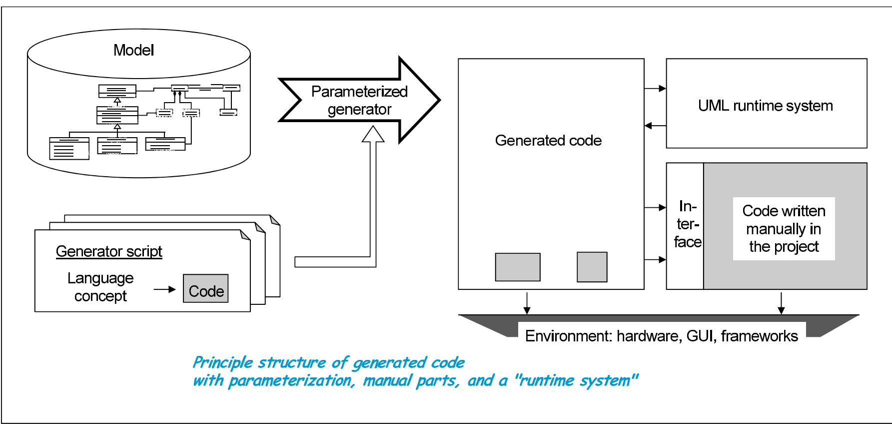


Fig. 4.6. Structure of a code generator

[10] These layers actually also arise here, but the layer architecture is developed not horizontally (i.e., layer for layer) but vertically, meaning that we can take the needs of higher layers into account directly when designing the lower layers. Developers, e.g., can change from RPC to the asynchronous form of communication efficiently with refactoring techniques [Fow99].

The term "UML virtual machine", defined in [BBWL01] and [RFBLO01] for example, corresponds to the right-hand part of Fig. 4.6, which consists of the UML runtime system and a platform-specific implementation of the defined interfaces. Based on the "Java virtual machine" (the interpreter of the Java bytecode), the code generator corresponds to the Java compiler. The "UML virtual machine" represents a type of operational semantics of the executable part of UML/P.

4.2.2 Functionality and Flexibility

We can use the parameterization of a code generator discussed in the last section not only for adaptation to platform-specific features, but also for adding additional functionality to the generated code. In principle, there is a lot of flexibility possible. This is discussed below using the simple and widely known example of code generation for attributes in the class diagram.

An attribute defined in a class of the class diagram has the "natural" transformation as an attribute in the generated Java code that is demonstrated in Fig. 4.7. With the exception of the tags for derived and read-only attributes (/ and `readonly`), all tags, types, and initial assignments to the attribute can be transformed directly. However, this direct transformation bears some disadvantages, such as the broken encapsulation and unsynchronized access by other objects.

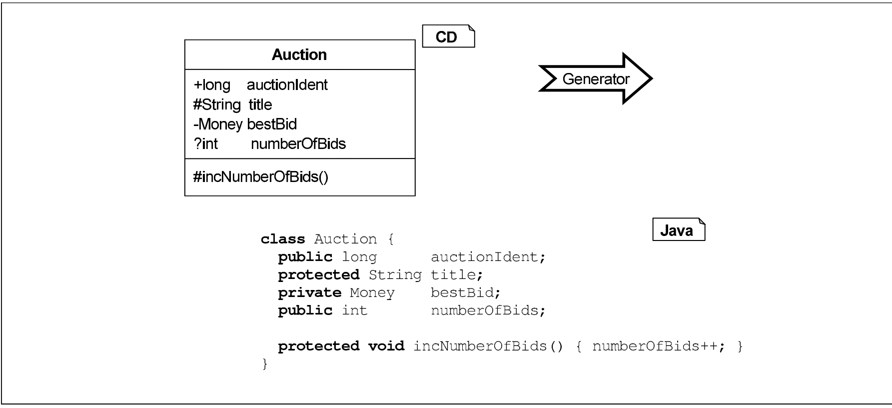


Fig. 4.7. Direct transformation of attributes

It is therefore not common practice today to transform attributes from analysis and design models directly into attributes of the implementation. Instead, it is usual to provide an infrastructure in the form of `get` and `set` methods. Fig. 4.8 shows the resulting code structure. Here, the attribute name is often preceded by a suitable prefix, (for example, the underscore "_"). The

use of access functions increases the flexibility. It allows, for example, the potentially necessary synchronization of threads or the realization of the access right `readonly` through two `get/set` methods with different visibilities.

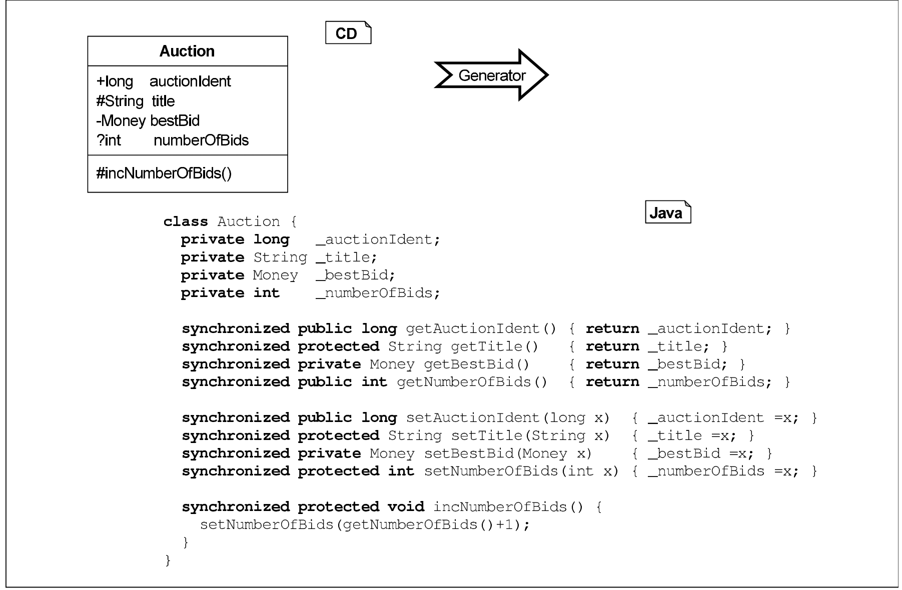


Fig. 4.8. Transformation of attributes using access functions

The transformations illustrated in Fig. 4.7 and 4.8 are today relatively widespread but are by no means the only ones. There are further variants which allow, for example, persistent attributes, storage of attributes in Enterprise JavaBeans [BR11], the propagation of attribute changes, and so on. To enable the various and generally unpredictable variants of code generation flexibly, however, we generally have to parameterize the transformation of concepts of UML/P heavily and allow additional functionality to be added on a technology-specific basis.

To a certain extent, we can identify "APIs"[11], which each have to be transformed, for the UML/P concepts. For example, for the concept "attribute" of the class diagrams, we can identify the following API at least:

- Setting of an attribute
- Reading an attribute
- Initialization of an attribute with a default value

We can define extensions of this API for the following:

- Serialization

[11] API in the sense of an *abstract programming interface*.

- Loading from and storing into a database
- Screen output and reading in from a screen

We can also define extensions of the API for type-specific functionalities. These include, for example, incrementing number values, appending strings (analog to the Java operator `+=`), or handling individual elements in container structures.

The desired flexibility for the code generation arises, therefore, not only in the form of the transformation of UML/P concepts, but also in the functionality offered by the generated code. The functionality offered not only has to be generated but must also be available in a form that allows developers to access it in other places. There are two general approaches:

1. The functionality of the generated API is disclosed, for example, in that the name and signatures of the respectively usable functions can be derived uniquely from the name and the type of an attribute.
   For example, if the attribute `title` is defined in the class diagram, it can be accessed in Java with `getTitle()` and `setTitle(...)`.
2. The model API itself is disclosed but the transformation remains hidden. The manually written Java code therefore uses the API directly and also has to be transformed during the code generation.
   In the example, the attribute `title` is then also used in the Java code. Depending on the form of use (read or write), during the code generation this is replaced with a `get` or `set` method.

While the first approach leads to more simple code generators and requires no handling of the Java code, the second approach offers more flexibility and coding security. Hiding the actual implementation allows us to replace it or supplement it relatively easily. Furthermore, the use of the API is more abstract and results in more compact code. In this approach, however, code parts formulated directly in Java also have to be transformed during the code generation.

Already the relatively simple example for realizing attributes shows some of the many effects that may occur. Therefore, Section 4.4 below first describes a readable form of the representation of code generations and demonstrates their applicability based on the transformation of attributes.

In some circumstances, the forms of code generation available for transforming UML/P concepts into the implementation have an impact on the semantics of the concepts. This is dangerous but is also an opportunity for UML users to use *semantic variations*—often referred to as *"variation points"*—to integrate project-specific or additional functionalities and capabilities. It is not possible to describe these variations precisely within UML: on the one hand because UML itself does not provide elaborated mechanisms for defining the semantics of the variations; and on the other hand because these variations are not valid generally but are instead dependent on potential target

platforms and desired functionalities.[12] In practice, therefore, the only option is to specify individual forms of code generation and the thus intended semantic interpretation in a constructive form. An extensive description of the number of possible variants across the entire spectrum does not appear to be possible. In [Grö10], feature diagrams were used to define variation points in languages and, for example, were applied to multiple diagrams in [GRR10] and [GR10].

The separation of the functionality and the platform-dependent code parts has already been discussed in depth in the field of aspect-oriented programming (AOP) [KLM+97, LOO01] and the closely related generative programming [CE00]. These types of programming include techniques that allow a further separation of independent program aspects. Special techniques ("weaving") allow us to combine code parts that are initially formulated independently of one another and usually connected only via an abstract programming interface. This approach is also used in a restricted form for the code generation discussed here.

We can also use a generative approach to realize the composition of classes from *features* discussed in [Pre97], [Pre00], and [KPR97]. Here, we can use suitable stereotypes and tags to determine which class receives which (additional) features as part of the generation.

4.2.3 Controlling the Code Generation

In order to fully make use of flexibility in the code generation, we must be able to control the transformation appropriately. Therefore, it is often is necessary, to realize different instances of the same concept differently within one project. The options for transforming attributes (discussed many times already in this book) can be dependent on the following, for example:

- The class that contains the attribute—because this class can have tasks such as data storage or application control or can act as an interface to other system parts and must therefore be synchronized
- The task of the attribute within the class—because, for example, the class is persistent but the attribute can be computed or contains only temporary data
- The type of the attribute—because this exists in UML/P, for example, but not in Java

We can supplement these static dependencies, defined at the time of the code generation, with dynamic dependencies if, for example, we use a flag to define whether an object should be persistent. Project-specific cases such

[12] At best, we can indicate a special meaning of a construct using stereotypes. In UML, it is not possible to define the stereotype and the intended semantics precisely. Instead, we can use an informal description, as presented in Section 2.17, Volume 1.

as these should typically no longer be realized by a predefined definition of semantics but rather through a self-defined piece of code that is added systematically by the code generator parameterized with templates.

We can generally control the form of implementation of any UML/P concept with stereotypes and tags. In practice, however, this is not sufficient. It is more practical to label UML diagrams with stereotypes, optionally supplemented with additional parameter values and use additional *templates* or *scripts* that are executed by the code generator and that parameterize the generator very flexibly, as illustrated in Fig. 4.6 for the transformation.

4.3 Semantics of Code Generation

In principle, code generation is the transformation of a model from one language into another language. A definition of semantics is also essentially a mapping of a language that is considered as unknown (here UML/P) into a *target language* considered to be known and understood. Furthermore, if the target language is *formal* and the mapping is formulated precisely, this is referred to as *formal semantics*. In this sense, a mapping of UML/P to the programming language Java implemented via a program can itself be understood as formal semantics [HR04]. However, there are several points that have to be taken into account in this argumentation:

1. To understand a complex language such as UML/P, it is helpful to have more than one explanation of the meaning (semantics). Using multiple approaches to define the semantics of a language allows different problems to be recognized and thus integrated in the language definition itself and when this definition is used (analysis, tests, refactoring).
2. In most cases, the generated code is not particularly easy to read, as it generally contains a multitude of technology-specific or framework-specific details that do not contribute to the actual semantics of the model. It is not generally acceptable when developers have to inspect the generated code to understand the meaning of a modeling language. However, there are a number of language descriptions that are based on discussing the principle of code generation in general terms. They appeal to a wide audience as today, programming languages such as Java are the most widespread "formal languages."
3. During the transformation into executable code, certain aspects of a language cannot be transformed or can only be transformed with a lot of effort. These aspects especially include concepts that allow *underspecification* and are integrated in UML/P at multiple points. For example, initializing values for attributes can be missing or multiple alternative transitions can be enabled in the Statechart simultaneously. For code generation, the resulting nondeterminism is usually replaced by the selection of a transition with a higher priority (see Section 5.4.4, Volume 1).

The system described by the generated code is therefore not generally identical to the initial model and instead represents one of multiple possible *specializations*. Because a programming language is executable per se, the mapping of underspecification and thus a complete definition of the semantics of UML/P according to Java, for example, is generally not possible.

4. UML/P is designed to a certain extent for executability, but also allows to use nonexecutable concepts at many occasions. As already mentioned, model information can be omitted. It is, for example, also possible to specify conditions with infinite quantifiers. Automatic transformation of OCL postconditions like discussed below to executable Java also belongs to this group of problems. Therefore, a complete mapping of UML/P to Java is not possible.

As an alternative to very implicit definitions of semantics, we can use techniques of the formal methods to define formal semantics for the source language independently of any form of code generation. These types of definitions of semantics are typically mappings that transform a UML model into a suitable target language. They use mathematically formal calculuses as target languages and the mappings are defined compactly so that they are more easily accessible for an analysis.[13] As argued in [HR00] and [Rum98], the existence of two mappings for a source language can be used to increase the confidence in the correctness of both mappings and thus in the code generation in particular.

If the code generation in the form proposed here is parameterized by scripts that influence the behavior and structure of the generated code, we can integrate this into a formal definition of semantics in two ways. Fig. 4.9 formalizes a variant for a definition of semantics in which the mapping of the semantics is independent of the script used.

The formalization represented in Fig. 4.9 uses a set-valued semantic mapping on a system model [BCGR09b] to thus represent the variability of the parameterized code generator. Such a formalization is heavily dependent on the *observed* aspects of a language and its models. If, for example, only the externally visible behavior is formalized, we have freedom in terms of the transformation of attributes, associations, and other structure elements. For such a big language as UML, it is actually not practical to fully formalize it even though there is a remarkably complete but not overly elegant formalization in [Öve00]. Instead, it makes sense to highlight individual, critical aspects more precisely and thus give feedback into the standardization process. The principle advantages and problems of a standardization have been discussed in a number of publications [BHH+97, FELR98b, FELR98a].

[13] A different form, for example, an axiomatic definition of semantics for UML or individual parts, can be found in [EHHS00] and [EH00b] and is based on a transformational approach that is based on graph grammars.

The following definitions are required to **formalize language and code generation**:

- The source language (UML/P) as a set UML of syntactically well-formed expressions
- A suitable formal target language Z
- The script language of the code generator with the vocabulary S
- The set J of all Java programs

A *code generator* is a (sometimes partial) algorithmic mapping $Gen : UML \rightarrow J$.
In contrast, *formal semantics* is a mapping $Sem : UML \rightarrow \mathbb{P}(Z)$. This maps a single, typically underspecified and abstract model from the source language to the set of all possible implementations. This mapping represents a form of *loose semantics*.
For a comparison of both mappings Sem and Gen, we need a form of semantics for Java programs such as $Sem_{Java} : J \rightarrow Z$. For each UML document $u \in UML$ for which code can be generated, the following must apply:

$$\forall u \in UML : Sem_{Java}(Gen(u)) \in Sem(u)$$

This means that generally, the code generator selects one of multiple possible implementations by filling out open aspects with defaults. It is only when $Sem(u)$ represents one single element that the specification was obviously complete and unambiguous.
A *parameterized code generator* is extended by the parameter, i.e., the script language S: $Gen_p : UML \times S \rightarrow J$. The following must now apply:

$$\forall u \in UML, s \in S : Sem_{Java}(Gen_p(u, s)) \in Sem(u)$$

This means that within the scope of the specification by $Sem(u)$, the script s may select a possible implementation for u.

Fig. 4.9. Semantics of the parameterized code generator

The formalization represented in Fig. 4.9 has an alternative viewpoint: To a certain extent, the source language and script language together represent a “programming language”. A definition of semantics can reflect this in the form of a function $Sem_p : UML \times S \rightarrow Z$ which selects precisely one element of the target language Z.

4.4 Flexible Parameterization of a Code Generator

Code generators and general transformations need suitable forms of representation. Chapter 5 demonstrates the properties of these representations in forms of scripts using an example transformation of the UML/P constructs. The concepts discussed here refer primarily to a code generator but can also be applied to other forms of analysis tools and transformers. For example, they can be used to describe changes in data structures or refactoring techniques.

4.4.1 Implementing Tools

Section 4.2.3 discusses the use of scripts and templates for a flexible parameterization of a code generator but also for analysis and test tools. A platform-specific code generator designed for different implementation options does actually require a flexible mechanism for controlling the code generation. In principle, we can control the generator with stereotypes and tags. However, we cannot document the details of the code to be created, or at least we cannot do so comfortably, using stereotypes.

In general, the code structures to be created can be rather complex and can take a wide variety of forms. Symbol tables are generally necessary and auxiliary conditions that describe the applicability of a code generation or possible optimizations must be checked. Due to the required flexibility, only a programming language that is (generally) complete (in terms of expressiveness) allows a compact formulation of conditions and transformations. Transformations formulated in this language are executed by the code generator in order to generate the implementation code. They are not present themselves at runtime. Fig. 4.10 shows a typical internal structure of a generator.

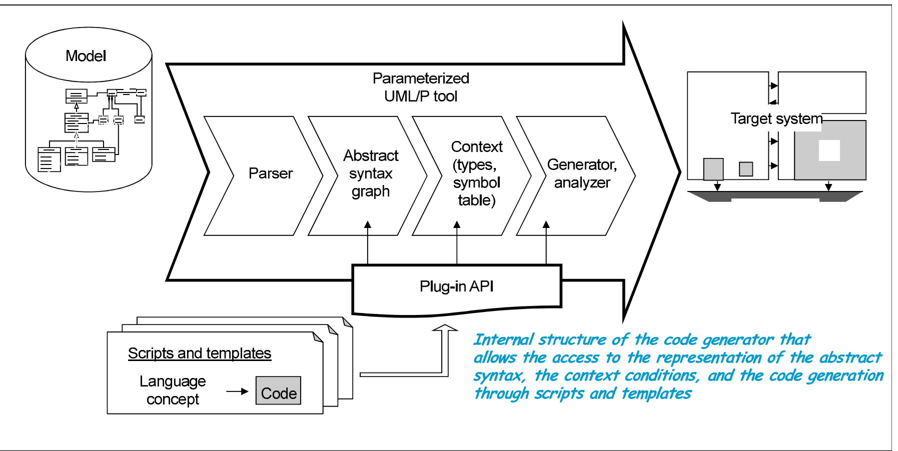


Fig. 4.10. Internal structure of a code generator

Various suggestions have been made for script or template languages in tool development. For example, interpreted derivatives of the language C, Visual Basic, or script languages such as Pearl, or Tcl/Tk are used. Functional programming languages such as ML [Pau94, MTHM97] also offer a very compact form of parameterization for such tools. For example, the verification tool Isabelle [NPW02] is written in ML and offers good extension options which also have to be formulated in ML.

Another alternative is the now popular XML technology [McL06, W3C00] and the use of XML/XSLT-based tools for transformation into the target code. However, due to the explicit embedding of the tags (nonterminals), on the one hand, XML often has a very poor ratio between usage information and structural overhead, and on the other hand, the current parsing and transformation tools for XML are still not as powerful as `Yacc/Lex` and its derivatives like `AntLR`, which have been used in the domain of compiler construction for a long time.

With its multitude of libraries, frameworks, and tools, as well as the option of loading class libraries dynamically, Java is also a good candidate for, on the one hand, realizing a code generator, and on the other hand, enabling parameterization via a plug-in mechanism. The `Poseidon` tool [BS01a, BBWL01] uses this mechanism, for example.

However, an active script language has some uncomfortable deficits with regard to the composition of artifacts of the target language. Therefore, a macro replacement mechanism that processes templates and generates code from them is also an option. In this sense, a template is a passive script language that contains macros that are replaced with real names or other language expressions by the template execution engine. A template can contain active elements to thus enable control structures, such as alternatives, repetition, or the integration of other templates, but also to perform calculations to a limited extent. Here, XML tools based on the transformation language XSLT, or a template mechanism combined with Java code similar to the JSP pages [FK00] presented for example by [SvVB02] or FreeMarker [Dib01] can be used.

The fact that the generator can be programmed flexibly means that developers have to develop not only in the target programming language but to a limited extent also in the script programming language. Due to the technology-dependent adaptations required on a regular basis, this additional development must take place parallel to the actual development and therefore often in the same project. For the learning curve, therefore, it would be an advantage if the script and target programming languages were similar.

The generator framework MontiCore [KRV10, Kra10, KRV08, GKR$^+$08] uses the template-based Java Template Engine FreeMarker [Dib01] to generate code from UML models, for example. The separation between the processing frontend (parser and analysis of the context conditions) and the generating backend allows us to adapt the generated code flexibly. This is demonstrated in particular in the code generator for UML/P which is based on this adaptivity [Sch12]. Here, the FreeMarker script language and the target language appear in interleaved form. In order to manage the templates and thus the target structure of the code easily, a structuring of the templates along the target structure of the code and support by MontiCore basic functions is proposed. The MontiCore generator framework is realized in Java and thus the UML/P generator [Sch12] can itself be extended with its own

basic functions. Normally, however, it should be sufficient for the user to cope with the template language.

If it was decided to take a script or a template language, it needs to be clarified how we can represent the effects of the scripts for the user of the generator in a compact, understandable, yet informal form that is usually not based on the script language (which includes a lot of implementation details). This desire is even more understandable if we have to use the XML-based, very verbose transformation language XSLT. However, this book focuses less on the specific formulation of the code generation but rather on the concepts of transformations, which is why an abstract representation for transformations is selected.

4.4.2 Representation of Script Transformations

The effect of a script used for code generation is based on transforming UML/P concepts into the target programming language Java. As this type of script also handles borderline and special cases, checks usage conditions, and uses a number of auxiliary functions to do so, it is practicable to represent the effect of such a script in compact form and to discuss special cases only informally so that they are understandable for the user. The template in Table 4.11 is proposed for this purpose. In addition to specifying the actual transformation and context conditions, it can be used to give a general description and to discuss potential alternatives. This description should be seen neither as complete nor as formal, although it is subject to some restrictions as described below. Instead, the template represents an abstract illustration for describing transformations.

Name of the transformation	
Explanation	Motivation and purpose of the transformation

Transformation rule	There is usually a primary transformation rule which is represented in the following form:
	`Origin`
	⇓
	`Target`
	• The `origin` and `target` can each be represented in text form or via a diagram.
	• The `origin` introduces the elements of the syntax class diagrams and EBNF products that are transformed. The context is also listed. Template diagrams and text blocks are used for this.

(continued on the next page)

(continues Table 4.11.: Name of the transformation)

	• The italic names represent *schema variables* that are assigned real language elements when the transformation is applied. • Explanatory texts describe the types of schema variables and which parts are optional or can occur multiple times, etc. • The contents of the schema variables themselves can be subject to certain transformations. For example, the schema variable *attr* can be used to construct a method name `set`*Attr* composed of the constant part `set` and the capitalized form of the attribute name.
Further transformations	The main transformations usually give rise to additionally required transformations that are represented in the same way. These transformations refer, for example, to elements of the API discussed in Section 4.2.2 that must also be transformed.
Noteworthy	This section rounds off the description with additional considerations, notes, and the discussion of potential problems.

Table 4.11. Name of the transformation

The rule mechanism used is based on formal rule calculuses in the following form:

```
 Origin  ||  Usage
 ---⇓--- ||  conditions
 Target  ||
```

In addition to the `origin` and `target`, these calculuses contain a precise specification of the usage conditions in a formal notation. The origin contains schema variables (placeholders, [BBB^{+}85]) that are assigned real language elements when the transformation is applied. Each schema variable belongs to to a certain nonterminal and is therefore *typed*.

Below is a sample application for a standard transformation which executes the example code generation shown in Fig. 4.8.[14]

As already discussed in Section 4.2.2, the transformation represented here is just one of many options that can be chosen, for example, by applying suitable stereotypes and tags to the attribute, the class, or the class diagram, but also by specifying certain scripts.

[14] However, without the use of leading underscores for attributes propagated in some coding standards.

Attribute1: Standard transformation of attributes

Explanation

The standard form for the transformation of attributes with encapsulation and access via explicit access functions.
It does not contain type-specific or project-specific functionalities.

Attribute definition

CD

Tags for visibilities and further tags

Class ...

tags Type attr = value

Optional initial value

This pattern describes the source structure for the transformation and the "placeholders" (pattern variables) are assigned for the transformation. A placeholder is shown in italic font.

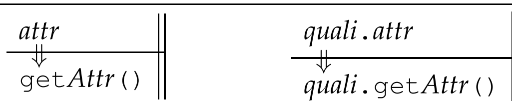
⇓

Java

```
class Class { ...
  private Type attr = value;
  tags' synchronized Type getAttr() {
      return attr;
  }
  tags" synchronized Type setAttr(Type a) {
      return attr=a;
  }
}
```

- The optional assignment with =*value* is used only if it is specified in the diagram.
- The visibility specified in *tags* is adopted with the exception of `readonly`. `readonly` is transformed into `public` for `get`*Attr* (i.e., in *tags*′) and into `protected` for *tags*″.
- `return` makes sense within `set`*Attr*, as the assignment with = has the same value.
- The multiplicity of the transformed attribute is either not specified and therefore “`1`”, or is “`0..1`”.

Attribute access



- The type of expression *quali* conforms to class *Class*.

(continued on the next page)

(continues Table 4.12.: Attribute1: Standard transformation of attributes)

Attribute assignment	*attr*=*expr* ⇓ `set`*Attr*`(`*expr*`)` || *quali*`.`*attr*=*expr* ⇓ *quali*`.set`*Attr*`(`*expr*`)` || • The type of expression *quali* conforms to class *Class*. • The type of expression *expr* conforms to the attribute type *Type*.
Noteworthy	Type-specific constructs such as *attr*++ can either be transformed efficiently via additional functionality or transformed into `set`*Attr*`(get`*Attr*`()+1)`. Tags and stereotypes are not considered in this standard transformation. In UML, attributes can be given a multiplicity. Multiplicity "*" enforces a transformation similar to associations.

Table 4.12. *Attribute1:* Standard transformation of attributes

5

Transformations for Code Generation

> If you have clear concepts,
> you know how to give orders.
>
> Johann Wolfgang von Goethe

This chapter adds specific techniques and transformations to the basic idea of code generation as introduced in Chapter 4. Sections 5.1 to 5.5 explain the approach for transforming class diagrams and object diagrams, generating code from OCL, executing Statecharts, and generating tests from sequence diagrams in Java. For class diagrams this chapter discusses known concepts and some alternatives in a compact transformational form. Section 5.1 demonstrates how we can systematically represent transformation rules for code generation from class diagrams.

The chapter also discusses alternatives for generation of code from Statecharts in great detail. For all other notations, this chapter primarily focusses on the basic principles of transformation allowing users to develop forms of transformations tailored to their target language or target environment or to perform transformations manually in the absence of a suitable tool.

5.1 Transformations for Class Diagrams ... **100**
5.2 Transformations for Object Diagrams ... **123**
5.3 Code Generation from OCL ... **132**
5.4 Executing Statecharts ... **146**
5.5 Transformations for Sequence Diagrams ... **155**
5.6 Summary of Code Generation ... **158**

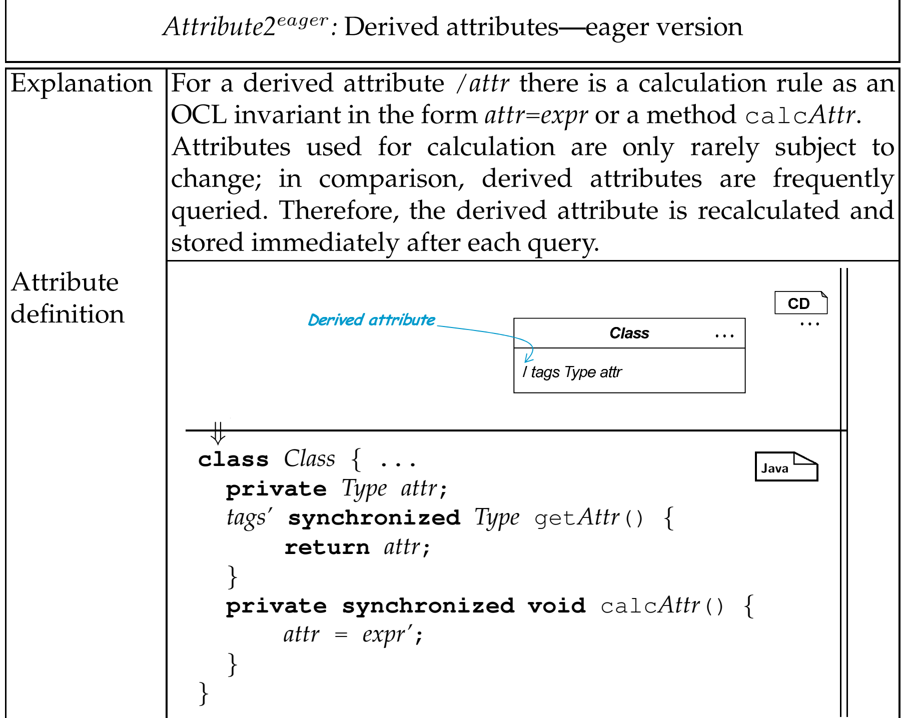
© Springer International Publishing AG 2017
B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_5

5.1 Transformations for Class Diagrams

Using a collection of transformation rules, this section describes how to transform class diagram and some related language constructs into Java as a more complex example of a transformation. These transformation rules also indicate the options for describing alternatives and compositions. However, the transformation described here does not consider Java frameworks or infrastructure concepts such as JavaBeans or middleware components; neither does it consider database connections. Each of these cases requires special generators and cannot be discussed here in the general form intended.

5.1.1 Attributes

For normal and static attributes of classes, the transformation rule *Attribute1* on page 96 already specifies a rule for transforming these attributes. In principle, we can also use this rule for *derived attributes*, although it does ignore a significant feature of this type of attribute: for a derived attribute, there is usually a calculation rule formulated as an invariant. Alternatively, there may already be a side-effect-free method formulated in the target language for calculating the attribute. According to our naming convention, such a method is called `calc`*Attr*.



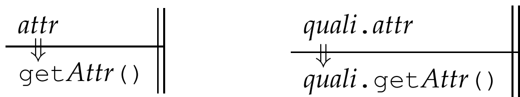
Attribute2^{eager}: Derived attributes—eager version

Explanation	For a derived attribute `/`*attr* there is a calculation rule as an OCL invariant in the form *attr=expr* or a method `calc`*Attr*. Attributes used for calculation are only rarely subject to change; in comparison, derived attributes are frequently queried. Therefore, the derived attribute is recalculated and stored immediately after each query.
Attribute definition	


```
class Class { ...
  private Type attr;
  tags' synchronized Type getAttr() {
      return attr;
  }
  private synchronized void calcAttr() {
      attr = expr';
  }
}
```

(continued on the next page)

(continues Table 5.1.: $Attribute2^{eager}$: Derived attributes—eager version)

	• The attribute definition and `get`*Attr* are transformed in the same way as for the standard rule *Attribute1*. However, there is no `set`*Attr* method. • If we specify the calculation rule as a nonrecursive OCL constraint in the form `attr=expr`, it is transformed into the Java code `expr′` and embedded in the method `calc`*Attr*. Alternatively, this method may already exist or be specified in the given form by a postcondition.
Attribute access	 • As in the transformation rule *Attribute1*.
Attribute assignment	Not possible.

Assignments to a source attribute	The source attributes for the calculation can be determined by analyzing the OCL expression or the existing `calc`*Attr* implementation. The attribute *attr* is derived from these source attributes. For each source attribute *source*, the `set` method is extended:
	⇓
	Java
	`class Class { ...`
	`  tags′ synchronized Type′ setSource(Type′ a) {`
	`      source=a;`
	`      calcAttr();`
	`      return a;`
	`  }`
	`}`
	• This shows only the (simple) case in which the source and derived attributes are localized in the same object. If this is not the case, we must ensure that there is a bidirectional connection between the involved objects. This connection should not be affected by the current modifications.[1]

(continued on the next page)

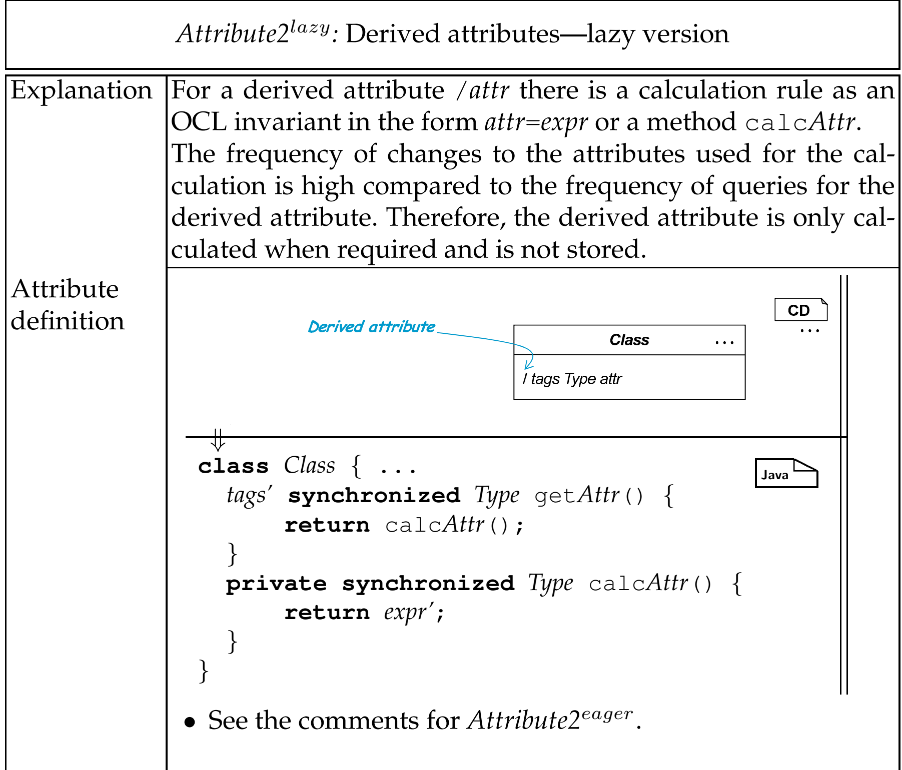
[1] A bidirectional association is necessary because the calculation does not follow the control flow; instead, it realizes a form of change propagation (publisher-subscriber pattern) and thus inverts the calculation order. A data flow analysis can check whether this pattern has to be integrated, with all of its infrastructure, or whether a bidirectional association is present.

(continues Table 5.1.: *Attribute2*^{eager}*: Derived attributes—eager version*)

Noteworthy	Modifying an attribute automatically modifies all attributes derived from this attribute. This can lead to cascaded recalculations of derived attributes and can be inefficient if there are more modifications than queries executed. Furthermore, circular dependencies lead to nonterminating recalculations and are therefore forbidden for this realization.

Table 5.1. *Attribute2*^{eager}*:* Derived attributes—eager version

The "eager" version of the transformation formulated above can be contrasted with a "lazy" version which calculates the attribute value only when it is required. Due to the similarities with the previous transformation rule, the rule is given in Table 5.2 in an abbreviated form.

Attribute2^{lazy}*:* Derived attributes—lazy version	
Explanation	For a derived attribute */attr* there is a calculation rule as an OCL invariant in the form *attr=expr* or a method `calc`*Attr*. The frequency of changes to the attributes used for the calculation is high compared to the frequency of queries for the derived attribute. Therefore, the derived attribute is only calculated when required and is not stored.
Attribute definition	

- See the comments for *Attribute2*^{eager}.

(continued on the next page)

(continues Table 5.2.: Attribute2^{lazy}: Derived attributes—lazy version)

Attributes	• The attribute access is the same as for *Attribute2*^{eager}. • In the same way as for *Attribute2*^{eager}, direct assignment of values to attributes is not possible. • The assignment of a value to a source attribute (that this derived attribute is dependent on) does not have to be adapted.
Noteworthy	The advantage compared to the *Attribute2*^{eager} version is that the control flow was not inverted and therefore no bidirectional associations or other infrastructure are necessary. However, repeated calculation of the attribute can lead to inefficiency. Here again, circular dependencies lead to nonterminating recalculations and are therefore forbidden for this realization.

Table 5.2. *Attribute2*^{lazy}: Derived attributes—lazy version

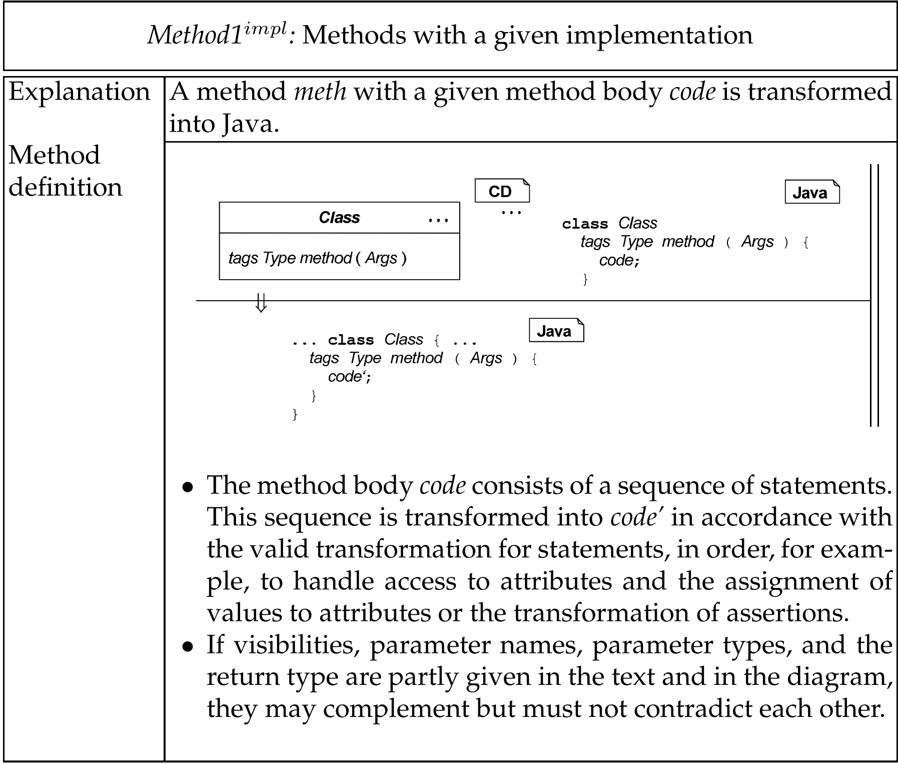
One form of the model-view-controller pattern applied occasionally for graphic interfaces uses advantages of both approaches: it merely notes the change propagation in a boolean status variable and a recalculation is only executed when required.

5.1.2 Methods

There are multiple strategies for implementing methods depending on the initial situation:

1. The method body has already been formulated in another artifact and only has to be inserted into the method. When this happens, the required transformations—for example, those of attribute accesses—are executed.
2. The method is described by a precondition/postcondition pair, whereby the postcondition, as discussed in Section 4.1, is formulated as an algorithm and can be transformed into code directly.
3. The method is described by a precondition/postcondition pair, whereby the pair cannot be transformed using an algorithm and is therefore suitable only for tests.
4. There is no implementation or specification for this method yet.

A separate approach is required for each of these cases. The first case requires only the integration of the method body with the signature and the transformation, for example, of the attribute accesses in the method body.



Method1^{impl}: Methods with a given implementation	
Explanation	A method *meth* with a given method body *code* is transformed into Java.
Method definition	• The method body *code* consists of a sequence of statements. This sequence is transformed into *code′* in accordance with the valid transformation for statements, in order, for example, to handle access to attributes and the assignment of values to attributes or the transformation of assertions. • If visibilities, parameter names, parameter types, and the return type are partly given in the text and in the diagram, they may complement but must not contradict each other.

Table 5.3. *Method1*^{impl}: Methods with a given implementation

Tools that are currently available offer several approaches for providing the method body. One option is to define the body as a piece of text attached to the method signature in the diagram (for example, as a comment) and to make it accessible in the editor when the method is selected. However, this is not practical for large systems with a lot of methods. The "roundtrip engineering" technique reads the method bodies directly from the source code and, subsequently, overwrites them.[2] This book proposes a further alternative that combines some of the advantages of aspect-oriented programming (AOP) with a compact denotation. The basic idea is to group method implementations according to functional criteria as well as according to the classes to which they belong. For example, the `protocol` methods of all classes, the methods for the iteration of a hierarchy of objects, or the serialization meth-

[2] "Roundtrip engineering" allows modifications in both the abstract diagram view and in the implementation, with the modifications then being imported in the respective other view. This approach is fragile, however, as according to the current state of technology, the connection between the two views is maintained using comments.

ods of multiple classes can each be grouped in one file respectively, which contains this *aspect* of the program.

We can use OCL specifications of methods in the form of pairs of preconditions and postconditions in addition to or instead of Java implementations. Section 3.4.3, Volume 1 discusses the integration of several method specifications for the same method. We can therefore assume a single pair here. If the specification has an algorithmic form as discussed in Section 4.1.2, we can generate code from it directly.[3]

As the transformation of a method specified in this way is based essentially on the transformation of OCL into Java code as discussed in Section 5.3, the transformation rule is not formulated explicitly here.

In the third case listed above we have both an implementation from somewhere and a separate OCL method specification. Therefore, the specification should be used for checking the condition during runtime. The generator knows whether it should generate efficient product code or code instrumented with these checks. For example, Eiffel and Java compilers only optionally encode assertions.

In principle only the precondition needs to be tested before the start of the method and the postcondition after the end of the method. Under some circumstances, however, variables defined locally with the **let** construct and any initial states of attributes used in the postcondition have to be stored. This may be a complex issue if the attributes used are located in other objects and the access paths themselves may have been changed. [RG02], for example, contains a more detailed discussion of this issue that goes beyond the one given in Section 3.4.3, Volume 1.

Finally, let us discuss the case in when there is no implementation and no algorithmically executable specification for a method. In this case, the method cannot be implemented automatically. However, for simulations and tests that affect this method perhaps only marginally, it is both possible and useful to generate appropriate dummy implementations.

- If the method is not relevant for the tests to be executed, an error call or the return of a default value can be generated into the method and the code is compilable.
- If the method has not been realized yet, we can use an interactive input field during simulation runs to allow users to determine the result themselves based on the current parameters.
- Results for a finite set of inputs can be stored in a table. These results may, for example, have been recorded from earlier interactive simulation runs.

[3] The executability lies primarily in the fact that the postconditions consist of a conjunction of equations in which the left-hand sides represent modifiable attributes or the result, and the right-hand sides use these attributes only in restricted form. For example, there must be no circular dependencies between attributes as this would prevent a direct sequential calculation. Each equation may also have a condition.

On the one hand, interactive input for individual methods in automated test runs is not an option; on the other hand, this approach does allow user decisions to be fed back into the system immediately while examining a running prototype. These user decisions can be logged and used later as test data, for example. This interactive form of obtaining insights is certainly limited but in some circumstances can lead to more effective communication with users.

In UML/P, it is not common to show auxiliary methods such as `get`*Attr* explicitly in class diagrams. This ensures that the model remains more compact and clear. We also do not have to use these functions explicitly in code bodies that are transformed during the generation. It is sufficient to add functionality for access to the attributes and their modification. A code generator transforms all assignments into method calls as described in the transformation rules. However, direct use of these methods should be allowed. Furthermore, under some circumstances, it is useful to anticipate the generation of such a method by specifying a manual implementation. This allows any possible optimizations to be executed or additional functionalities to be realized.

5.1.3 Associations

By default, a unidirectional association is transformed into an attribute, with the role name used as the attribute name. If the required role name is missing, an attribute name is created from the association name or the role name of the opposite class, according to the navigation rules given in Section 3.3.8, Volume 1.

Multiplicities are taken into account accordingly: "`0..1`" results in a simple attribute that can take the value `null`; "`1`" results in a simple attribute that always has a value; and an association with multiplicity "`*`" is set-valued. Set or list implementations are available dependent on additional tags such as {`ordered`}. For qualified associations, a mapping (`Map`) is provided accordingly.

Bidirectional associations are realized by attributes on both sides which are kept consistent by means of a suitable set of methods. If no navigation direction is specified, a suitable navigation direction is determined from the context and, if applicable, both directions are realized.

To ensure that bidirectional associations are actually consistent, access to the association is managed via generated methods. The form of these generated methods (i.e., the API that can be used for an association) depends on the properties and tags of the association.

For the tags {`addOnly`} and {`frozen`}, for example, corresponding functions for modification are restricted. Derived associations are handled with the same principles as derived attributes. This means that only query methods are provided and these are implemented via calculations.

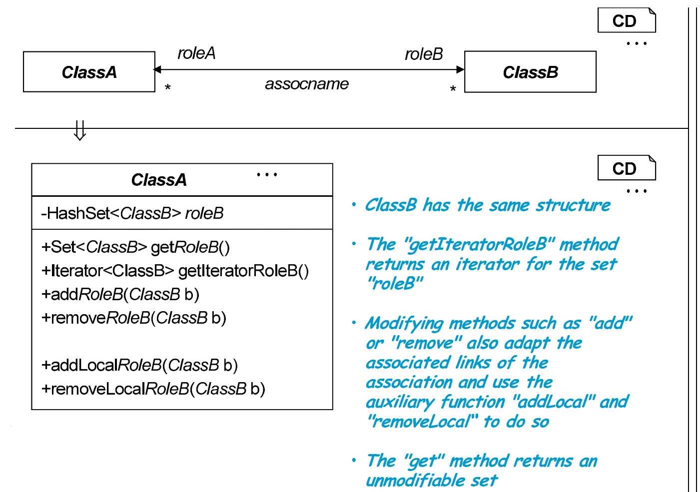
The following transformation is an example for bidirectional associations with a multiplicity "`*`" in both directions.

$Association^{*,*,bidir}$: Bidirectional association

Explanation

Associations are transformed into the state space of at least one of the classes involved. This is done by generating corresponding attributes and access functions.
This transformation rule is suitable for bidirectional associations with a multiplicity "$*$" in both directions. The association is not derived and is not a composition.

Definition of the association



- The following explanations apply for *ClassB* correspondingly, as the situation is symmetrical.
- Access to the association are modeled by access to the attribute *roleB*. This attribute has the signature of `Collection<`*ClassB*`>` introduced in Section 3.3.5, Volume 1.
- The attribute name *roleB* is extracted from the role name, the name of the association (*assocname*), or if both of these are missing, from the name of the associated class (*classB*). However, the name must be unambiguous (see Section 3.3.8, Volume 1).
- The transformation of the constructs used is relatively schematic. However, modifying operations such as `add`*RoleB* or `remove`*RoleB* are adapted accordingly to ensure the consistency of the bidirectional association.

(continued on the next page)

(continues Table 5.4.: Association^{*,*,bidir}: Bidirectional association)

Access functions	*roleB*.`isEmpty()` ⇓ *roleB*.`isEmpty()` ‖ *roleB*.`contains(`*obj*`)` ⇓ *roleB*.`contains(`*obj*`)` ‖ *roleB*.`size` ⇓ *roleB*.`size()` ‖ *roleB*.`iterator()` ⇓ `getIterator`*RoleB*`()` ‖ etc. • Read access is generally retained. The transformation corresponds to the standard transformation of the OCL `collection` interface to Java. • The iterator can also be used to delete links (analog to `remove`).
Modification	*roleB*.`add(`*obj*`)` ⇓ `add`*RoleB*`(`*obj*`)` ‖ *quali*.*roleB*.`add(`*obj*`)` ⇓ *quali*.`add`*RoleB*`(`*obj*`)` ‖ *roleB*.`remove(`*obj*`)` ⇓ `remove`*RoleB*`(`*obj*`)` ‖ *quali*.*roleB*.`remove(`*obj*`)` ⇓ *quali*.`remove`*RoleB*`(`*obj*`)` ‖ etc. • Modifying access is mapped to specially generated methods. • Further modifying access, such as *roleB*.`clear()`, is also mapped accordingly.
OCL navigation	*roleB* ⇓ `get`*RoleB*`()` ‖ *quali*.*roleB* ⇓ *quali*.`get`*RoleB*`()` ‖ • The result of `get`*RoleB* is an unmodifiable set.[4]

(continued on the next page)

[4] The method `unmodifiableSet` of the class `java.util.Collections` produces an unmodifiable set from an arbitrary set, without, however, modifying the signature.

(continues Table 5.4.: Association^{*,*,bidir}: Bidirectional association)

Additional methods	⇓ Java
	`class ClassA { ...`
	`  public synchronized Set<ClassB> getRoleB() {`
	`      return Collections.unmodifiableSet(roleB);`
	`  }`
	`  public synchronized void addRoleB(ClassB b) {`
	`      roleB.add(b);`
	`      b.addLocalRoleA(this);`
	`  }`
	`  public synchronized void`
	`                  addLocalRoleB(ClassB b) {`
	`      roleB.add(b);`
	`  }`
	`}`
	• Auxiliary functions such as `addLocal`*RoleB* or `removeLocal`*RoleB* must not be used outside the protocol even though they need to be generated as `public`. They are therefore not available to the developer.[5]
	• Further modifiers such as `remove`*RoleB* or `clear`*RoleB* are generated in a similar form. However, in bidirectional associations, `clear`*RoleB* requires, for example, the removal of all links on the opposite side, and therefore has linear complexity.
	• If we specify the tag {`addOnly`}, `remove` operations are not available.
	• If we specify the tag {`ordered`}, a list implementation is selected and the corresponding functionality is offered in addition.

Noteworthy	The consistency between two ends of a bidirectional association, which is ensured by a protocol, usually involves only a constant additional effort and is therefore acceptable. If an association is only unidirectional, this effort can, however, disappear.

Table 5.4. *Association^{*,*,bidir}*: Bidirectional association

The transformation of associations into Java code shows the extent of the variation options for code generation. It is not only the API of an association

[5] This can be achieved either with suitable context checks for the code bodies or by the use of method names not known outside the generator.

(i.e., which functions are available in UML/P for access and for manipulation) that is variably dependent on the properties of the association but also the data structure used internally. As the selection of the data structure has an effect on at least the runtime behavior of the implementation, we can use suitable control mechanisms such as the tag {`HashMap`} or a suitable adaptation of the scripts to select the best implementation.

Furthermore, for associations with limited multiplicities, we must clarify how an attempted violation of the multiplicity is handled. There are various possibilities for doing this—for example, from permitting it robustly but logging a warning, right up to throwing an exception which then has to be handled by the calling object.

In addition to the form of implementation of an association proposed above, there are proposals to realize the links with independent additional objects or to use an external data structure managed globally. The goal of all of these extensions is to offer additional functionality that is made accessible via the API to the modeler, or to optimize behavior or security properties. A globally static data structure in the form of a mapping of source object to target object is interesting, for instance, if the association is very sparse and the storage shall be efficient. This should remain hidden to the users of the API as it is a detail of the realization.

The Java code that has to ensure consistency of the association shows that it is important for the code generator to have complete control over all parts of the generated code, including method bodies. This way we can ensure, for example, the consistency condition valid for the bidirectional associations:

```
context ClassA a, ClassB b inv:
  a.roleB.contains(b) <=> b.roleA.contains(a)
```

OCL

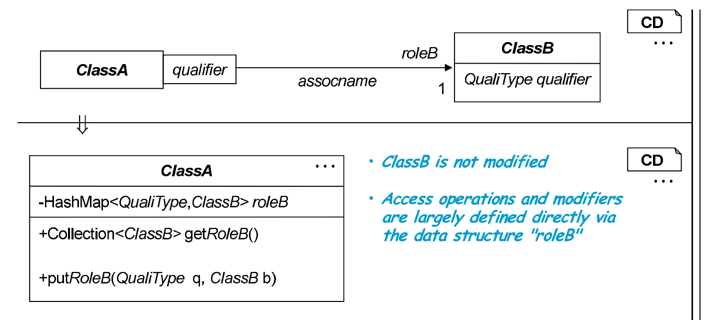
Roundtrip engineering processes cannot ensure this as they give the developer the opportunity to intervene arbitrarily in generated data structures. In that case, this consistency condition would have to be checked at runtime. If the method bodies are transformed by the code generator, the access and modification attempts for an association can be extended by consistency checks or transformations to prevent, e.g., the method `addLocal`*RoleB* being available to the developer for programming.[6]

5.1.4 Qualified Associations

Compared to the normal association, a qualified association offers an adapted API that allows qualified selection and manipulation but also prohibits some of the operations for unqualified associations. Therefore, a separate set of transformations needs to be defined for qualified associations. This list also describes the API.

[6] Normally, the generated methods should be given a different and hard to read name, only internally known, thus preventing an incidental name match. It may also be different in each generation run

<table>
<tr><th colspan="2">$Association^{quali}$: Qualified association</th></tr>
<tr><td>Explanation</td><td>A qualified association is transformed similarly to a normal association but offers some adapted and additional functionality for qualified access.
This transformation rule is suitable for unidirectional qualified associations with the multiplicity “1”.[7] The association is not derived and is not a composition.
The specified qualifier called *qualifier* is an attribute of the associated class. The value of the qualifier is therefore identical to the attribute value.</td></tr>
<tr><td>Definition of the association</td><td>



- Access functions to the association are derived from access functions to the attribute *roleB*, which has a signature of the form `Map`<*QualiType*, *ClassB*>.
- Additional methods of the unqualified associations, such as `add`*RoleB* defined below, are possible because the target object contains the qualifier.

</td></tr>
</table>

(continued on the next page)

[7] As described in Section 2.3.7, Volume 1, the multiplicity “1” means that qualified access delivers exactly one object.

(continues Table 5.5.: Association^{quali}: Qualified association)

Access functions	*roleB*`.get(`*key*`)` $\Downarrow$ *roleB*`.get(`*key*`)` *roleB*`.isEmpty` $\Downarrow$ *roleB*`.isEmpty()` *roleB*`.containsValue(`*obj*`)` $\Downarrow$ *roleB*`.containsValue(`*obj*`)` *roleB*`.keySet()` $\Downarrow$ *roleB*`.keySet()` *roleB*`.containsKey(`*obj*`)` $\Downarrow$ *roleB*`.containsKey(`*obj*`)` *roleB*`.values()` $\Downarrow$ *roleB*`.values()` *roleB*`.size` $\Downarrow$ *roleB*`.size()` • Read accesses are generally retained.
Modification	*roleB*`.clear()` $\Downarrow$ *roleB*`.clear()` *roleB*`.put(`*key*`,`*obj*`)` $\Downarrow$ `put`*RoleB*`(`*key*`,`*obj*`)` *roleB*`.removeValue(`*obj*`)` $\Downarrow$ *roleB*`.remove(`*obj*`.`*qualifier*`)` *roleB*`.removeKey(`*obj*`)` $\Downarrow$ *roleB*`.remove(`*obj*`)` *roleB*`.add(`*obj*`)` $\Downarrow$ *roleB*`.put(`*obj*`.`*qualifier*`,`*obj*`)` etc. • Further modifying accesses, such as *roleB*`.putAll`, are mapped accordingly. • The method `remove(`*obj*`)` for unqualified associations is not offered for this form of qualified association because a method for `Map`s with the same name fulfills a different functionality (it removes key values). Instead, two operations with separate names respectively are offered. • If the tag {`addOnly`} is set, the `remove` operations are not available.
OCL navigation	*roleB* $\Downarrow$ *roleB*`.values()` *roleB*`[`*key*`]` $\Downarrow$ *roleB*`.get(`*key*`)` • Qualified and unqualified navigation is possible.

(continued on the next page)

(continues Table 5.5.: Association^{quali}: Qualified association)

<table>
<tr><td>Additional methods</td><td>⇓
Java<pre>
class ClassA { ...
  public synchronized Collection<ClassB>
                                   getRoleB() {
    return Collections.unmodifiableCollection(
                                 roleB.values());
  }
  public synchronized void putRoleB
                  (QualiType q, ClassB b) {
      if(q==b.qualifier) {
                      // Object types use equals()
        roleB.put(q,b);
      } else {
                      // Exception, warning, or
                      // robust implementation
      }
  }
}
</pre>
• The method putRoleB is used to ensure that the value of the qualifier (key) and the value of the attribute qualifier are identical.</td></tr>
<tr><td>Noteworthy</td><td>Depending on the purpose of the code (test, simulation, production), we can use different strategies—from an error message through a notification in a log up to a robust implementation—to handle the error case.
The access to the attribute roleB used here must also be transformed in accordance with a valid transformation for this attribute.</td></tr>
</table>

Table 5.5. *Association*^{quali}: Qualified association

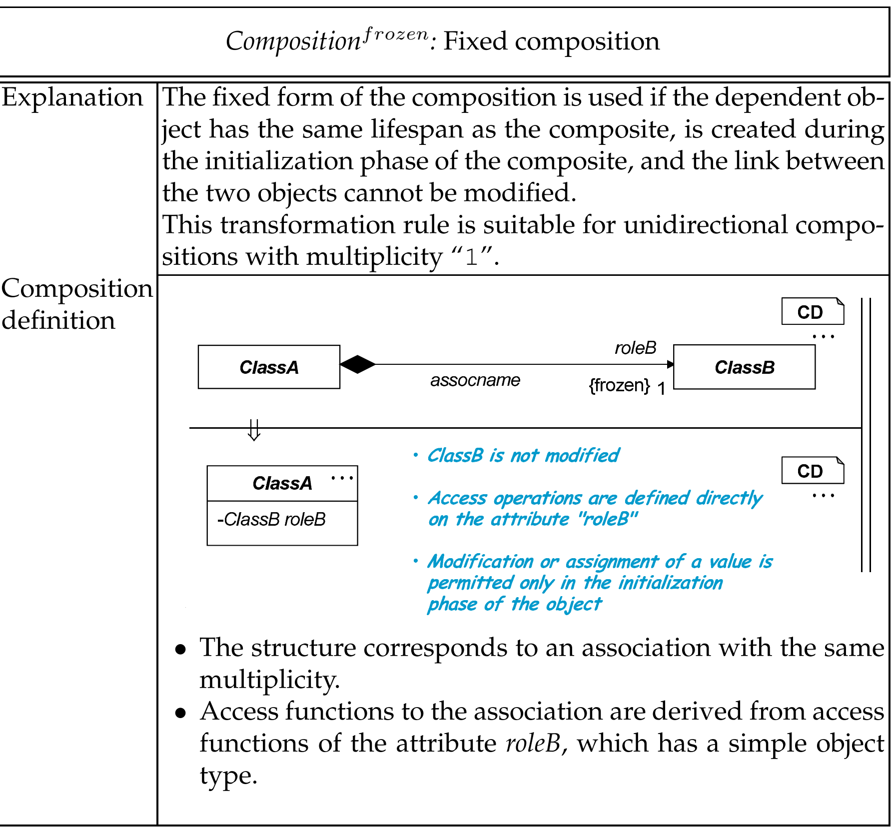
The transformation of the qualified association uses the composability of transformation rules, because here an association is initially transformed into an attribute that is encapsulated. In a further transformation step access methods are added. Please note that the method get*roleB* has to fulfill two different tasks. For qualified associations, we must differentiate between (1) the set of all objects reachable via the links, and (2) the contents of the attribute. The two meaning variants are only identical for normal associations. The method get*roleB* realizes variant (1). For variant (2), a method with the name get*roleB*Attribute is introduced where necessary. Here it returns a Map object. However, thanks to the numerous qualified access options, there

exists no necessity for the modeler to access the realizing `Map` data structure directly.

5.1.5 Compositions

As discussed in Section 2.3.4, Volume 1, there is a correlation between the life cycles of a composite and its dependent objects. However, the correlation can have significant differences in its interpretation. From a structural perspective, a composition is treated like a normal association; however, the creation or removal of links from a composition is subject to the respective interpretation. Accordingly, some operations of the association API are not offered or are subject to restrictions.

One interpretation of a composite, which is relatively widespread, is presented below.



Composition^{frozen}: Fixed composition	
Explanation	The fixed form of the composition is used if the dependent object has the same lifespan as the composite, is created during the initialization phase of the composite, and the link between the two objects cannot be modified. This transformation rule is suitable for unidirectional compositions with multiplicity "`1`".
Composition definition	[diagram] • The structure corresponds to an association with the same multiplicity. • Access functions to the association are derived from access functions of the attribute *roleB*, which has a simple object type.

(continued on the next page)

(continues Table 5.6.: Composition^{frozen}: Fixed composition)

	• The attribute name *roleB* is extracted from the role name, the name of the association (*assocname*), or if both of these are missing (which is often the case for compositions), from the name of the associated class (*classB*). However, there must be a guarantee that the name is unambiguous (see Section 3.3.8, Volume 1).
Access function	*roleB* ⇓ *roleB* • Read accesses to the attribute are transformed using a downstream transformation rule (usually in `get`*RoleB*`()`).
Modification	A value may only be assigned to the attribute *roleB* in the constructor, i.e., during the initialization phase.[8] To achieve this, we use either a factory or a `new` command: *roleB*=*factory*`.new`*ClassB*`(`*arguments*`)` *roleB*=`new` *ClassB*`(`*arguments*`)` Static analysis ensures that the unique assignment of a value to the attribute always takes place. • In a bidirectional composition, we also set the corresponding link in the dependent object using a method described in the transformation *association*^{*,*,bidir}. To do this, we replace the assignment *roleB*= shown above with a method call `set`*RoleB*.
OCL navigation	Similar to preceding transformation rules
Noteworthy	The restriction that the dependent object is only created in the constructor of the composite ensures that dependent objects are not used more than once.[9] A less strict transformation would allow, for example, that dependent objects are given to the constructor as parameters. However, if that were the case, it would no longer be possible to determine whether the object was created as new and thus definitely satisfies the composition relationship.

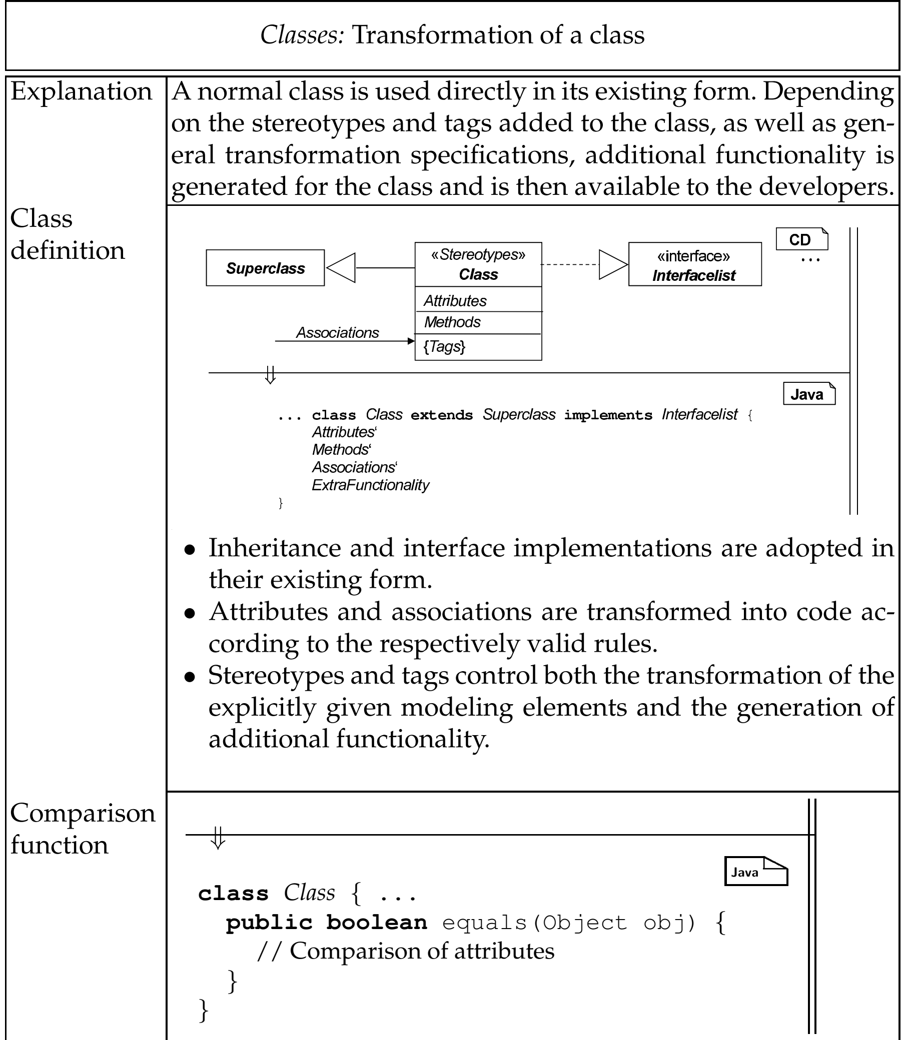
Table 5.6. *Composition*^{frozen}: Fixed composition

[8] For certain classes, for example Applets, the initialization is outsourced to a method `init()` which then counts for the initialization phase.

[9] The assumption is that a factory actually creates new objects.

5.1.6 Classes

The transformation of a class, along with its attributes, methods, associations, compositions, and the later discussed inheritance relationships, is relatively schematic as the canonical approach is to map the UML class to the Java class directly. However, the transformation of classes is strongly driven by stereotypes and tags. They control which additional functionality and which variants of the attribute transformation are executed. No stereotypes are taken into account in this core transformation.



Classes: Transformation of a class	
Explanation	A normal class is used directly in its existing form. Depending on the stereotypes and tags added to the class, as well as general transformation specifications, additional functionality is generated for the class and is then available to the developers.
Class definition	(diagram) • Inheritance and interface implementations are adopted in their existing form. • Attributes and associations are transformed into code according to the respectively valid rules. • Stereotypes and tags control both the transformation of the explicitly given modeling elements and the generation of additional functionality.

Comparison function	`class Class { ...`
	`  public boolean equals(Object obj) {`
	`    // Comparison of attributes`
	`  }`
	`}`

(continued on the next page)

(continues Table 5.7.: Classes: Transformation of a class)

	• The method `equals` compares the newly defined attributes and uses the method of the superclass with the same name. • Associations and derived attributes are not usually used in a comparison whereas compositions, in which the class represents the composite, are included. • The tag {`Equals=`*list*} allows an explicit listing of the attributes and associations included in the comparison. For an abbreviated form, we can use the tag `Equals+` to specify a list of additional associations or the tag `Equals-` to specify a list of attributes to be removed. • If we have already specified an `equals` method explicitly for the class, this is adopted in its existing form and does not have to be generated.

Hash function	⇓ Java
	`class Class { ...`
	`  public int hashCode() {`
	`    // Appropriate calculation from the attributes`
	`  }`
	`}`
	• The hash function is implemented appropriately.
	• We can use the tags {`Hash=`*List*}, {`Hash+`}, and {`Hash-`} analog to the comparison function to control which attributes are used.
	• If we have already specified a `hash` method explicitly for the class, then this is adopted in its existing form.

String conversion	⇓ Java
	`class Class { ...`
	`  public String toString() {`
	`    // for attributes and associations`
	`  }`
	`}`
	• The method `toString` delivers a simple transformation into a string that contains the involved attributes. This form of output is used primarily for tests and simulations and should not normally be used in the product system.
	• Associations stored in the state space of the class, derived attributes, and compositions can also be included.

(continued on the next page)

(continues Table 5.7.: Classes: Transformation of a class)

	• We can use the tags {ToString=*list*}, {ToString+}, and {ToString-} to control which class elements are considered as output. • We can use the tag {ToStringVerbosity=*number*} to control the verbosity—0: no output; 1: class name; 2: attribute contents very compact (with no reachable and dependent objects); and 6: verbose output of every attribute and every association in the form *attribute name*=*attribute value*, which encompasses all reachable objects. • If a toString method is already specified explicitly for the class, then this is kept in its existing form.

Constructors	⇓ Java
	`class Class { ...`
	`  public Class() {`
	`    // Assignment of default values to the attributes`
	`  }`
	`  public Class(AttributeList) {`
	`    setAttribute(attribute); ...`
	`  }`
	`}`
	• Constructors are created according to the generation strategy.
	• Unless explicitly excluded, as standard, the empty constructor and a constructor for assigning values to all attributes are included.
	• Because we can use the tag {new(*AttributeList*)} multiple times, we can create any number of constructors. Alternatively, we can also specify constructors directly, as this allows us to realize additional functionality in the constructor.

Log output	⇓ Java
	`class Class { ...`
	`  public String stringForProtocol() {`
	`    // Writes attributes and some of the associations`
	`  }`
	`}`

(continued on the next page)

(continues Table 5.7.: Classes: Transformation of a class)

	• This method works similarly to `toString` but is used for output in logs. We can parameterize it in the same way or implement it manually.
Noteworthy	In addition to `stringForProtocol`, there are a number of other functions that are realized, but that have not been mentioned here. Some are derived from the class `Object` (e.g., `clone`), others are from interfaces that have to be implemented (e.g., `compareTo` from the interface `Comparable`), and others are the result of implementation specifications for the code generator. These include functionalities for log output as described above, storage, error handling, or additional functions useful for processing tests.

Table 5.7. *Classes:* Transformation of a class

In particular, when we use a class in test environments, under some circumstances a number of other methods and data structures have to be generated for this class. A code generator can provide valuable assistance for generating such uniform methods for implementation and tests.

However, let us briefly consider one of the few and very rarely selected alternatives to the mapping described here: instead of using the type system of the target language Java, this alternative approach stores attributes as a mapping of the attribute name to the value with the `HashMap<String,Object>`. In principle, it is then sufficient to implement a single Java class in the form represented in Fig. 5.8—this form offers a certain amount of additional flexibility although it is also less efficient and more error prone to wrong modifications. A similar form is used, for instance, to manage parameters.

5.1.7 Object Instantiation

A final interesting point in the context of code generation for classes is the management of their objects. This includes, for instance, the creation of objects, management of and efficient access to individual objects, or the storage in and loading from databases. Management activities are often implemented in "management objects" which, in addition to collecting the objects located in the memory, allow a transaction-controlled mapping to the database and an efficient access to loaded objects. Of all of these activities, below we will look only at the object instantiation in Java, as it must, e.g., be instrumentable for tests.

The form of `new Class(...)` used in the code bodies can be transformed during code creation by calling suitable factory methods. This increases the flexibility during the code creation considerably, as it allows us to

```java
class Chameleon { ...
  // Carrier of all attributes
  HashMap<String,Object> attributes;

  public Object get(String attributeName) {
      return attributes.get(attributeName);
  }

  // Type check: determine whether certain attributes are present
  public boolean isInstanceOf(Set<String> attributeNames) {
      return attributes.keySet().containsAll(attributeNames);
  }
}
```

Fig. 5.8. Dynamic management of attributes

use subclasses or insert mocks in automated tests.[10] This approach is based on the design pattern *Abstract Factory* from [GHJV94].

Object instantiation: Creating objects with a factory	
Explanation	In the source code, we create the objects with the `new` construct. In contrast, the generated code contains factory calls. A standard factory is generated and we can adapt it to specific situations by creating subclasses.
Object instantiation	`new` *Class*(*Arguments*) ⇓ `Factory.new`*Class*(*Arguments*) • The generated class `Factory` has a static method `new`*Class* that creates the new object. • Attribute `f` holds a standard value after the system initialization but may be overwritten using a subclass. • Redefining `create`*Class* allows us to create objects from subclasses and singletons, to manage object sets, and much more.

(continued on the next page)

[10] Mocks (also called dummies) simulate an object without actually implementing the functionality. This is just as suitable for simulating an interaction with the system environment as it is for component interfaces.

(continues Table 5.9.: Object instantiation: Creating objects with a factory)

<table>
<tr><td>Class
Factory</td><td>

```java
public class Factory { ...
  public static initFactory() {
    f = new Factory();
  }

  // For each class Class
  public static Class newClass(Arguments) {
    return f.createClass(Arguments);
  }

  // Allows the static methods above to be redefined
  protected static Factory f;

  protected Class createClass(Arguments) {
    return new Class(Arguments);
  }
}
```

Java

- Corresponding factory methods are generated for each class of the system.
- Multiple factory methods with different parameter sets are created for the same class if there are corresponding constructors.
- We can subdivide the factory into multiple classes, for example, in accordance with a subsystem structure, but this must then be controlled by the generator script.
</td></tr>
<tr><td>Alternative</td><td>Instead of a single attribute f, we can use a separate attribute for multiple groups of classes to be created or even for each class so that we can adapt the generation of objects individually:</td></tr>
</table>

(continued on the next page)

(continues Table 5.9.: Object instantiation: Creating objects with a factory)

Java

```
public class Factory { ...
  public static initFactory() {
    fClass = new Factory(); ... // For each class
  }

  // For each class Class
  public static Class newClass(Arguments) {
    return fClass.createClass(Arguments);
  }

  protected static Factory fClass; ... // For each class

  protected Class createClass(Arguments) {
    return new Class(Arguments);
  }
}
```

Whereby calls are transformed again as follows:

```
new Class(Arguments)
⇓
Factory.newClass(Arguments)
```

Table 5.9. *Object instantiation:* Creating objects with a factory

Multilevel Transformation

The variants for transforming class diagrams discussed above using examples show the wide variety of possible forms of generation. As already discussed, this means that the code generation requires a high level of flexibility to fulfill the required tasks. One way of increasing the flexibility is the option of selecting from multiple templates or scripts. Furthermore, the templates use each other: for example, associations are first transformed into attributes and these are then encapsulated by access methods. The rules for transforming concepts of class diagrams into Java code shown in this section are therefore not independent of one another. Fig. 5.10 shows the dependencies of the transformation rules.

Only the explicitly defined rules are described here. However, to enable a suitable infrastructure, there should be further transformation rules defined by further templates. The selection of the respective alternatives is sometimes stipulated by the context or the properties of the transformed language concept (e.g., as for the associations here) or can be controlled via generator settings (e.g., as for the derived attributes here).

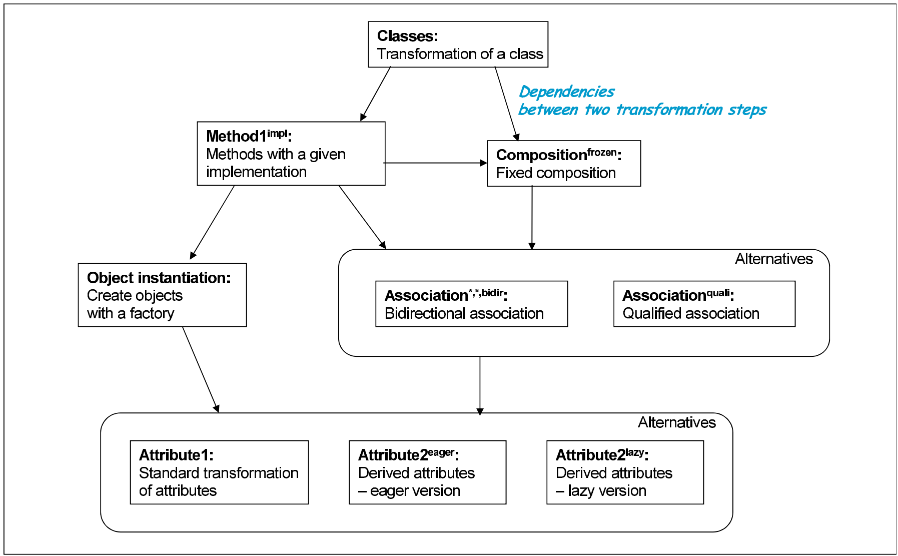


Fig. 5.10. Dependencies of the transformation rules discussed for generating code from class diagrams

5.2 Transformations for Object Diagrams

Describing the generation of code from UML/P completely is beyond the scope of this book. Therefore, the following sections explain some of the interesting aspects of the transformation of further types of models into Java code without discussing all of the details.

We can use object diagrams in two ways. On the one hand, we can use them in a constructive form to create object structures conforming to the object diagram. We can use this functionality both in the product system and to represent object structures that have to be used for automated tests. On the other hand, we can use object diagrams as predicates to check whether a certain object structure is present. Section 4.4, Volume 1 looks at these types of use from a methodological perspective. Therefore, this section discusses aspects of code generation from object diagrams. The possible integration of object diagrams and OCL discussed in Chapter 4, Volume 1 is thereby ignored.

5.2.1 Object Diagrams Used For Constructive Code

The transformation of an object diagram into functionality that constructively creates the objects as described in the diagram can be controlled by a script or by tags that we add to the object diagram. For this form of code generation, some parameters can be identified:

1. The class that contains the code for creating the object structure. If there is a principal object identifiable in the diagram, we may by default take the class of this object.
2. The name of the method to be created: If this name is unique, a suitable default is `setup`*DiagramName*.
3. The objects of the diagram, which already exist and are given to that method as parameters: this is usually none (the object structure is completely created), the principal object (from which the rest is created) or some objects that have already been created earlier (and only the rest needs to be newly created).
4. If the object diagram contains free variables, they are also interpreted as parameters for the generated method that need to be filled on execution.

As the situations in which the already existing objects are can vary, it can be useful to generate multiple methods from one object diagram. We can differentiate the methods using the signature or, if this is not unique, the method name.

The use of free variables and attributes with no value assigned as parameters of the generated function allows object diagrams to be interpreted as *patterns* with *prototypical* objects that allow multiple instantiations with different contents, as discussed in Section 4.2.2, Volume 1.

There are several aspects that we have to consider when generating these `setup` methods. To create an object we need a constructor. It is relatively simple, if a constructor without parameters is available, creating only the empty object. If no such constructor is available, it can be generated, potentially only for the test system. In the product system, however, we have to use an existing constructor that has been labeled accordingly.

Furthermore, attributes need values to be the assigned to like described in the object diagram. For this purpose, suitable auxiliary functions should be available or the attributes should be accessed directly. The `set` methods discussed in Section 5.1.1 are only partially suitable here because they might contain additional functionality.

In principle, constructors with parameters can also be used if the constructor contains only assignments of the parameters to attributes.[11]

An object diagram can generally be incomplete it the class of an object is not specified or some attributes have no value. On the one hand, attributes with no value assignment can be understood as free variables and can be included in the generated method as parameters. If this is not desired, depending on the type of use, we can identify different strategies for handling attributes with no value assigned in the test system, for simulation, or in the product system. In the test system, we use a specific failure strategy: unassigned attributes should be irrelevant for the system run under test and access to such an attribute should result in an immediate test failure. We can

[11] This is the case, for example, if the constructor was also generated, as described in Section 5.1.

integrate corresponding behavior in the `get` functions. For simulation, the approach discussed in Section 5.1.2 is useful. In this approach, missing attribute values are queried interactively during the simulation run or default values are used as an alternative. Finally, to generate code for the product system, a complete definition of the objects in the object diagram is a prerequisite. This prevents carelessness in the definition of object diagrams and thus gives the developer confidence in the reliability of the modeled system.

Not all details that can be formulated in an object diagram are used for constructive or predicative (this will be discussed later on) code generation. Visibilities, the information about the compositionality of a link, tags such as {`frozen`}, and so on do not require transformation into the code and are not discussed here. Instead they are compared with the information in class diagrams or are used in tests.

An alternative approach for using object diagrams constructively is discussed in Section 4.4.7, Volume 1. This approach is associated with method specifications. It uses an object diagram in the postcondition of a constructor or an initialization method that can be transformed into constructive code according to the same principles as those described here.

5.2.2 Example of a Constructive Code Generation

Instead of discussing the transformation rules for each model element of the object diagram, we will discuss them using the object diagram shown in Fig. 5.11. This diagram describes an excerpt of the initial object structure of the applet in the auction system and is embedded in an OCL method specification for the initialization function `init()`. Fig. 5.12 shows the generated code, whereby here and in the following examples, we assume for the sake of simplicity that the direct access to the attributes still has to be transformed.

From the related class diagram, which is not shown here, the system can derive that the links are realized by the attributes `loginPanel` and `httpServerProxy` in the class `WebBidding`. For the classes `HttpServerProxy` and `LoginPanel`, a suitably parameterized constructor is used and assigns values to the specified attributes.[12] The links are assigned by `set` methods that also ensure the correct setting of the return direction.

5.2.3 Object Diagram Used as Predicate

An object diagram can be mapped to a Boolean predicate that checks whether a set of objects conforms to the object diagram and that the values specified in the diagram match to the attributes in the objects. Such a method for example helps to understand, whether a good (or bad) situation is achieved, an assertion holds, etc. Similarly to the discussion above, multiple parameters control the code to be created:

[12] If no suitable constructor exists, a code generator can generate one.

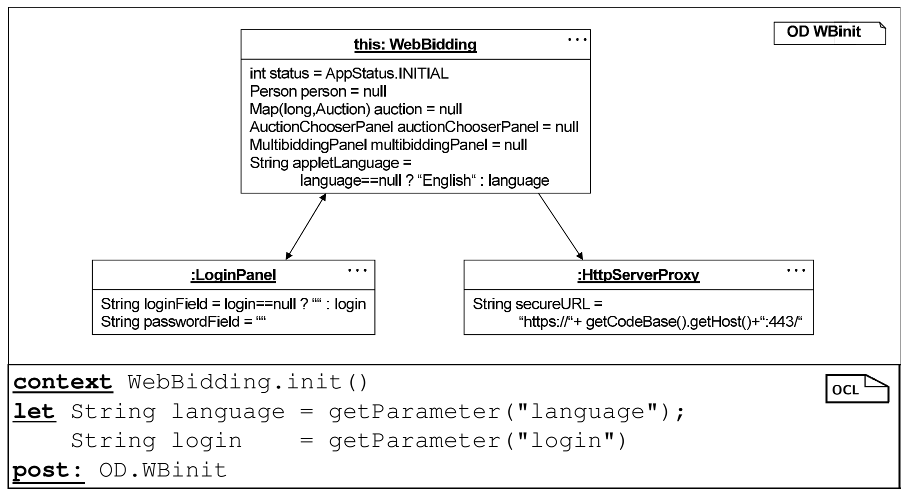


Fig. 5.11. Object diagram for initializing a structure

```
class WebBidding { ...                                    Java
  public void init() {
    // From the let construct
    String language = getParameter("language");
    String login    = getParameter("login");
    // From the object this
    status = AppStatus.INITIAL;
    person = null;
    auction = null;
    auctionChooserPanel = null;
    multibiddingPanel = null;
    appletLanguage = language==null ? "English" : language;

    // From the object :LoginPanel
    setLoginPanel(new LoginPanel(login==null ? "":login,""));

    // From the object :HttpServerProxy
    setHttpServerProxy(
      new HttpServerProxy(
          "https://"+getCodeBase().getHost()+":443/"));
  }
}
```

Fig. 5.12. `init()` function generated from the object diagram

1. The class that contains the Boolean method may be the class of a uniquely identifiable principal object in the diagram, for example by the name

`this:Classname`. Alternatively, the method can be static and/or defined within a test class.

2. The name of the method to be created: if the name is not given, then `isStructuredAs`*DiagramName* is used per default.
3. If some objects in the system have already been identified, these objects become parameters of the method. Usually there is a single object that acts as a kind of master for the object structure and potentially the method is part of this object's class.
4. If the object diagram contains free variables, these are usually not considered further. However, if these variables are to take certain values, they also become parameters for the generated method in order to describe desired values that then can be matched against the real object attribute values.

In contrast to the constructive variant of an object diagram, a diagram used predicatively can be incomplete in several respects. Attributes and attribute values can be just as equally omitted as the classes of the objects. Properties with a temporal implication, such as the tag {`frozen`} for links, cannot be checked in a predicate via a state either. To check these properties in a predicate, additional infrastructure is required that either ensures this property constructively, by not offering any methods for modifying a link, or performs a check at runtime.

Section 4.3, Volume 1 discussed the meaning of an object diagram as a predicate in the context of the integration with OCL constraints in detail. There we established that we can generally represent an object diagram as an OCL constraint. Those predicative object diagrams are transformed into corresponding Boolean methods labeled with the stereotype ≪query≫ introduced in Section 3.4.1, Volume 1. These generated methods can be used in Statecharts and also in Java bodies of the product code.

When the methods are used in product code, however, we have to take the efficiency of the generated code into account. As discussed in Section 4.3, Volume 1, anonymous objects of the object diagram are regarded as existentially quantified. Named objects that are not parameters of the Boolean predicate are handled in the same way. The predicate searches for these objects itself by checking the corresponding associations. Values are assigned to the free objects in a form analog to the structure matching of graph grammars [Roz99, EEKR99].

For set-valued associations, this type of search can be complex and should be avoided. Options for improving the situation include using a qualifier for the association or explicitly transferring intended objects as parameters if they can be determined efficiently from the context.

The transformation into a predicate is illustrated using the object diagram presented in Fig. 5.13 which comes from the auction system. The code generated for the use in a test system is represented in Fig. 5.14.

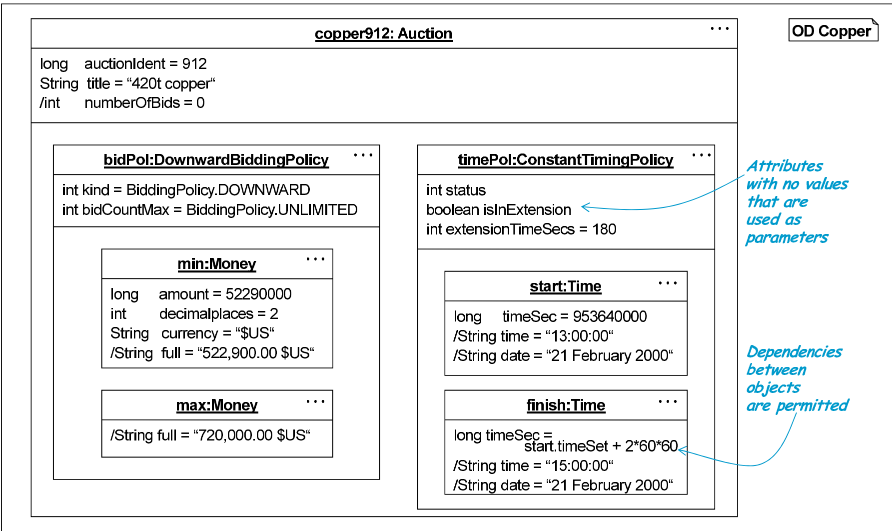


Fig. 5.13. Object diagram determined for a predicate

After a phase of assigning real objects to the diagram names all attributes are checked. Expressions for attribute values may use other attributes as long as usage is acyclic. If real subclasses are specified instead of the types specified in the class diagram (not shown here), the generated code checks whether the corresponding object actually belongs to this subclass.

From the length of the code shown in Fig. 5.14, it is obvious that a representation in an object diagram is more compact and clearer, giving the modeler an advantage, particularly for gaining a quick overview and when searching for individual values.

A further method called `isExactlyStructuredAs`*DiagramName* can be generated during the code generation as an extension or alternative. This method can also ensure that the object structure contains only the objects specified in the diagram. This is particularly interesting for multivalued and optional associations as well as for object diagrams, which can be controlled with a suitable stereotype ≪complete≫. Table 5.15 gives a brief introduction to this stereotype.

Stereotype ≪complete≫	
Model element	Object diagram
Motivation	The meaning of an object diagram as a predicate is normally defined such that the properties specified explicitly must hold. Further objects not specified in the diagram can exist.

(continued on the next page)

(continues Table 5.15.: Stereotype ≪complete≫)

Usage condition	In an object diagram labeled ≪complete≫, all attributes and links must be specified. Ordered associations must be represented completely.
Effect	The stereotype ≪complete≫ requires that no further objects exist in the specified object structure. The object diagram specified is therefore a complete representation of the object structure. Thus method `isExactlyStructuredAs`*DiagramName* is to be generated; it checks the completeness of the object diagram and whether these properties are satisfied.

Table 5.15. Stereotype ≪complete≫

5.2.4 An Object Diagram Describes a Structure Modification

Another interesting form of use of object diagrams results from the combination of both forms of use: an existing object structure is checked for the presence of the objects described in the object diagram, the missing objects are generated, and the attributes with wrong content are modified. These types of methods have the name `adaptTo`*DiagramName*. We can use them to restructure an existing object structure dependent on the state currently desired. We can therefore use object diagrams as state invariants in a Statechart or to adapt the respective object structure in the entry action of states. If the object diagram shown in Fig. 5.11 is used in this form (without being embedded in the OCL constraint that exists there), the method represented in Fig. 5.16 is created.

If an object is missing or has the incorrect type, it is created. If the object already exists, its attributes are modified in the desired form. Therefore, both constructors, which already contain attribute values as parameters in this example, as well as `set` methods are used to assign values to attributes.

Objects to be identified along a link, such as the anonymous object `:LoginPanel`, are potentially ambiguous for associations with multiplicity "`*`" or "`1..*`". For set-valued associations, a comparison with all existing objects must be performed and must include the attributes of the association. If the object structure does not contain any object that corresponds to the prototypical object specified in the diagram, none of the existing objects are adapted and a new object is created instead. The size of the set of links increases accordingly. The disadvantage of this method, however, is that we cannot represent the deletion of undesired objects. This method can also be inefficient if, for example, one hundred persons are created individually with the object diagram (without the OCL constraint) given in Fig. 3.28 and the method generated from this object diagram and represented in Fig. 5.17.

```
Java
class Auction { ...
  public static boolean isStructedAsCopper(Auction copper912,
                        int status, boolean isInExtension) {
    // Define the name (can be optimized)
    BiddingPolicy bidPol = copper912.bidPol;
    TimingPolicy timePol = copper912.timePol;
    Money min = bidPol.min;
    Money max = bidPol.max;
    Time start = timePol.start;
    Time finish = timePol.finish;

    return
     // Main object
     copper912.auctionIdent == 912 &&
     copper912.title.equals("420t copper") &&
     copper912.numberOfBids == 0 &&

     // BiddingPolicy
     bidPol instanceof DownwardBiddingPolicy &&
     bidPol.kind == BiddingPolicy.DOWNWARD &&
     bidPol.bidCountMax == BiddingPolicy.UNLIMITED &&

     // Money objects
     min.amount == 52290000 &&
     min.decimalplaces == 2 &&
     min.currency.equals("$US") &&
     min.full.equals("522,900.00 $US") &&
     max.full.equals("720,000.00 $US") &&

     // TimingPolicy
     timePol instanceof ConstantTimingPolicy &&
     timePol.status == status &&
     timePol.isInExtension == isInExtension &&
     timePol.extensionTimeSecs == 180 &&

     // Times
     start.timeSec == 953640000 &&
     start.time.equals("13:00:00") &&
     start.date.equals("21 February 2000") &&
     finish.timeSec == start.timeSec + 2*60*60 &&
     finish.time.equals("15:00:00") &&
     finish.date.equals("21 February 2000") ;
  }
}
```

Fig. 5.14. Predicate generated from an object diagram

```java
class WebBidding {

  public void adaptToWBinit(String language, String login) {
    // From the object this
    status = AppStatus.INITIAL;
    person = null;
    auction = null;
    auctionChooserPanel = null;
    multibiddingPanel = null;
    appletLanguage = language==null ? "English" : language;

    // From the object :LoginPanel
    if(loginPanel != null &&
                loginPanel instanceof LoginPanel) {
      loginPanel.loginField = (login==null) ? "" : login;
      loginPanel.passwordField = "";
    } else {
      setLoginPanel(new LoginPanel(
                login==null ? "" : login, ""));
    }

    // From the object :HttpServerProxy
    if(httpServerProxy != null &&
                httpServerProxy instanceof HttpServerProxy) {
      httpServerProxy.secureURL =
                "https://"+getCodeBase().getHost()+":443/";
      httpServerProxy.connectStatus =
                HttpServerProxy.NOT_CONNECTED;
      httpServerProxy.lastConnectionTime = Time.now();
    } else {
      setHttpServerProxy(
        new HttpServerProxy(
                "https://"+getCodeBase().getHost()+":443/"));
    }
}}
```

Fig. 5.16. Adaptation method generated from an object diagram

5.2.5 Object Diagrams and OCL

An important mechanism for increasing the expressiveness of object diagrams is the integration with OCL from Section 4.3, Volume 1. The transformation of an object diagram extended with OCL constraints into constructive code or into a predicate depends on the extent to which the OCL constraints can be transformed. The transformation and the executability of OCL constraints were discussed in Section 4.1.2.

```
class Auction { ...                                              Java/P

  public void adaptToNPersons(Auction test32, int x) {
    // Set object test32
    test32.auctionIdent = 32;
    test32.title = "TestAuction";

    // Is p:Person present?
    Person p = null;
    for(Iterator<Person> ip = participants.iterator();
                                  ip.hasNext() && p==null; ) {
      Person pit = ip.next();
      if(pit.personIdent == 1000+x &&
         pit.login == "log" +x &&
         pit.name == "Tester " +x &&
         pit.isActive == (x%2 == 0)) {
             p = pit;
      }
    }
    // Create p:Person
    if(p == null) {
      p = new Person(); // Default constructor
      p.personIdent = 1000+x;
      p.login = "log" +x;
      p.name = "Tester " +x;
      p.isActive = (x    }
  }
}
```

Fig. 5.17. Adaptation method with search in $\star$ association

Note, however, that the linear search complexity for existentially quantified objects discussed previously can increase polynomially if navigation via a chain of set-valued associations occurs and the reachable objects are linked via an OCL constraint.

5.3 Code Generation from OCL

Based on the options described so far for using OCL as the specification language for UML/P programs, we can use OCL, for instance, to model invariants in database systems [DH99] or to describe the business logic [DHL01]. In essence, the underlying semantics of OCL is identical to the previously discussed meaning of OCL, but the form of use and the mapping of OCL to executable code is significantly different. Instead of classes and objects, relational databases offer *entities* and *rows* that are used to interpret OCL expressions.

Several, partially prototypical tools offer realizations for OCL code generators. [HDF00] presents a modular architecture for OCL that is suitable as the basis for an integration with other UML tools [BS01a, BBWL01]. A language related to OCL, “Java Interface Specification Language” (JISL), is presented in [MMPH99] for method specification, with a discussion of how the specification can be executed by means of a transformation to Java.

As there are a number of concepts that can be uniquely transformed to Java from OCL/P due to the syntactic and semantic similarity between the two languages, we will discuss primarily the interesting OCL constructs below. These include the transformation of the following: context definition, OCL logic, comprehension, navigation, quantifiers, and some special operators.

Many but not all of the pieces of code described below can be encapsulated in reusable methods. The encapsulation is not discussed here, as it merely increases the size (number of lines) of code. However, if these pieces of code are used manually (as a type of design pattern), encapsulation is recommended.

5.3.1 An OCL Expression as a Predicate

Even though, as described in Section 4.1.2, OCL constraints can be transformed into implementation code in a very restricted form, because an OCL expression has a natural meaning as a predicate. Therefore, OCL expressions such as the following are transformed into boolean methods:

```
context Auction a inv Bidders1:                                    OCL
  a.activeParticipants <= a.bidder.size
```

There are two variants. The interpretation as an invariant produces the following:

```
public static boolean invariantBidders1() {                     Java/P
  boolean res = true;
  Set<Auction> auctions = Auction.getAllInstances();
  for(Iterator<Auction> ia = auctions.iterator();
                             ia.hasNext() && res; ) {
   Auction a = ia.next();
   res &= a.activeParticipants <= a.bidder.size;
  }
  return res;
}
```

The generated method checks the invariant `Bidders1` for all objects of the class `Auction`. This requires an infrastructure for managing the set of all auction objects and this is, for instance, provided by the method `getAllInstances`.

Using the OCL constraint in the form described above is not actually suitable for practical applications. In a live system, relatively few auction objects

will usually have been modified since the last check of the invariant. In some circumstances it is worth investing in additional infrastructure for an efficient invariant check. This can be achieved by requiring the invariant check explicitly at certain points. In that case, only certain auction objects would have to be tested rather than all of them. The method suitable for this is:

```
public static boolean checkBidders1(Auction a) {
  return a.activeParticipants <= a.bidder.size();
}
```

Java

In particular, the variant in encoded class `Auction` is relevant:

```
class Auction ... {
  public boolean checkBidders1() {
    return this.activeParticipants <= this.bidder.size();
  }
```

Java

All of these forms are suitable for use in tests in which the invariant check can be explicitly required. Note that, according to convention, the complete invariant has the prefix `invariant`, while `check` is used for checking individual cases. The explicit transfer of the context of an OCL expression in the form of the objects to be checked is normally executed by means of the keyword `import`. Therefore, the last two methods are also created from the following condition:

```
import Auction a inv Bidders1:
  a.activeParticipants <= a.bidder.size
```

OCL

As, in practice, a closed context with the keyword **context** often has to be applied to individual objects as well, a closed method is created from it as well as additional parameterized methods. If the context specification consists of multiple objects, they are all used as parameters. Table 5.18 below describes the transformation:

OCL context: Transformation of the context of a condition	
Explanation	In OCL, the context is specified explicitly in the form of objects. These objects act as parameters for the Boolean methods generated from the context.

(continued on the next page)

(continues Table 5.18.: OCL context: Transformation of the context of a condition)

<table>
<tr><td>Context</td><td>

```
context Classcontext inv Name:                    OCL
  Body
⇓
                                                  Java/P

public static boolean invariantName() {
  boolean res = true;
  // Iteration represented only once
  Set<Class> s = Class.getAllInstances();
  for(Iterator<Class> it = s.iterator();
                      it.hasNext() && res; ) {
   Class ob = it.next();
   res &= Body';
  }
  return res;
}
public static boolean checkName(Classcontext) {
  Body'
}
```


The iteration is repeated nested for each element Class ob of the context. This results accordingly in polynomial complexities during the check of the complete invariant.
The generated code is stored in a static method within a suitable class. If the context consists of only one object, a nonstatic method can also be generated in the class of the object.
The method <code>getAllInstances</code> is described below.
The alternative keyword <code>import</code> generates only the parameterized forms.
The body of the condition Body is transformed into Body′ using further rules.

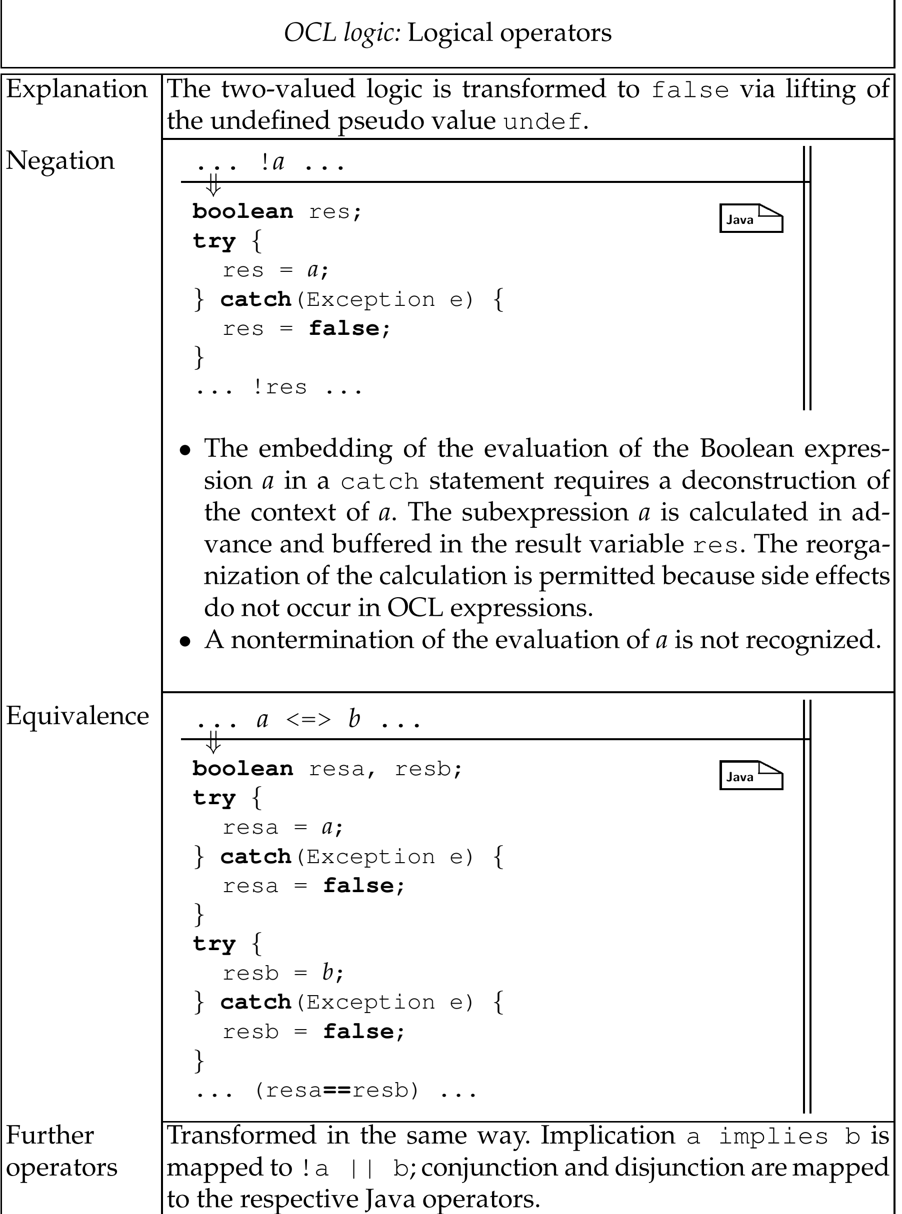
</td></tr>
</table>

Table 5.18. *OCL context:* Transformation of the context of a condition

5.3.2 OCL Logic

As discussed in Section 3.2.2, Volume 1, the OCL logic is a two-valued logic and uses an implicit *Lifting operator* to interpret undefined results as `false`. Due to its property ↑`undef==false`, the lifting operator, denoted with ↑, cannot be implemented completely. However, there is a transformation of this operator to Java, discussed in Section 3.2.2, Volume 1, which catches exceptions and thus can recognize only terminating calculations. The lifting operator can be found implicitly in all Boolean operations of OCL, meaning

that the transformations of the Boolean operations `&&`, `||`, `!`, `implies`, and `<=>` each require the use of the lifting.



OCL logic: Logical operators

Explanation

The two-valued logic is transformed to `false` via lifting of the undefined pseudo value `undef`.

Negation

```
... !a ...
⇓
boolean res;
try {
  res = a;
} catch(Exception e) {
  res = false;
}
... !res ...
```


- The embedding of the evaluation of the Boolean expression a in a `catch` statement requires a deconstruction of the context of a. The subexpression a is calculated in advance and buffered in the result variable `res`. The reorganization of the calculation is permitted because side effects do not occur in OCL expressions.
- A nontermination of the evaluation of a is not recognized.

Equivalence

```
... a <=> b ...
⇓
boolean resa, resb;
try {
  resa = a;
} catch(Exception e) {
  resa = false;
}
try {
  resb = b;
} catch(Exception e) {
  resb = false;
}
... (resa==resb) ...
```


Further operators

Transformed in the same way. Implication `a implies b` is mapped to `!a || b`; conjunction and disjunction are mapped to the respective Java operators.

Table 5.19. *OCL logic:* Logical operators

The corresponding Java control statements can be used to transform the OCL control structures. The **let** construct introduces local variables and catches any exceptions that occur during the calculation of the variable values.

5.3.3 OCL Types

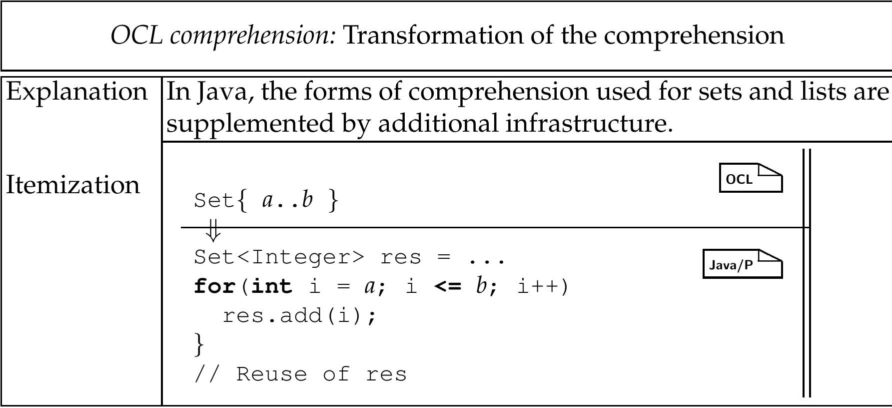
The OCL/P primitive data types match those from Java, which means that a transformation is not necessary. The comparison of the primitive data types == and the arithmetic operations are also adopted without modification.

The containers `Set<X>`, `List<X>`, and `Collection<X>`, which OCL offers, represent a subset of the containers from Java. Therefore, the OCL containers can be transferred to Java almost unmodified. The explicit enumeration is essentially mapped to a constructor and element-wise addition. The generator specifies which specific set or list implementation is used.

The use of containers for primitive data types requires special treatment. In contrast to OCL, Java cannot store `int` values directly in container structures. Instead, the objectified form “`Integer`” must be used. The type `Set<int>` is therefore transformed into `Set<Integer>`. The boxing of values into their objectified versions available in Java allows application points to remain unchanged, however.

As it is not the object identity but the comparison of the content that is interesting for containers, the comparison == is transformed into a separately generated method that interprets equality on containers as described in Section 3.3.3, Volume 1.

The transformation of OCL/P's comprehension constructs is explained in Table 5.20 below.



OCL comprehension: Transformation of the comprehension	
Explanation	In Java, the forms of comprehension used for sets and lists are supplemented by additional infrastructure.

Itemization	OCL:
	`Set{ a..b }`
	⇓
	Java/P:
	`Set<Integer> res = ...`
	`for(int i = a; i <= b; i++)`
	`  res.add(i);`
	`}`
	`// Reuse of res`

(continued on the next page)

(continues Table 5.20.: OCL comprehension: Transformation of the comprehension)

<table>
<tr><td></td><td>

- The enumeration is realized with a loop.
- The variable `res` allows the reuse for the construction of a set from multiple enumerations, for example, in the form `Set{a,b..c,d..e}`.
- `res` must be assigned a `new HashSet()`.

</td></tr>
<tr><td>Compre-
hension
with
generator</td><td>

OCL

```
Set{ expr | var in expr2 }
```

⇓

Java/P

```
Set<Class> res = new HashSet<Class>;
for(Iterator<Class2> it = expr2'.iterator();
                              it.hasNext(); ) {
  Class2 var = it.next();
  res.add(expr');
}
// Reuse of res
```

- The calculation of the OCL expression is broken up into multiple statements with intermediate results `res`. A reorganization of the calculation is permitted because OCL and the transformation of OCL have no side effects.
- The expressions *expr* and *expr2* are also transformed.
- *expr* has the type *Class*.
- *expr2* has the type `Set<Class2>`.

</td></tr>
<tr><td>Compre-
hension
with
filter</td><td>

OCL

```
Set{ expr | var in expr2, boolexpr }
```

⇓

Java/P

```
Set<Class> res = new HashSet<Class>;
for(Iterator<Class2> it = expr2'.iterator();
                              it.hasNext(); ) {
  Class2 var = it.next();
  if(boolexpr)
    res.add(expr');
}
// Reuse of res
```

- As described in Section 3.3.2, Volume 1, a filter *boolexpr* removes elements from the set.
- The filter obtains its expressiveness in combination with the already formulated generator. Therefore, the transformation is represented with an added generator.

</td></tr>
</table>

(continued on the next page)

(continues Table 5.20.: OCL comprehension: Transformation of the comprehension)

Local variables	OCL
	`Set{ expr | var in expr2, Type x = expr3 }`
	⇓
	Java/P
	`Set<Class> res = new HashSet<Class>;`
	`for(Iterator<Class2> it = expr2'.iterator();`
	`it.hasNext(); ) {`
	`Class2 var = it.next();`
	`Type x = expr3;`
	`res.add(expr');`
	`}`
	`// Reuse of res`
	• A local variable definition acts like a **let** construct.

Combi-nation	Arbitrary combination of generators, local variable definitions, and filters within comprehensions are possible. However, if there are multiple generators, the computational complexity quickly increases.

Table 5.20. *OCL comprehension:* Transformation of the comprehension

The comprehension is related to SQL statements but is significantly more expressive. A transformation of OCL into a database query can at least partly rely on the efficiency of database queries.

5.3.4 A Type as an Extension

OCL allows the use of type expressions such as `Auction` as an extension that simultaneously describes the set of objects of this type that currently exists. For instance, the following can be formulated:

```
forall a in Auction: a.person.size < 500
```
OCL

In order to allow a check of this condition at runtime, access to all objects of the type `Auction` that exist at a point in time is required. Therefore, a corresponding infrastructure must be provided for managing the extensions of the class, i.e. the set of all instantiated objects. This can be realized based on `WeakHashMap`s that cooperate with the garbage collection.

A Boolean variable `OCL.oclmode` describes whether the system is currently evaluating an OCL constraint and therefore whether any new objects created should not be added to the extension[13].

To evaluate the possible constructs *Class*`@pre` and `new(.)`, which are used in method specifications discussed below, additional infrastructure

[13] See Section 3.4.1, Volume 1 regarding object instantiation in queries.

to manage this information about the past is necessary in the system. As method calls can be nested, the call stack is replicated when old values of object instances are stored. This enables any postcondition to access the respective state for the precondition. Here, an efficient management and optimization dependent on the actually required values is important.

5.3.5 Navigation and Flattening

For the various variants of the collections, the flatten operator `flatten` is realized in the form described in Section 3.3.6, Volume 1 and applied when flattening navigation results.

The OCL navigation is set-valued. Because Java does not have a navigation based on set-valued and list-valued structures, these have to be transformed accordingly. Table 5.21 shows an efficient transformation of the navigation.

Navigation: Set-valued and list-valued navigation	
Explanation	In Java, navigation based on container structures must be realized with iterators.
Simple navigation	*a*.*role* ⇓ *a*.*role* • Expression *a* describes a single object. The navigation along the desired association is enabled by *role*. • A further transformation using rules for associations should be applied afterwards. For example, `a` can be encapsulated by access methods. • The navigation can have a set-valued result or can be qualified. This may require an adaptation of the container form.

Navigation over a set	*sa*.*role* ⇓ (Java/P)
	`Set<Class> res = new HashSet<Class>;`
	`for(Iterator<Class2> it = sa'.iterator();`
	`it.hasNext(); ) {`
	`Class2 a = it.next();`
	`if(a.role != null)`
	`res.add(a.role);`
	`}`
	`// Reuse of res`

(continued on the next page)

(continues Table 5.21.: Navigation: Set-valued and list-valued navigation)

<table>
<tr><td></td><td>

- The expression *sa* describes a set of objects. The navigation along the desired association is enabled by *role*. The multiplicity is “`1`” or “`0..1`”.
- See also expressions for simple navigation.
- *Class* is the type of the class reached by the association.
- *Class2* is the type of the initial class.

</td></tr>
<tr><td>Set-valued navigation over a set</td><td>

sa.*role*

⇓

Java/P

```
Set<Class> res = new HashSet<Class>;
for(Iterator<Class2> it = sa'.iterator();
                          it.hasNext(); ) {
  Class2 a = it.next();
  res.addAll(a.role);
}
// Reuse of res
```

- In contrast to the above-mentioned case, the multiplicity is now “`*`”, i.e., `a.`*role* is set-valued.
- Otherwise, explanations from the previous transformation apply.

</td></tr>
</table>

Table 5.21. *Navigation:* Set-valued and list-valued navigation

5.3.6 Quantifiers and Special Operators

Infinite quantifiers, that are, quantifiers for the primitive data types such as `int` and for container structures such as `List<Auction>` are not supported. However, all object-valued quantifiers are finite and can therefore be transformed into Java. A corresponding realization in the form of an iterator was shown in the initial example in this section. Operators such as `any` or `iterate` can be transformed in similar form. The transformation of object diagrams into OCL is also easy. An object diagram acts as a predicate and can therefore be transformed into a Boolean method which is used in the generated code. The `typeif` construct provided by OCL/P for the type-safe transformation of objects is realized by a query with `instanceof` and a cast.

5.3.7 Method Specifications

The following method specification describes how, after the receipt of a new bid, a new time is defined for the end of the auction. Certain time-based dependencies must be ensured:

```
context Time ConstantTimingPolicy.newCurrentClosingTime(          OCL
             Auction a, Bid b)
let    long old   = a.closingTime.timeSec;
       long now   = b.time.timeSec
pre:   status==RUNNING && isInExtension && now <= old
post:  result.timeSec ==
             min(now + extensionTimeSecs,a.finishTime.timeSec)
```

The precondition acts as an `ocl` statement formulated at the beginning of the method and the variables defined with the **let** construct available in OCL are understood as local variables. This specification is transformed into extended Java in Fig. 5.22 omitting the actual method body.

```
                                                          Java/P
class ConstantTimingPolicy {
  public Time newCurrentClosingTime(Auction a, Bid b) {
    let long old   = a.closingTime.timeSec;
    let long now   = b.time.timeSec;
    let boolean precond = (status==RUNNING && isInExtension
                                            && now <= old);

    // Body of the Java function (with no return statement)
    Time result = // Expression of the return statement
    ocl !precond || (result.timeSec
         == OCL.min(now + extensionTimeSecs,
                      a.finishTime.timeSec));
    return result;
  }
}
```

Fig. 5.22. An extended Java method derived from its OCL specification

As described in Appendix B, Volume 1, this method from Fig. 5.22 can be transformed into normal Java. According to the meaning of method specifications described in Section 3.4.3, Volume 1, the postcondition only has to be satisfied if the precondition applies. Therefore, the precondition is evaluated and the result stored in the variable `precond` so that it can be checked in the postcondition. This form of transformation allows us to test multiple method specifications defined or inherited independently of one another simultaneously. Therefore, an integration as described in Section 3.4.3, Volume 1 is not necessary for this purpose.

Depending on the generator settings, `ocl` and `let` statements may be transformed or omitted for efficiency. If an error is detected, an exception is raised, a warning is noted in the log, and, if necessary, an excerpt of the current object is printed. As expected, the `let` construct for the definition of auxiliary variables in Java, which do not belong to the product code, is

realized with local variable definitions. The code shown in Fig. 5.23 arises, with the catching of exceptions omitted for the sake of simplicity.

```
class ConstantTimingPolicy {                                          Java
  public Time newCurrentClosingTime(Auction a, Bid b) {
    long old   = a.closingTime.timeSec;
    long now   = b.time.timeSec;
    boolean precond =
            (status==RUNNING && isInExtension && now <= old);
    // Body of the Java function (with no return statement)
    Time result = // Expression of the return statement
    if(precond && !(result.timeSec
         == OCL.min(now + extensionTimeSecs,
                    a.finishTime.timeSec))
            alert(...);
    return result;
  }
}
```

Fig. 5.23. OCL transformed into standard Java

This relatively simple transformation is contrasted with a substantially complex transformation of method specifications if the postcondition accesses the original values of the variables at the beginning of the method call. Transforming such accesses requires an infrastructure for the generation that determines which values from the beginning of the method execution may be required and places them in a suitable intermediate storage. As described below, this can involve a considerable amount of effort. Let us look at the following example adapted from Section 3.4.3, Volume 1 which describes the effect of a person changing companies under the assumption that the company is already in the system:

```
context Person.changeCompany(String name)                          OCL
pre:  exists Company co: co.name == name
post: company.name           == name &&
      company.employees      == company.employees@pre     +1 &&
      company@pre.employees == company@pre.employees@pre -1
```

In the postcondition, `company@pre` is used to access the previous company of the person changing companies and `company.employees@pre` is used to access the number of previous employees of the new company. The code generator can easily identify the expression `company@pre` and store its content. This has the same effect as the creation of an internal auxiliary variable. The same applies for the expression `company@pre.employees@pre`:

OCL

```
context Person.changeCompany(String name)
let     Company companyPre = company;
        int     employeesPre = company.employees
pre:    exists Company co: co.name == name
post:   company.name            == name &&
        company.employees       == company.employees@pre +1 &&
        companyPre.employees    == employeesPre -1
```

However, `company.employees@pre` creates difficulties because the earlier state of the new company is accessed. It is actually impossible for a code generator to generate code that, at the beginning of the execution of a method, guesses for all cases which object will be the new `company` object and then saves the former state of this object. This problem becomes even clearer with the expression `ad.auction@pre[id]`, which can be used in the class `AllData` to select an auction with the identifier `id`. As `id` is only evaluated when the method has been executed, it is not clear which auction of the original state will be selected by `ad.auction`. There are three strategies for handling this:

1. The old states of *all* `Company` objects are stored. However, even for small test data structures this results in a significant loss in efficiency and cannot be implemented for testing of product systems with large data sets.
2. For tests, an infrastructure that performs an internal logging of all attribute changes, including the original values, is generated. This also requires a suitable infrastructure but under some circumstances, has the advantage that the change history of a failed test can be analyzed.
3. The code generator warns of the inefficiency of the specification or even rejects the specification with an instruction to find a more efficient formulation. In most cases, the developer has a good idea of which object states actually have to be saved and stores these explicitly in `let` statements.

If an increase in efficiency through a manual transformation is desired, in the example, the relevant `company` object can be "guessed" from the precondition and thus the precondition is simplified:

OCL

```
context Person.changeCompany(String name)
let     Company newCo = AllData.ad.company[name]
pre:    newCo != null
post:   company.name            == name &&
        company.employees       == newCo.employees@pre +1 &&
        company@pre.employees == company@pre.employees@pre -1
```

To ensure that the new company actually matches the company defined in the **let** statement, `company==newCo` can also be included in the postcondition. This form requires only minimal additional storage space but is already so detailed and implementation-related that sometimes, a direct implementation may be preferable.

5.3.8 Inheritance of Method Specifications

As described in Section 3.4.3, Volume 1, the stereotype ≪not-inherited≫ determines whether a method specification applies for the implementations of the subclasses. Thus, in general we also have to test the inherited preconditions and postconditions. For a non-redundant implementation, the conditions to be checked are not formulated directly as assertions and are instead outsourced to stand alone methods that are available in suitable form in subclasses. This technique is called "percolation" [Bin99] and results in the implementation represented in Fig. 5.24.

```
class ConstantTimingPolicy {                                         Java
  // Precondition without/with predefined parameters
  boolean preNewCurrentClosingTime(Auction a, Bid b) {
    long old   = a.closingTime.timeSec;
    long now   = b.time.timeSec;
    return preNewCurrentClosingTime(a,b,old,now);
  }
  boolean preNewCurrentClosingTime(Auction a, Bid b,
                                        long old, long now){
    return status==RUNNING && isInExtension && now <= old;
  }

  // Postcondition
  boolean postNewCurrentClosingTime(Time result,
                    Auction a, Bid b, long old, long now) {
    return result.timeSec
         == OCL.min(now +
                    extensionTimeSecs,a.finishTime.timeSec);
  }

  // Actual function
  public Time newCurrentClosingTime(Auction a, Bid b) {
    long old   = a.closingTime.timeSec;
    long now   = b.time.timeSec;
    boolean precond = preNewCurrentClosingTime(a,b,old,now);
    // Body of the Java function (with no return statement)
    Time result = // Expression of the return statement
    if(precond &&
             !postNewCurrentClosingTime(result,a,b,old,now))
            alert(...);
    return result;
  }
}
```

Fig. 5.24. OCL constraints transformed into Java predicates

An advantage of this approach is that a test driver can also check whether the precondition of the method specification is satisfied by checking it against the given test data in advance. This prevents a test being evaluated as a success because the test data was set up incorrectly such that it does not satisfy the precondition.

We can use the approach represented in Fig. 5.24 in the same way for invariants so that these are also available in the subclasses for tests.

5.4 Executing Statecharts

As David Harel, the inventor of Statecharts likes to comment: "Buildings are there to be, but software is there to do". Statecharts can naturally be used to describe behavior completely, and therefore the aspect of code generation, that is, the execution of Statecharts, is particularly interesting. This section addresses realization strategies for Statecharts but without describing complete transformation algorithms. The goal of this section is also to describe the relationship between syntactic concepts of the Statechart and Java code elements. Thus, it gives the reader the opportunity to transform Statecharts into code manually. This description will also help the reader to understand Statecharts and their use during code generation. These aspects are equally important. The transformation of Statecharts into Java is used as both, the definition of semantics and for improving the intuitive access to Statecharts. The realization strategies discussed below show alternative but semantically equivalent options for transforming Statecharts into Java. The selection of a suitable transformation strategy for a project is therefore dependent on the desired flexibility and efficiency.

Generating code from Statecharts is not as widespread as generating code from class diagrams. However, the use of Statecharts for implementation is continually increasing. Embedded systems are the trailblazers here: "... automatically generated code can and is being used today in a variety of hard real-time and embedded systems." [Dou99, p. 156].

The variants for transformation discussed below are in no way complete and represent merely some of the transformation options. The stereotypes already introduced (≪statedefining≫, ≪completion:ignore≫, etc.) can be used to choose between these variants. However, not everything can be described with stereotypes, which is why templates and scripts are used for generating code from Statecharts.

5.4.1 Method Statecharts

The states of a Statechart can be interpreted in different ways. In a method Statechart, such as the one represented in Fig. 5.25, the diagram states represent intermediate states within a method sequence. The diagram states are therefore differentiated by the program counter. They can be split in two

classes. On the one hand, there are diagram states that correspond to points between statements that are continued by means of spontaneous ε transitions. On the other hand, there are states in which the result of a method call started in an incoming transition is awaited. Source states of spontaneous transitions are used primarily to model the control flow, which also allows to describe branching. Source states of transitions labeled with the stimulus `return` represent real interruptions in the execution of a method.

A method Statechart is transformed canonically to the described method and its body.

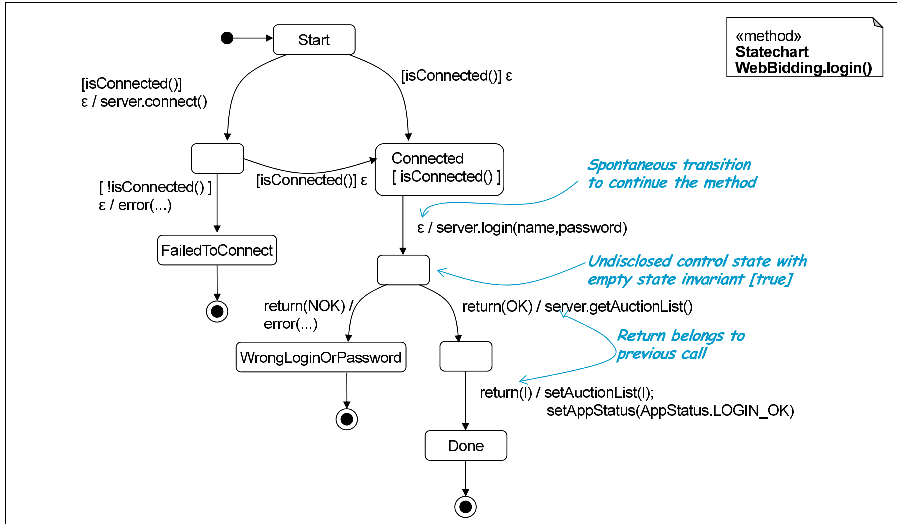


Fig. 5.25. Representation of intermediate states of a method

The handling of ε loops within a Statechart is particularly interesting. The existence of a ε loop means that the control flow within a method Statechart has a loop. The Statechart itself is therefore no longer completely capable of describing the behavior of the method. The initialization, the body, and the termination condition of the loop must each be described by actions. There is no guarantee that the loop will definitely terminate, but we can assume that modeled behavior terminates in principle. This assumption is useful when in an appropriate development approach, loops are generally checked with tests such that a nonterminating loop would be detected.

5.4.2 The Transformation of States

The Statecharts that are not used to represent a method sequence do not contain any control states and therefore no spontaneous transitions either. The

states of such a Statechart therefore correspond to the data states of the object. This means that it must be possible to reconstruct the diagram state of the Statechart from the data state of the object. This was discussed with the transformation into simplified Statecharts in Section 5.6.3, Volume 1 and a technique for implementing diagram states was shown.

Lightweight: Using State Invariants

The strategy of using the pairs of disjoint state invariants of the simplified Statechart to *calculate* the diagram state from an object is regarded as particularly simple. Fig. 5.26 shows an example of this type of transformation. Here, the left-hand transition is given priority over the middle transition as its precondition is evaluated first. If the two preconditions overlap, that is, *precon1*`&&`*precon2* is not equivalent to `false`, then a design decision was taken that selects a special implementation from the set of possible implementations.

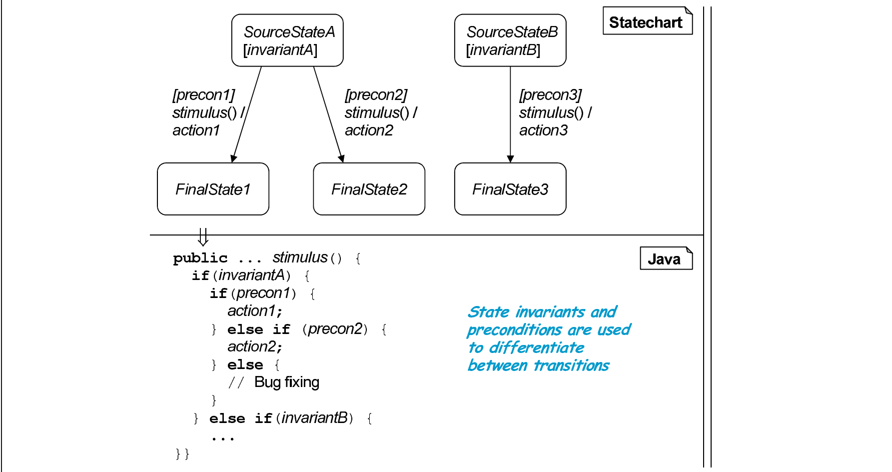


Fig. 5.26. Transformation using state invariants

The advantage of this transformation is that no additional attribute is required to save the diagram state in the object. In contrast, in the worst case, the state invariants of all states have to be evaluated, which can lead to a significant loss of efficiency. Therefore, this transformation is only useful for state invariants that can be evaluated efficiently. In most cases, there are a number of possibilities for optimization because state invariants of "related" states often have common subconditions that only have to be evaluated once. The same applies for the evaluation of the preconditions of transitions with the same source state. If, for example, *precon1*`<=>!`*precon2*, the internal if query in Fig. 5.26 can be simplified to `true`.

Optimizations for the code generation such as those described above cannot generally be recognized automatically. However, we can justifiably hope that the tools for generating code from models will integrate similar optimization algorithms in the near future. Today this is already the case for compilers for textual programming languages. Until then, we must expect that the increased developer efficiency from using abstract models will lead to certain efficiency disadvantages for the realized code. Under some circumstances, similar to manufacturers of embedded systems, we can see that after completion of the models for simulation and validation purposes, an additional manual step is useful to optimize the result. We can use refactoring techniques at the level of the target language, just as well as additional control mechanisms during the generation of the code. For example, optimizations can be proposed to the code generator by adding priorities or asserting the disjunction of preconditions.

States as Predicates

A variation of the transformation shown is presented in Fig. 5.27. Here, the evaluation of individual state invariants and actions was outsourced to separate methods. The auxiliary methods for transforming actions can be reused, if applicable and if various transitions have the same action. The outsourcing of the evaluation of state invariants to separate predicates also has the advantage that the diagram state of the Statechart can be determined at arbitrary points in the code. This is particularly helpful for methods whose behavior is not specified in the Statechart but is still dependent on the states modeled in the Statechart.

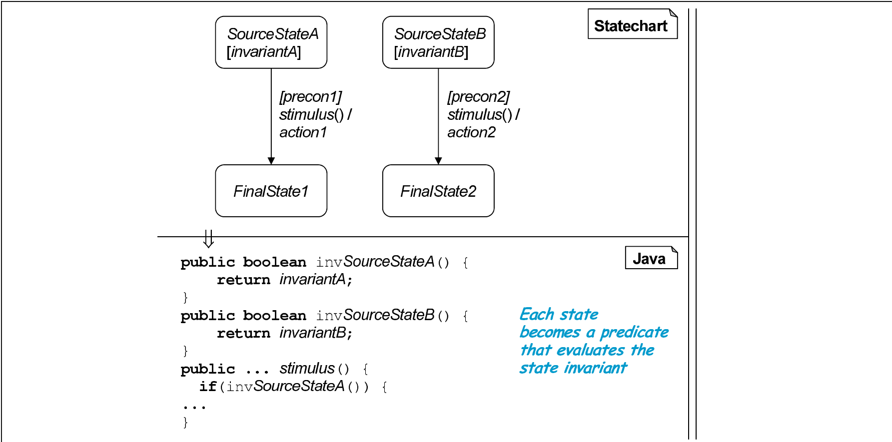


Fig. 5.27. Predicates evaluate state invariants

Itemization Attribute as Repository for the State

If the evaluation of the state invariants is too inefficient, the standard technique generally used today is to store the diagram state explicitly in the state space of the object in the form of an enumeration attribute. Checking the `status` attribute is sufficient to determine the diagram state. Fig. 5.28 shows the code generation that can be used.

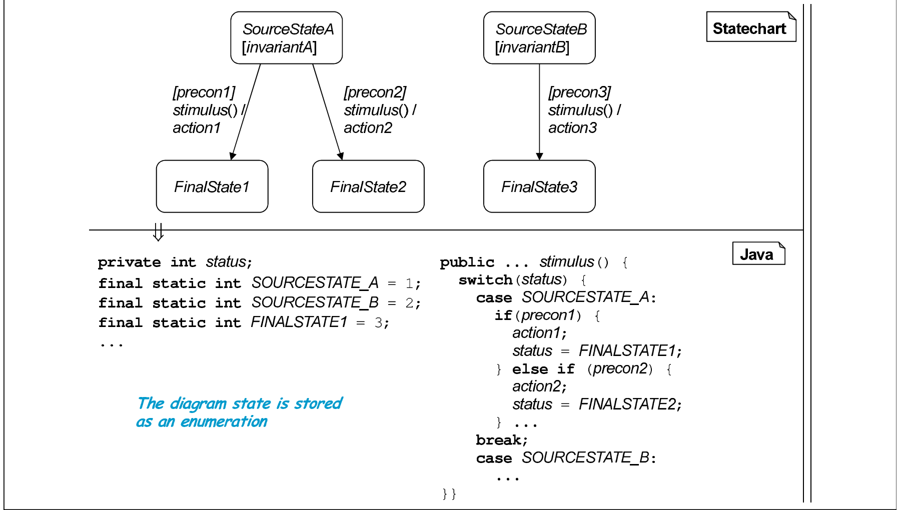


Fig. 5.28. Diagram state is stored in the attribute

Complete Statecharts

Using an attribute to store the diagram state improves the runtime efficiency but does not mean that preconditions of alternative transitions do not have to be checked. However, we can use the last respective precondition and the state invariant in an `assert` statement to be used for test purposes instead of in the constructive part of the implementation.[14] Fig. 5.29 shows a correspondingly modified transformation.

If the Statechart was not completed explicitly as described in Section 5.6.3, Volume 1, this completion can also be performed during the code generation dependent on the selected semantics. Here, we can use the `default` statement, which can catch all situations not covered by explicit transitions. We can also introduce and adopt a further value in the enumeration of the states in the form `ERROR==-1` in such situations.

[14] To obtain meaningful test results, using the test framework discussed in Section 6.2.3 instead of the `assert` statement provided by Java is recommended.

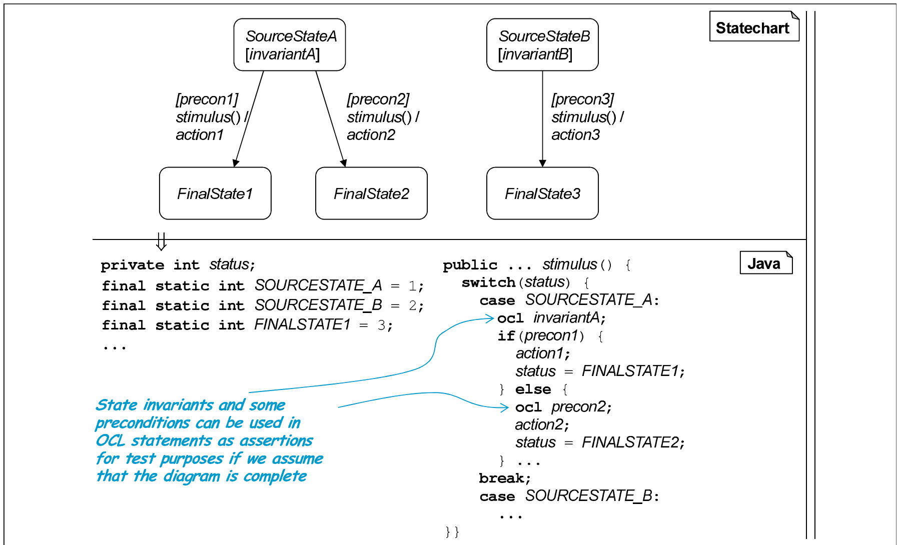


Fig. 5.29. Use of state invariants as assertions

We can use a comprehensive `try-catch` statement to catch exceptions caused due to the stereotype «exception». However, depending on the selected transformation strategy, we have to use multiple such statements in various methods or `case` alternatives.

Heavyweight: State Design Pattern

A further option for transforming the state concept is, for example, the use of the state design pattern, which is described in [GHJV94]. This design pattern is classified as heavyweight because it leads to a number of additional classes. The state of the object actually modeled is outsourced to a separate state class. This class has multiple subclasses each corresponding to one state of the object actually modeled. The current state of the object is stored as a reference to the currently active state object. The behavior is not implemented directly in the modeled method but *delegated* to this currently active state object. This means that the behavior, which is distributed among the transitions in the Statechart, is not aggregated within a single method but is instead grouped according to the states in different classes. This offers the flexibility of modifying the behavior on a source state by forming additional subclasses or rewiring the transitions. However, the notational and operative effort involved in the state design pattern is huge. The state objects have to be managed by either creating such objects dynamically or saving them in a pool of objects. Fig. 5.30 therefore shows only a small excerpt of the transformation in a state design pattern. The code parts *invariantA′* and *action1′*

have to be adapted correspondingly as the attributes and method calls have to access the source object `k` instead of `self`.

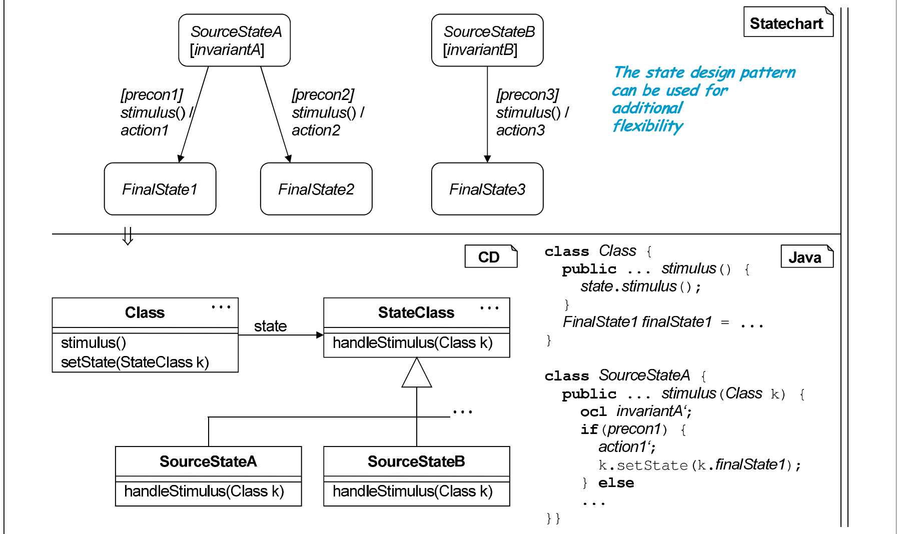


Fig. 5.30. Heavyweight: state design pattern

Using the state design pattern is mainly recommended if the created flexibility can justify the additional effort for the generation. As the transformation of a Statechart into Java code is usually automated, a manual intervention in the generated code is generally not useful and the state design pattern will, hence, be less appropriate.

5.4.3 The Transformation of Transitions

Stimuli

As shown in Fig. 3.37, we differentiate between three types of stimuli. Spontaneous transitions and the receipt of return results only occur within method Statecharts. There is no differentiation between the transmission of asynchronous messages and the method call in the Statechart. The differentiation between a method call and the use of message objects for asynchronous transmission is, however, relevant for the transformation into code.

If the stimuli are objectified, events are typically transformed in the form of a class hierarchy with an abstract superclass `Event` whose subclasses correspond to individual message types. Messages are created by calling the corresponding constructor and are executed after transmission and through a scheduling mechanism on the target object. A number of frameworks and

middleware components, such as CORBA [OH98], are available for transporting these messages. Further variants are, for instance, the serialization of the message objects with XML [W3C00] or the transmission by means of a self-defined, typically more efficient protocol. Due to the high number of solutions available, we will not examine this technical aspect of the communication in greater depth here. For the transformation of Statecharts, the only important thing is that the message object is transferred to the object to be processed by calling a suitable method. Fig. 5.31 illustrates a possible realization based on a double `switch` statement. By defining methods for each specific message object, or by delegating the processing of a method to dependent objects, we can apply the previously discussed variants for code generation.

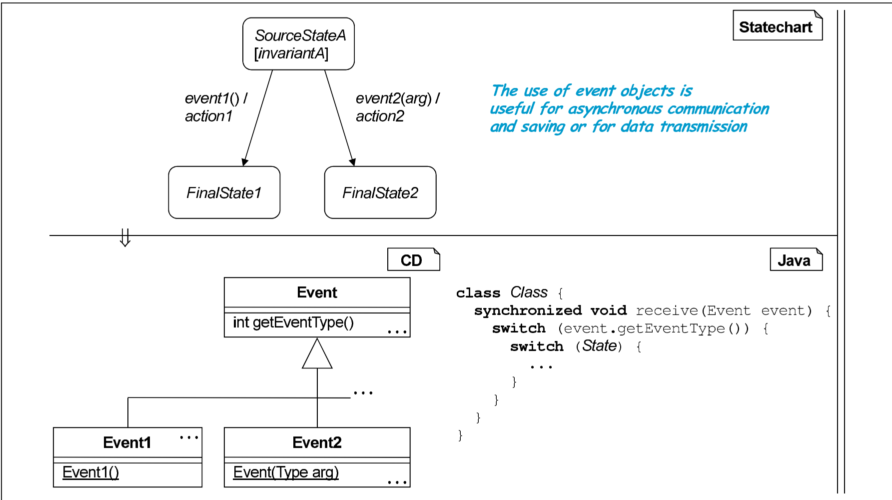


Fig. 5.31. Use of event objects as stimuli

An advantage of the use of asynchronously communicated messages is the avoidance of recursive object calls. A message object is always processed with exclusive access to the object state. Parallel processing of multiple messages is excluded. In order to guarantee this, we use suitable synchronization mechanisms of the programming language Java. In a Statechart, the processing of messages and method calls can also be mixed. For the Statechart, it is ultimately irrelevant whether the stimulus is processed by the special method `receive` for message processing or by a normal method call. We must merely ensure that within an action of the Statechart, there are no further method calls to the same object that have a state-modifying effect on the Statechart. This condition, referred to as recursion-freedom, was discussed in detail in Section 5.1, Volume 1.

Actions

The action descriptions of a Statechart consist of two possible components. Once we have adapted attribute accesses etc., we can include actions formulated procedurally in the generated code as described in Section 5.1. If we introduce an additional attribute to store the diagram state, an additional assignment of the new state is necessary at the end of each action. The transformation shown in Fig. 5.28 illustrates this.

If we used additionally or exclusively OCL postconditions for the actions, then no operational code can usually be generated from them. Therefore, we cannot use this type of Statechart for programming. These types of postconditions are therefore typically used only for abstract specification of behavior, which is manually transformed into executable code for an implementation and then can be used for tests. The postconditions can therefore only be transformed into code using `ocl` statements.

Existing Forms of Code Generation for Statecharts

Of course, the form of transformation of Statecharts into code described above is not the very first. Various tools, such as Statemate or Rhapsody [HN96], as well as some of the newer UML-based tools and approaches such as [SZ01], [BS01a], and [BLP01] transform their version of Statecharts into productive code. In some cases, concepts such as parallel states, a history mechanism, pseudo states, or real parallelism are also transformed into the code. These concepts are not defined here—due, amongst other things, to the previously predominant use of Statecharts for modeling distributed and embedded systems which usually have a greater share of control than business systems with the focus on a heavy data load [Dou98, Dou99].

Statecharts were introduced in [Har87]. [vdB94] compared the variety of semantics that had arisen by then. The comparison [vdB01] also includes the UML semantics of the Statecharts. An interesting feature of these different semantics is that they sometimes describe different behavior and therefore result in different implementations of the Statechart. In other cases, they merely use different mechanisms to assign the essentially identical semantics to the Statecharts.

Particular attention is given to the handling of the prioritization and uninterruptibility of transitions of different hierarchy levels and the closely related "run to completion" problem for Statecharts. While the outer transitions (seen from the source state) are preferred for embedded systems, in [HG97], for example, this prioritization is reversed for object-oriented Statecharts and preference given to inner transitions. This approach allows the modeler to decide by using stereotypes, which leads to greater flexibility. In the UML/P Statecharts, "run to completion" is irrelevant—due to the underlying machine model based on Java, the assumption is that the exceptions, which occur as stimuli, are caused by the Statechart itself or by a method

called from an action of the Statechart, and therefore continuing the transition to a natural end has only limited use. The possibility of parallel processing of transitions in the same object is excluded by Java´s synchronization mechanism.

5.5 Transformations for Sequence Diagrams

A sequence diagram is by nature exemplary. It shows one single possible sequence of a system run that typically allows alternative sequences depending on the current object structure, the contents of the attributes, and the system environment. Describing an exemplary sequence is not really suitable for constructive code generation. From a single diagram, complete code can only be generated if a method does't contain branches or iterations in its control flow. Test drivers and test observations typically have such a simple structure.

Therefore, sequence diagrams are used primarily for modeling tests and code generation is useful for performing and checking test runs.

5.5.1 A Sequence Diagram as a Test Driver

Fig. 5.32 contains a typical sequence diagram from which a method is to be generated for the object `t`. The method executes the calls labeled with the stereotype ≪trigger≫.

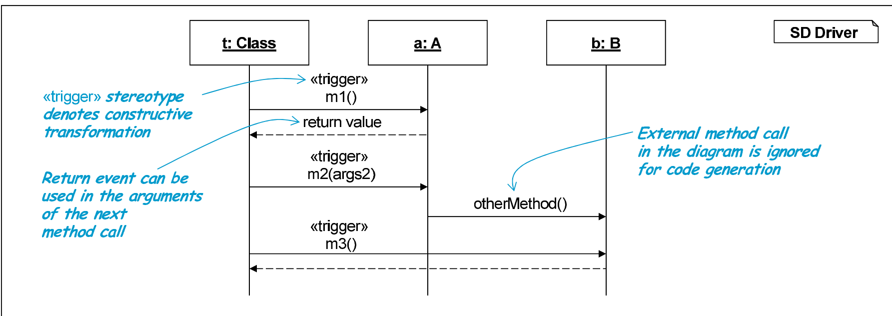


Fig. 5.32. Sequence diagram with driver

The method to be generated requires a name which can be extracted from the diagram name. In combination with the prefix `runSD` defined as standard, this results in the method name `runSDDriver` generated in the class `Class`.

Similar to the constructive transformation of object diagrams, the signature of the resulting method is characterized by the free variables of the sequence diagram. In the example in Fig. 5.32, therefore, at least the two other

objects a and b are needed as parameters. Free variables that appear the first time as arguments of a ≪trigger≫ call are also included as parameters. In contrast, free variables that are used for the first time in returns are matched to this return value. Calls and returns between other objects of the sequence diagram as well as calls to driver objects do not contribute to the generated method. The result is the code shown in Fig. 5.33.

```
class Class { ...
  public void runSDDriver(A a, B b, Type2 args2) {
      Type value = a.m1();
      a.m2(args2);
      b.m3();
  }
}
```

Java

Fig. 5.33. Driver generated from a sequence diagram

Whereas they are not obvious from the sequence diagram itself, the types of the variables specified in the sequence diagram can be determined from the context—for example, from the signature of the classes A and B.

We can also use the technique for the constructive transformation of parts of a sequence diagram if we specify the method call of the generated method in the sequence diagram itself. Fig. 5.34 describes a dummy object that requires a simple implementation of the method foo(). This is generated using the same technique and is thus available for the test specified in Fig. 5.34. The target object b of the method to be called by foo() is defined as an attribute in the class Dummy. Values are typically assigned to the object structure by an object diagram.

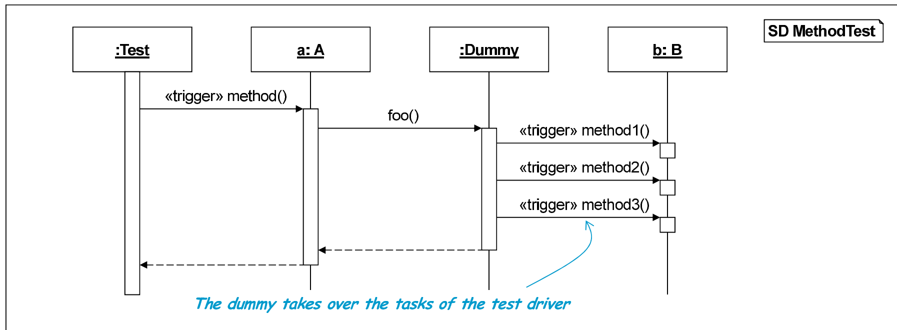


Fig. 5.34. Sequence diagram with driver and dummy

5.5.2 A Sequence Diagram as a Predicate

Just like an object diagram, a sequence diagram describes an exemplary property of the system. In contrast to an object diagram, however, this property cannot be checked from a snapshot of a system and instead, has to be checked during a system run. In order to be able to check interactions that occur during such a run, a corresponding instrumentation of the code is required. At the beginning and end of each method observed, there must be a notification about the call or the termination of this method. The abnormal termination by an exception must be logged. This can take place within the instrumented method but also using the adapter design pattern [GHJV94]. We can form this type of adapter, for example, by redefining the method in a subclass. The principle for the class `A` from Fig. 5.34 is represented in Fig. 5.35.

```
class Ainstrumented extends A { ...                                    Java
  public Type method() {
    // Log method call (object, method, empty argument list)
    SDlog.call(this, "method", new Object[] {});
    Type result;
    try{
      // Actual call
      result = super.method();
    catch (Exception ex) {
        // Log exception
        SDlog.exceptionReturn(this, "method", ex);
        throw ex;
    }
    // Log return + result
    SDlog.normalReturn(this, "method", result);
    return result;
  }
}
```

Fig. 5.35. Adapter for code instrumentation

In practice, however, it is useful to retrieve this information from the call stack via the virtual machine instead of transferring it to a global object `SDlog`.

The algorithm for recognizing a sequence diagram interprets a sequence diagram as a regular expression as specified in Fig. 6.14, Volume 1. The regular expression is initially realized in a nondeterministic finite automaton[15]

[15] The automaton is used to recognize input sequences; it has nothing to do with the Statechart described in Chapter 5, Volume 1.

which allows this regular expression to be recognized. Fig. 5.36 demonstrates this in an example.

A sequence diagram can contain prototypical objects for which there is initially no assignment to real objects. This assignment takes place during the first interaction with a suitable object. Therefore, the states of the nondeterministic automaton are extended with the configuration of these objects and further free variables. An automaton can therefore be in the same state multiple times with different object configurations.

After each interaction, any subsequent OCL constraints in the sequence diagram are checked and the configurations that do not satisfy the constraint are removed. In the automaton, we can represent this with an additional transition that does not process any interactions but has a precondition. Starting with a configuration in the initial state, in some circumstances an empty set of valid configurations is reached.

A sequence diagram is *satisfied* if, after the execution of the test, the reached configurations contain the final state. As a byproduct of the check, the free variables and the prototypical objects contain concrete values.

The different semantics for sequence diagrams that can be selected via the stereotype ≪match≫ are transformed by the restriction of interactions described in Fig. 6.14, Volume 1. This allows us to ignore certain forms of interactions.

The mapping can be illustrated using the sequence diagram shown in Fig. 5.36. The illustration ignores specific values of the parameters. These can be handled in the same way as the OCL constraints, i.e., they can also be checked. Fig. 5.36 also contains the automaton that is used to recognize the sequence diagram.

The well-known construction for making the automaton deterministic and minimizing it [HU90] usually cannot be used due to the configurations which influence the enabledness of further transitions and the evaluation of the OCL constraints. The automaton could be made deterministic, for example, if the objects represented in the sequence diagram could be assigned to real objects in advance. On the other hand, an automaton that arises from a sequence diagram labeled with the stereotype ≪match:complete≫ does not have any loops and is therefore already deterministic.

A sequence diagram that has already been used in part for the constructive generation of drivers can also be used for a check. The fact that the test driver is also instrumented creates only minimal additional effort but it is necessary to be able to check the order of the messages appearing and the OCL constraints.

5.6 Summary of Code Generation

Chapter 4 and this chapter discuss basics for code generation, beginning with the expressiveness of UML/P up to a useful architecture of a code genera-

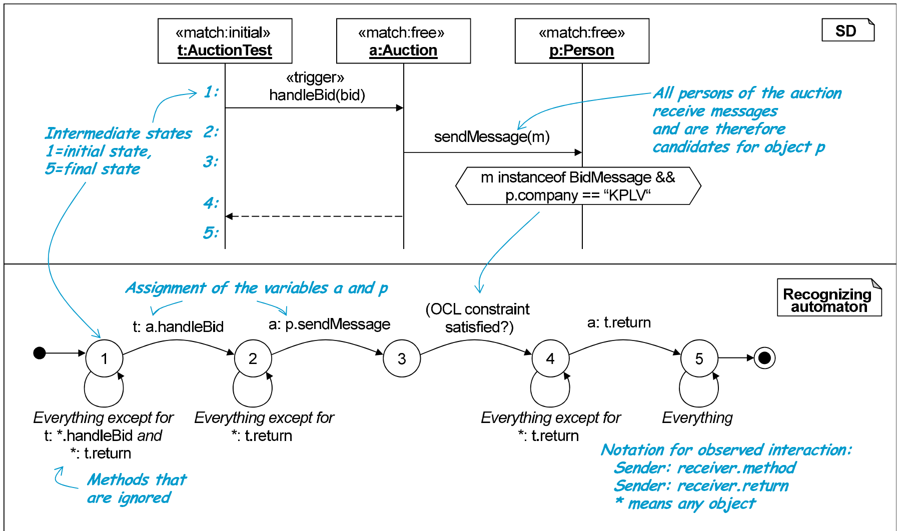


Fig. 5.36. Recognizing automaton from a sequence diagram

tor that has the required flexibility and parameterization to satisfy various target platforms and different areas of application of UML/P. Instead of an actual representation of the *scripts* and *templates*, transformations were explained in abstract form or motivated by examples. This chapter described the transformation of the individual UML/P notations into code based on these transformation rules.

Outlook for Code Generation

The first generation of CASE tools show that when code generation is used, some disadvantages have to be taken into account alongside the advantages described in this chapter. The maintainability of a system developed with code generation is dependent on whether the code generator is available during the period of use of the product system. If this is not the case, the only solution is to modify the generated code further manually. If, for example, the technical platform is migrated, then developers with the ability to adapt ("program") the generator must also be available.

If a code generator is developed in an independent project, then it should be written in its simplest form and it must be possible to reconstruct its use for maintenance and evolution of the system when required. However, the complexity of today's source languages, such as UML, stands contrary to this desire. Therefore, as a reasonable alternative, an externally developed, product-mature code generator was discussed. Such a generator must demonstrate sufficient stability to be available for a longer time. In order to prevent or at least minimize incompatibilities in the event of version changes,

here, a standardization similar to the standardization of compilers and the semantics of the underlying languages is needed. The current discussions on the standardization of UML contain no standardization that would be sufficiently detailed for this purpose. Nevertheless, the forthcoming interoperability of today's UML tools could have a certain standardizing effect on code generation.

6

Principles of Testing with Models

Quality is never an accident;
it is always the result of intelligent effort.

John Ruskin

Testing is an important part of managing the quality of all parts of the product system. Testing the implementation for robustness, conformity with the specification, and correct implementation of the requirements must therefore be an essential part of any quality-oriented and model-driven software development method. This chapter explains the basic principles of testing that are necessary to achieve this.

6.1 An Introduction to the Challenges of Testing 162
6.2 Defining Test Cases 177

© Springer International Publishing AG 2017
B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_6

A system that successfully supports the user must be free of errors or at least should have as few as possible. As software systems are usually very complex, we must assume that any system practically relevant will never be completely free of errors. Therefore, when developing a system, it is important that we minimize the number of errors and the malicious effects of those errors as far as possible given the time and personnel resources available. Furthermore, we need to achieve the level of quality required for the application domain. Testing the software system is an important means of detecting errors. It is generally accepted that testing can make up approximately 40–50% of the total software development effort [Kru03, Mye01].

Tests are recognized as an essential activity for quality management in both traditional and agile approaches. Therefore, the techniques discussed in this chapter are not limited to agile methods; they have actually been in use in various forms for a long time. However, the intensive integration of testing in development methods, the desire to define tests as easily and effectively as possible, and the consistent use of UML for generating tests and code mean that existing approaches for defining tests need to be adapted and extended.

The extensive amount of literature available on the subject of testing in general, and on testing object-oriented systems in particular, shows that there are a lot of different approaches and techniques for developing tests. Reproducing all of the knowledge about developing and managing tests that this literature contains would go far beyond the capacity of this chapter. Instead, there are references included to relevant, more extensive literature. As far as terminology is concerned, the contents of this chapter is based primarily on [Bin99], [Mye01], [Bal98], and [Lig90].

This chapter does not provide a general introduction to test procedures and technologies; instead, it provides a compact definition of the terminology involved in testing and discusses relevant basic technical principles. Chapter 7 then implements these basic principles using UML/P.

For additional approaches for developing tests, see [LF02] for a pragmatic look at the "test-first" approach, the extensive collection of testing techniques in [Bei04] and [Bei95], the comprehensive book [Bin99] on object-oriented tests, [RBGW10] for model-based tests, and [Bal98] for a compact overview of the theory of testing. The basic principles of [Mye79] still apply. Books about testing in individual domains are also useful—for example, [Vig10] for embedded systems or [Sax08] for the automotive industry.

6.1 An Introduction to the Challenges of Testing

There are a number of approaches for ensuring or improving the quality of software. We can use these approaches in one individual software development project or across all projects. "*Quality*" is a term that is widely used but difficult to define. [Lig90] specifies a number of quality properties—some of which we can check using tests—including properties important for using

the product, such as *functional correctness* and *robustness*. We can use statistical tests to check further quality properties, such as runtime and memory efficiency, at least to a limited extent.

6.1.1 Terminology for Testing

Fig. 6.1 contains several definitions of the terms "test" and "testing" that can be found in literature.

The literature available on the subject of testing contains a number of different definitions for the widely used terms *test* and *testing*:

- "*Testing* is the process of executing a program with the intention of finding errors." [Mye01]
- According to [Som10] is testing the execution of a software implementation on test data and the examination of the results and the operational behavior in order to check, whether the software behaves as required.
- The application of test, analysis, and verification procedures serves primarily to check the quality properties of functional correctness and robustness (according to [Lig90]).
- [Bin99] provides a detailed explanation pointing out that a test is the design and implementation of a special form of software system. It examines another software system in order to find errors. Tests have to be designed such that they cooperate with the system under test based on its physical interfaces, the runtime environment and structure of the system under test. Manual tests still have their role, but testing mainly means development of automatic tests.

  [AM01] emphasises the distinction between (1) unit tests and (2) acceptance tests. Developers write unit tests parallel to the code and users write acceptance tests after the use cases are designed.
- [PKS02] puts emphasis on the process of testing, consisting of planning, preparation, and measurement with the goal of determining the features of an IT system and demonstrating the difference between the current state and the required state.

Fig. 6.1. Terminology definitions for "test" and "testing"

It is typical for Extreme Programming that it lacks a property-oriented definition of the term "test". Instead, the approach gives an operative description which simultaneously defines who is responsible for developing tests. The description is also limited to two simple types of tests [AM01].

The definition from [Bin99] is particularly suitable as a basis for this book. From this definition, we can derive the characteristics for tests summarized in Fig. 6.2 which we use as the basis for the testing activities described in this book. However, there are a number of exceptions to this characterization which are also referred to as tests. Some exceptions are *manual, interactive*

tests performed by the user during the *acceptance test*. Another alternative is *symbolic testing*, which involves symbolic calculations—that is, the system being tested does not actually "run". *Load tests* in real distributed systems do not always produce the same results; instead, the repeated execution of such tests determines average values. This book does not cover any of these types of tests.

1. A test—in contrast to a static analysis—*executes* the system being tested.
2. Tests are *automated*: since manual tests are very time-consuming for large systems, the quality of tests would otherwise suffer or project team members would only perform tests and do nothing else.
3. An *automated test* sets up the test object, all necessary test data, and potentially necessary context. Then it executes the test and checks the test results automatically. The *success* or *failure* of the test is recognized and reported by the test run.
4. A test suite is a software system itself that runs together with the system being tested.
5. A test is an *example*: it works on a set of input data—the *test data*.
6. A test can be *repeated* and is *determined*: it always produces the same results for the system being tested.
7. A test is *goal-oriented*: it either demonstrates the presence and effects of an error or alternatively, shows that the system has the required functionality for the test case and is robust with regard to the test data.
8. Where a system has been modified, a test can prove that it demonstrates the same behavior as the original system and can thus help to avoid errors during further development.

Fig. 6.2. Characterization of tests

In recent years, it has become increasingly important to define fully automated tests. [FG99], for example, describes good arguments for doing so. It also provides a detailed differentiation between such tests and semi-automated or manual approaches.

It is also important to differentiate between the activity of testing, with the goal of finding errors, and *bug fixing or correcting errors*, which takes place after testing. Further measures for quality management include a *code inspection* (in which the source code written by developers is examined for potential errors) and *verification* (which cannot usually be performed for industry-relevant systems—or rather, it cannot usually be performed *yet*). In contrast to a test, however, verification could prove that an implementation is completely *correct* with regard to a specification.

Table 6.3 classifies test terminology according to the type of system element to be tested and indicates who is normally responsible for defining and performing these tests. Depending on the development process used, the re-

sponsibility for developing tests lies either with a separate test team or with the developers themselves.

Test type	Who creates (or executes) the test?	Test object
Acceptance test	User, mostly interactively	The product system installed
System test ("acceptance test")	Test team with users help	The instrumented product system in the test environment
Subsystem test ("integration test")	Test team, developers	Subsystem
Class test ("module test", "unit test")	Developers, test team	Class: Auction — #void incNumberOfBids() +boolean bid(Person, Time, Money)
Method test	Developers	Method

Table 6.3. Tests at various levels in the system

6.1.2 The Goals of Testing Activities

Tests are very important in the portfolio of quality management measures. This is because we can create and execute automated tests systematically for a specific purpose with an acceptable level of effort. Test code usually exceeds the size of the code to be tested but has a much simpler structure. In contrast to the product code, it is much less important for the test code to be elegant and non-redundant. It is acceptable, therefore, to develop tests using copy and paste [Den91, Fow99], although it is also helpful to identify reusable abstractions and separate those into reusable methods. As our auction system works with large sums of money and is subject to strict time-based conditions, the required system quality in terms of the software being error-free was rated particularly highly. Accordingly, 63% of the total code developed is part of the test system.[1] Nevertheless, developing and maintaining the test cases made up approximately only 20% of the overall effort, which is comparatively low.

The additional costs involved in developing and maintaining an automated test suite are therefore acceptable. A study in [KFN93] estimates that the break-even point between the costs and benefits of automated tests is comparable with approximately ten manual test cycles. Where systems evolve dynamically and are subject to change even after installation, the number of ten manual test cycles is quickly reached. Thanks to tools and

[1] Measured in Java-LOC = lines of code, including comments.

frameworks that are now available, such as JUnit [JUn11, BG98, BG99], the process of defining test cases is also becoming more efficient, meaning that the break-even point should reduce even further. The advantages of automated tests are:

- Errors in the program logic, errors in border cases, and incorrect coding are detected early and almost always eliminated on occurrence.
- The confidence of the developers in their own code and the code of colleagues is significantly higher than usual due to the tests.
- The self-confidence of a developer to adapt code that he has not developed himself in order to meet modified requirements increases due to the automated, repeatable tests and the knowledge about the system functionality embedded in those tests.
- An extensive test suite can be understood as a second model for the system alongside the actual system specification. This model contains only example descriptions of the system which are concealed very implicitly in the test case; however, the model has the invaluable advantage of executability.
- A failed test can be viewed as an error description that documents the symptoms of the error. Users of an interface can thus demonstrate errors to those implementing the interface and can check very easily that the error has been corrected.
- Automated tests are indispensable for repeatedly checking the behavior of a system that is evolving. In this situation, interactive *regression tests* would increase the repeated test effort to such an extent that a lot of personnel resources would be tied up with the tests over the course of the project.
- According to [Fow99] and the author's own experience, it is helpful to approach unknown and in particular, untested code by checking the expected behavior of the system (the details of which may not yet be clear) in the form of new test cases based on a code review. The understanding of the code that arises as a result is more intensive and a side effect is that (further) tests exist afterwards.
- Ultimately, an extensive test suite is also documentation for customers—they may not normally have the capacity or the knowledge to understand the test cases but can understand the success reports from the tests. This allows other developers to subsequently improve and extend the system.

Tests are developed *based on goals*. Tests for individual methods, classes, and small subsystems serve primarily to *detect errors* and to eliminate and thus document those errors. In contrast, the goal of integration tests and system tests is primarily to demonstrate that errors are (largely) absent and that the behavior of the implemented system matches the predefined descriptions/specifications.

6.1.3 Error Categories

We can differentiate between a number of error categories in a software system. Fig. 6.4 gives definitions for the most important categories.

Failure: is the manifested inability of a system or component to perform a required function within specified limits.
Fault: is missing or incorrect code.
Error: is a human action that produces a software fault.
Omission: is a required capability that is not present in an implementation.
Surprise: is code that does not support any required capability.

Fig. 6.4. Terminology definitions for errors according to [Bin99]

A *fault* in the software is expressed by the fact that executing the faulty code can lead to a *failure* of the software system. Performing a test allows us to detect the failure of the software in the test situation and to conclude that there is a fault in the software. A failure can be the result of a combination of faults; one fault can also lead to different forms of failures. This means that some detective work may be necessary to localize the fault after a failure is identified. One way of doing this is to perform *debugging* with manual tracking of individual steps in the system. A better solution, however, is to define tests at every level of the system so that a fault is detected with even the smallest possible system configuration. We can then define further tests to localize a fault specifically.

The term *bug* is often used as a generic term for failures and errors[2,3]. In this book the term *error* is used as a synonym for *bug* and generally refers to the failure of the system caused by a user action, an interaction with the system environment, or a test.

Omissions and *surprises* cannot be detected by automated tests. Detecting omissions requires formal or informal descriptions of the required functionality that are then used for a static analysis or a comparative review. For example, omissions are discovered in form of noncompilable programs or by acceptance tests. In contrast, surprises are less problematic. Although they lead to unnecessary additional effort during system development, they do not disrupt the essential functionality. Static analyses can detect unreachable code in a method, for example.

[2] The term *bug* originates from [Hop81], which describes how a moth was trapped in a relay, and this resulted in the search for errors being identified with the search for *bugs*.

[3] Although the term *bug* is widely used, it is slang; in this book, therefore, the term *error* is used.

6.1.4 Terminology Definitions for Test Procedures

In the context of testing, there are a number of further terms that have already been used in this chapter and that require further explanation. Fig. 6.5 describes the main terms in brief.

Validation: Used to check whether the system fulfills the user's requirements (according to [Boe81]). This is done, for example, by means of prototyping during the project and acceptance tests at the end of the project.
Verification: Used to prove that the implemented system fulfills the formal specification and is therefore correct (according to [Boe81]).
System being tested: Also referred to as *system under test* and *test object*.
Test procedure: An approach for creating and executing tests. The theory of testing provides a larger variety of test procedures, differing, e.g., in the identification of appropriate sets of test data.
Test data: The test data consists of a specific set of values for a test that also contains the object structure including the objects to be tested.
Expected test result: The *expected result* of a test. This can be given explicitly by a dataset or implicitly by a test predicate—for example, as a comparison with the result of a test oracle.
Test case: Consists of a description of the state of the system to be tested and the environment before the test, the test data, and the expected test result.
Test suite: A set of test cases.
Test run: The execution of a test including the *actual test results*. A *test driver* organizes the execution, from the construction of the test data to the check of the test success.
Test success: A test is successful if the actual result matches the expected result. Otherwise, the test has *failed*.
Test result/verdict: A binary result indicating whether the test was successful or failed.

Fig. 6.5. Terminology definitions for tests

[Bin99] uses the term "*test point*" for a set of test data, and this captures the exemplary, selective nature of a test very well. The TTCN standard [ISO92] differentiates between further test results. In addition to the test success ("pass"), there are two types of failure ("failure", "error") and the alternatives "inconclusive" and "none", indicating that the test goal could not be tested. In the TTCN standard, the test result is also referred to as the "*verdict*". In Java, the test result "error" can be interpreted as the termination of a test object by an exception, for example.

The fact that a test *failure* means that the test and implementation are not conform or that an unexpected exception occurred during the test is extremely important. The test or the implementation is thus faulty and this must be clarified. [Mye79] points out that a *failed* test in this sense has fulfilled

its purpose, namely the *detection of errors*, and can therefore be evaluated as a success for the test activity.

6.1.5 Finding Suitable Test Data

One of the main problems associated with testing is the efficient development of a systematic test suite that *covers* all *important* situations. If test data is given for a test case, the expected result has usually been derived from the specification or defined by the developer or user. Specifying an expected result can involve a lot of effort and is prone to error. However, the product system and the test suite test each other, meaning that errors in the test cases can also be detected and eliminated.

Therefore, the main difficulty with testing lies in identifying suitable test data and defining how many test cases are sufficient for an adequate test suite.

[Mye79] described heuristics and procedures for developing sets of test data. These heuristics and procedures are discussed in more detail in [Lig90] and [Bei04]. They include procedures based on the *control flow* of the implementation which *cover* all statements, branches, condition variations, or paths within a method according to certain criteria. Other testing procedures additionally identify equivalence classes of test data as well as border cases.

Data flow-oriented test procedures use access to attributes and variables to develop test data. The prerequisite for both control flow-oriented and data flow-oriented procedures is that the implementation of the system must be known. Another factor these procedures have in common is that the test cases are created based on the analysis of the implementation.

This contrasts with the class of *functional* or *specification-based tests*; these tests are based on the specification. They check the functional properties of a system. The tests therefore detect conformance errors in the system and demonstrate whether the system matches the specified functionality.

We use metrics to determine the quality of a test suite. The metrics measure *test coverage* according to different criteria. However, even test coverage that is complete according to these metrics does not guarantee that the system is correct. In practice, testing is often based on experience and usually described by *test patterns*, [Bin99], checklists [PKS02], and *best practices*. We can see from the pragmatic test patterns that elements of test theory are still retained but without a dogmatic 100% compliance requirement. For example, the Extreme Programming approach intends statement coverage as a minimum goal and, where possible, a minimum path coverage is desired.

6.1.6 Language-Specific Sources for Errors

A system can be *robust* even if there is no specification that it does *conform* to. *Robust* implementations avoid abnormal crashes (exceptions), for example, as a result of uninitialized attributes, references to objects that do not exist,

and other similar problems. A typical source for problems with robustness are language-specific deficiencies.

C++ is a prime example of a language that contains an extraordinary number of error sources. This is due to the high number of unsecured C++ constructs. Memory management (which is the responsibility of the developer), pointer arithmetics, and unchecked accesses to fields are just some of the possible error sources.

Although syntactically similar to C++, Java is significantly more robust concerning these types of errors. In Java, many error sources have been eliminated with restrictive context conditions in the language and thus are detected by static analyses. For example, a sophisticated data flow analysis [GJSB05] can check whether values have been assigned to variables before the variables are used. The extension of ESC/Java [RLNS00] with assertions allows an even more extensive static analysis but requires, however, a more detailed description of assertions in the Java code. Nevertheless, this can reduce errors even further.

Other kinds of errors are in Java detected by runtime checks and reported as exceptions. These include the exceeding of array limits, division by 0, or illegal casts. It is therefore relatively easy to design a robust program in Java. However, the use of exceptions should be limited as far as possible, as exception processing has characteristics of the `goto` statement and can easily lead to complex code. In general, only external error sources—such as a missing file, a database that cannot be reached, or a broken Internet connection—should be handed with exceptions. Internal error sources, such as division by 0 or incorrect array limits, should be avoided by explicit handling.

Regardless of the technique to detect such errors, a robust treatment of the error and corresponding tests are necessary to demonstrate that the error is being handled correctly. This is also why the call hierarchy should not contain to many exceptions to be followed.

Although Java was designed to be much more safe than C++, it does still contain a number of error sources. We can prevent many of these error sources with restrictive programming. In Java, for example, an attribute of the superclass should not be hidden by the subclass containing an attribute of the same name.

Object-oriented programs have a much higher degree of complexity than procedural languages had. This is due to their high level of dynamics, the inheritance hierarchy, and the dynamic binding of methods. Object-oriented methods are usually much smaller compared to traditional procedures and they interact more intensely with other methods. This gives rise, for example, to the flexibility which is so important in frameworks: the flexibility is created by the adaptation of methods in subclasses [FPR01, FSJ99]. However, this possibility to redefine methods requires additional effort in the test. In particular, in this situation it is no longer sufficient to test all possible control flows within one method only; instead, all potential constellations of collaboration between methods in all subclasses must be checked. The size of the

task multiplies with the number of collaborating objects (each of which can originate from one of multiple subclasses). It is therefore often no longer possible to develop tests that cover all combinations of method calls and object structures.

As UML/P uses Java as a target language, the problems that exist in Java can generally also be found in UML/P. One exception is the afore-mentioned hiding of attributes that cannot be represented in UML due to its context conditions. Therefore, this problem does not arise when code is generated according to Java.

Where UML diagrams are used constructively to generate code, there are a number of ways in which the resulting program can violate the given specification. Depending on the form of the generation and the concepts implemented, certain errors of the underlying language Java are avoided or new problems are introduced. For example, according to the transformation specified in Section 5.1.3, restrictive multiplicities of associations are not normally implemented constructively. This can lead to a violation of the invariants which can only be prevented by the environment of the association respecting the invariant. We have to check this in tests. Similarly, state invariants and postconditions in Statecharts are not necessarily ensured constructively and must therefore be tested. On the other hand, as shown in Section 5.1.3, generating suitable functionality can ensure, for example, that a bidirectional association is always consistent and tests are then no longer necessary.

The definition of language-specific tests that can detect errors in robustness is therefore essentially dependent on the form of the code generated. Since it must be possible to parameterize the main elements of the code generator, it is difficult to predict which language-specific tests are necessary for UML/P. For points at which a generator does not prevent the violation of invariants, it is therefore helpful to define suitable test cases for checking these invariants.

6.1.7 UML/P as the Test and Implementation Language

As discussed in Section 4.1, UML/P can adopt numerous roles in the software development process. It is suitable as both the implementation language and as a language for defining tests. It thus takes over similar tasks to those performed by the respective programming language used in Extreme Programming projects. In these projects, the tests and implementation are also formulated in the same language. Furthermore, experiences with UML as the language for modeling tests show that it improves the efficiency of the developers [BMJ01, BPR04]. Nevertheless, it is important to use UML in a form in which it can be tested [BL01, Rum03]. *Testability* generally means the ability to derive tests from the model or—ideally—to generate them automatically.

If we use UML/P as the implementation language and want to develop tests systematically, we have to know which language-specific and typical

object-oriented problems UML/P and the used generator solves and which they cause. A typical problem of implementing bidirectional associations is ensuring the consistency between the attributes on both sides that store the association. If code is generated from a class diagram that may no longer be modified manually, this consistency can be *ensured constructively* by the code described in Section 5.1.3. This means that we no longer have to check the consistency by means of tests.

On the other hand, using an attribute of the referenced object as a qualifier in a qualified association introduces redundancy which can lead to inconsistency when the attribute value is changed. A static analysis of the code can determine whether this attribute is actually changed. However, if it is changed, dynamic tests are necessary to determine whether this change violates the consistency for a qualified association.

Therefore, to check the robustness of implementations formulated in UML/P, we have to examine the notations of UML/P and how they are used for code generation and analyze them critically for possible error sources. Note that UML/P consists not only of multiple types of diagrams and OCL, but also allows Java code explicitly as method bodies and as procedural actions in Statecharts. As already mentioned, this means that many of the typical error sources for Java are retained. However, the potential error sources of the implementation language UML/P depend primarily on the specific implementation by the parameterized code generator and therefore cannot be discussed in general terms.

As shown in the associations example above, UML/P needs to be examined for potential errors. We have five main strategies:

1. A *static analysis* of UML/P models can clarify whether a generated element or instances of this element can be manipulated in a way that is not permitted. Such manipulation can either be forbidden in the form of a context condition or communicated by means of warnings. In UML/P this applies, for example, for manipulations of attributes or associations labeled with ≪frozen≫.
2. The code generator adds a *runtime check* which, does not prevent an attempted forbidden manipulation, but detects such a manipulation and, for example, issues an exception. Java does this in the event of illegal array accesses. UML/P can adopt this concept for qualified associations, for example. However, these exceptions must be communicated to the developer as part of the API discussed in Section 4.2.2.
3. The code generator designs not only the code for implementing a UML/P construct but also *test code* which checks at runtime, whether a property is respected. For example, the restricted association multiplicity can be ensured by *checking an invariant*. This checking the invariant is similar to the runtime check described above. However, it only exists only in instrumented product code and it does not throw an exception if violated, but instead reports the failure of a test into a log.

4. The code generator uses a model as specification of the expected behavior and from this model, extracts test cases according to a coverage criterion. Certain forms of Statecharts, for example, are suitable for generating test cases that cover states, transitions, or paths.
5. The developer designs further tests himself to test properties of the code generated from a UML/P construct.

The last point is not actually necessary if the code generator works correctly. However, a code generator is parameterized, as described in Chapter 4. Thus scripts can control code generation flexibly. This may result in the generation of code which, for example, does not ensure the consistency of the bidirectional association with suitable measures. This means that tests for such properties typically check the code generator and its scripts for correctness and are therefore justified.

Therefore, we have to decide for each individual project which parts of the system should be tested and how intensive the testing should be. The project goals, the level of quality to be achieved, and the form of use of the product of course also influence the decision.

Section 4.1.2 discusses, from the point of view of code generation, which parts of UML/P can be used constructively or for tests. However, object diagrams that cannot be transformed directly into constructive code, OCL constraints, and Statecharts do not necessarily form *test models* that the code generator can use to perform tests.[4] If the afore-mentioned diagrams are initially used in the development process only for communication between developers or for abstract representation, they normally have to be enriched with more details so that they can be used as *test models* suitable for generating tests. Chapters 4 and 5 discussed the transformation of individual diagrams and OCL constraints into test code in detail. Therefore, we discuss in the following the methodological use of combinations of UML/P artifacts to create test cases for conformance tests.

Fig. 4.3 illustrates the typical use of UML/P diagrams for tests and implementation. While the product code represents a complete, independent system, the test code can only be executed in combination with the product code. The product code is furthermore *instrumented* for use in tests. This means that additional pieces of code are injected into the product code so that the test code can also access all necessary information during the test process. These additional pieces of code include, for example, special functions for assigning values to and reading encapsulated attributes if the corresponding `get` and `set` functions are not available or have additional effects such as, for example, for maintaining the consistency between multiple attributes. We also

[4] The term "test model" was introduced in Fig. 4.1. Due to the aspect of generation from UML models, in that diagram it is defined more narrowly than in the Rational Unified Process [Kru03], for example—there, a manual conversion into scripts and test drivers is proposed, with the scripts and test drivers also counting as part of the test model.

insert code that checks invariants and OCL method specifications at runtime and code that enables the logging of method calls for comparison with predefined orders of calls. When programming in conventional Java, we have to develop test monitors or adapters for such tasks [Wil01], although these have only limited access to the test objects.

The instrumentation must not affect the functional behavior of the product code, which is why the use of modifying methods in OCL constraints is forbidden. This is the only way to guarantee that the product code that is approved for release but is not instrumented has the same behavior as the code that has been tested.

It may be necessary to instrument the same product code differently for different tests. For example, the order of calls is irrelevant for only some tests. Testing all OCL invariants in all tests can also be very inefficient: tests become impractical if it takes too long to execute them. Therefore, the instrumentation must either be individualized dependent on the test currently being executed or parameterized through Boolean flags at runtime. The latter has the disadvantage that the instrumented code can become very large; however, it has the advantage that no repeated code generation and transformation of the generated code is necessary. The test driver can set the flags directly or they can be set by stereotypes in the test description. Due to the constantly increasing power of computers and the efficiency of good compilers, we can assume that the code instrumentation for test purposes and the simulation of the environment and distribution that are needed for efficient testing, as well as the dynamic check of invariants, preconditions, and postconditions in the product code are practicably realizable.

The instrumented product code and the test code test each other. If a test case has failed, the test object or even the test case itself may be faulty. In practice, and as observed in the auction project, it is much more frequently the case that the test cases are faulty, but the product is ok—for example, because a change in the functionality of a method was not implemented appropriately in the test case. The situation becomes awkward if both the test object and the test case are consistently faulty and a test success is wrongly reported. This problem occurs in particular when a developer designs both the product code and the test and in doing so, repeats an error in his logical considerations. In the Extreme Programming approach, this is countered, for example, by two developers working on the code together at the same time. In addition, tests of the layers above or the integration levels should still be able to detect such an error.

6.1.8 A Notation for Defining Test Cases

There also exist languages used specifically for the specification of tests. For example, the telecommunications industry uses TTCN [ISO92, GS02] in combination with MSCs [IT11, Krü00] and SDL [IT07b]. In this domain the use of a special test notation is regarded as an advantage compared to the use of

a general purpose programming language for defining test drivers, because it allows more abstract and thus clearer test specifications. With the JUnit framework [JUn11] support by a well-designed library of auxiliary functions is available and can relatively easily be extended. For an abstract test notation some investment is necessary to develop a test tool for interpreting the test notation to drive software and hardware components as well as to train the developers in that notation. Thus, both approaches have their merits. However, using the same programming language for tests and implementation has more advantages compared to using a separate test notation:

- The effort involved in learning a test notation should not be overlooked, but only applies if the test language is different from the realization language.
- The expressiveness of test notations is typically limited. This results in an inability to formulate certain tests or the test notation being extended ad hoc. The latter is not possible, for example, if the tool used cannot be modified. If it is possible, it involves an enormous amount of additional effort that is usually significantly lower if a corresponding framework is extended instead.
- The best situation for integration between a test notation and an implementation language is given when both are identical or the test language builds on the other. Otherwise, the concepts of the implementation language (for example, attributes or method calls) must be made available in a suitable way in the test notation so that they are accessible in tests.

It is not surprising, therefore, that in many software development projects, and in particular those that use agile methods, the same notation is used for realizing tests and for programming. In projects that use UML/P—that is, a combination of modeling techniques from UML and from Java—it is therefore an advantage to also use UML/P to model tests:

- UML/P provides an abstract notation for modeling tests.
- The combination of UML with Java allows us to still describe special cases of tests that cannot be formulated directly in UML within an integrated framework.
- The afore-mentioned effort involved in learning a new test notation does not arise or is reduced to understanding how UML/P, which is already used for system development, can also be used to model tests.
- The mental obstacle to developing tests in a new notation does not apply.
- There is no conceptual breach between the test notation and the modeling or implementation language.
- No additional notational or technical knowledge is required to model tests, meaning that developers are generally capable of defining tests themselves.

UML/P therefore combines the advantages of an abstract notation for test cases with the good integration of the test and implementation language.

This integrated use of UML/P means that it is actually realistic to create a sufficient test suite in a short time parallel to the system development. It is therefore not surprising that Java is being used increasingly in parallel to test notations. This is the case not only for the development of business software, but also for the development of embedded systems, such as telecommunications systems.[5]

As a test usually consists of multiple UML diagrams, some additional syntax is necessary to define tests compactly. The proposal presented here is restricted to the referencing of UML diagrams, OCL constraints, and the integration of Java code in order to formulate parts of the test. Fig. 6.6 shows the complete pattern for defining tests. Any unused parts can be omitted.

```
                                                                     Test
test Test object, e.g., method or class e.g., Auction.bid {
  name:          Generation target for the test, e.g., AuctionTest.testBid
  testdata:      Object diagrams prepare the set of test data
  tune:          Java code allows individual, additional adjustment of the
                 test data
  driver:        Java method call(s) | sequence diagram
  methodspec:    OCL method specifications are checked for a method test
  interaction:   Sequence diagrams are checked as sequence descriptions
  oracle:        Java method call | Statechart produces comparable oracle results
  comparator:    Java code | OCL code compares test results with oracle results
                 The default is a match for structure and attribute contents
  statechart:    Statechart for object name from initial state
                 to { final states }
                 The test has to trigger transitions in the specified object,
                 from the initial state to one of the final states
  assert:        Object diagrams | OCL constraints | Java test code
                 Boolean conditions for the test result
  cleanup:       Java code cleans up resources used
}
```

Fig. 6.6. Template for defining a test

The individual components are discussed later on in this chapter. The discussion uses a table-like variant of this pattern that is suitable for defining multiple tests clearly.

[5] For example, Ericsson used Java to model function tests in the mobile communications sector and TTCN [ISO92] for protocol validation only already in 2004.

6.2 Defining Test Cases

6.2.1 Implementing a Test Case Operatively

According to the definition, a test case consists of a set of *test data*, a description of the *initial state of the test object and its environment*, and the *expected result* of the test. In order to implement a test case operatively, we have to realize an executable *test driver*. The task of a test driver is to set up the test object, including the test data, and to insert it in a test environment so that the test object can run as if it were in the product system. At the same time, the data of the actual result that is relevant for the comparison with the expected result is stored. To do this, the test driver uses an instrumentation of the test object that gives it the required access to the data of the test object and the *dummy* objects described in Section 8.1 in order to simulate the environment of the test object. Side effects, such as database access, screen output, or external communication are captured in dummies and logged if necessary.

Fig. 6.7 demonstrates a typical approach for simple test drivers. This approach is refined in Section 7.4: a method sequence is executed instead of just one method and the order of the internal sequences is observed.

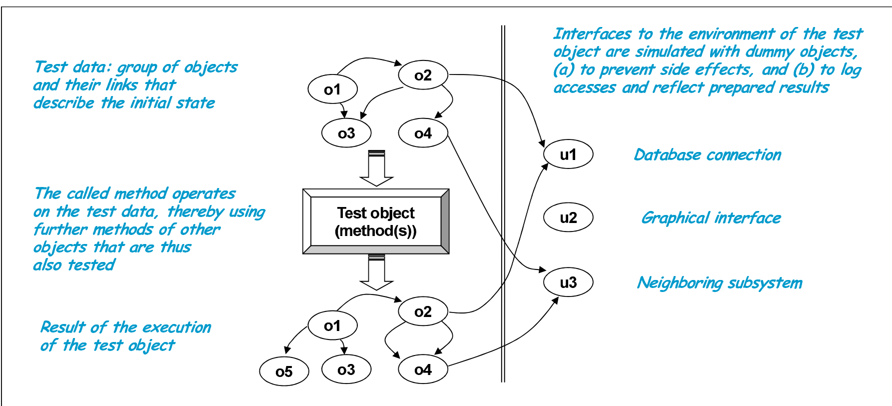


Fig. 6.7. Structure of a simple test driver

Fig. 6.7 shows an object structure with four objects that is handed over to the called method as a set of test data. The called method itself is usually attached to one of these objects (for example `o1`). The other objects, `o2` to `o4`, are necessary because the called method access these objects and may change them. Further objects (`u1`, `u2`, and `u3`) are necessary to allow access to the environment. The environment is replaced by a simulation with the *dummies* discussed in Section 8.1. The test object and the simulated test environment are distinguished differently depending on the test goal. In a method test, the test object actually consists only of the called method. Therefore, the direct

environment can be replaced with dummies and, if necessary, simulate other methods of the same class. In class tests, the objects of the environment are also typically replaced by dummies. In contrast, for integration and system tests, as little as possible is replaced, often only the real system boundary.

Implementing a test case operatively requires a test driver that is normally realized in a test method. A test driver consists of three main phases:

1. The object structure required for the test, including the dummies for the environment, is set up.
2. The test object is executed. Often, only one single method is called.
3. The actual result obtained is compared with the expected result. The actual result consists, as shown in Fig. 6.7, of the resulting object structure, any access logs to the simulated environment found in the dummies, and a return value for the method tested.

If a file or an Internet connection has to be opened for a test case, or a database has to be adapted, etc., at the end of the test these have to be cleaned up. In C++ this includes, for example, releasing any object structures created.

The Java framework *JUnit* [JUn11, BG98, BG99, HL02], which is now available for a large number of programming languages, offers excellent support for defining tests. JUnit offers an infrastructure and methodological guidelines for defining test data and for executing tests in the form shown in Fig. 6.7.

In addition to internal test drivers, there are approaches for defining test drivers as scripts outside the actual system. In that situation, the test data and results are typically stored in files. The comparison of the expected and actual results is reduced to a file comparison. Based on today's technology, the use of external test data and test drivers primarily for file-based systems, such as compilers, generators, or XML transformers, is a good supplement for internal test cases but is only suitable for system tests. [Den91] proposes a variant for a test system that only stores the results in files or a database externally and compares them there, while the test drivers are defined in the system programming language.

6.2.2 Comparing Test Results

Design Patterns for Comparison

Creating test data is a rather complex process, quite like the comparison of expected and actual results. Providing the developer with adequate support for analyzing the actual result of a test by means of comparison with an explicitly given expected result, verification of an invariant, or similar techniques has a significant effect on the efficiency of developing tests. On the one hand, it must be possible to formulate this comparison as quickly and compactly as possible; on the other hand, it is equally important that the comparison has a certain level of resilience against changes to elements of

the system that are used in the test but are not actually part of the test. We can use equivalence comparisons or abstractions here. Therefore, specific design patterns are useful for selecting the correct form of comparison in each case. We call them in short *comparison patterns*. Fig. 6.8 classifies possible comparisons with mathematical means. These techniques, which are similar in theory, have very different effects in the practical implementation discussed below.

We assume that A and B are the sets of object structures handled (one object structure generally contains multiple objects). A method f is understood as function $f : A \rightarrow B$. P represents a predicate. For simplification, A and B contain method parameters and results as well as read and modified attributes and objects.
We can identify the following comparison patterns and thus represent the expected result explicitly or implicitly, whereby in some cases, a pattern specializes its predecessor. Let us assume $x \in A$:

Difference comparison $P(x, f(x))$: The before/after comparison describes the comparison between the source structure x and the actual test result $f(x)$ based on a comparison predicate P.

Property check $P(f(x))$: Checks a property that is not directly dependent on the source structure. This includes invariants but also specific tests of individual attributes and attribute combinations of the actual result.

Equivalence comparison $f(x) \equiv y$ with explicitly predefined $y \in B$ and an equivalence $\equiv$ that compares only relevant aspects. An equivalence that can be realized by simple means is the structural and value-based equality.

Comparison after abstraction $Ab(f(x)) = Ab(y)$ or $Ab(f(x)) = z$ is a variant of the equivalence test with an explicitly represented abstraction $Ab : B \rightarrow Z$ and $z \in Z$. A simple example of an abstraction is the selective comparison of attributes of an object that is occasionally implemented in the method `equal`, for example.

Identity check $f^{-1}(f(x)) = x$, whereby it must be possible to invert f in order to restore the original state.

Check with oracle function $g(x) = f(x)$, in which a second realization g exists and is independent of f, but implements the same functionality.

These comparison techniques can be combined. If, for example, an oracle function produces only an equivalent but not an identical result, a comparison in the form $g(x) \equiv f(x)$ with a suitable equivalence is useful.

Fig. 6.8. Design patterns for comparison in a test

Advantages and Disadvantages of Comparison Patterns

The *difference comparison* $P(x, f(x))$ can check desired modifications and detect undesired modifications. Due to the parameterization of P, we can apply

the comparison to various initial datasets x in which f should exhibit the desired behavior. One example is checking whether a counter was increased. OCL method specifications are a suitable means for describing such comparisons.

The *property check* determines, for example, whether invariants have been respected or whether certain values have been assigned to certain attributes. In most cases, the actual result is checked only selectively and thus an *abstraction* is performed. This has the advantage that if the system is changed at a point that is not relevant for the test case, the test case is still successful. As a special case of the property check, the *equivalence comparison* uses an explicit representation y of the expected result. In a comparison, we can use an appropriate *equivalence* and thus, for example, omit the check of irrelevant attributes or ignore the order in a container. Using an *abstraction* is mathematically equivalent but is implemented differently.[6]

The *identity check* is only possible in very limited circumstances if (a) the function can be inverted and (b) the inverse function f^{-1} can be realized with a reasonable level of effort. Ideally, the inverse function f^{-1} is also present in the system to be tested and has already been checked through other test cases.

An alternative implementation—also referred to as the *oracle function*—is useful, for example, if we want to replace an existing implementation with an improved version in a refactoring. This, for example, works nicely when replacing a sorting algorithm. Another option is to use an executable specification as an oracle provided the specification has not already been used for the implementation. Statecharts are suitable in this case, for example. If the result received using the oracle function deviates from the test result in irrelevant details, we can specify an explicit comparison function that checks for an equivalence instead of equality.

Requirements for Operative Implementation

From the mathematical characterization of the comparison patterns shown in Fig. 6.8, we can derive information about which functionality a framework should provide to support an operative implementation. The framework functionality is understood as part of the UML runtime environment according to Fig. 4.6.

For the *difference comparison* $P(x, f(x))$, after the test execution the source structure x and the actual result $f(x)$ must be provided simultaneously. Therefore, we have to create a *clone* of the initial data before executing f. If a comparison fails, a useful *output* of the actual result $f(x)$ is required to be able to analyze the error. Both the cloning functionality and the output must access the underlying object network.

[6] An abstraction Ab defines an equivalence through $x \equiv y \Leftrightarrow Ab(x) = Ab(y)$.

The *equivalence comparison* $f(x) \equiv y$ requires an implementation of the comparison operator. We cannot always use the comparison operators == and `equals()` provided by Java. For objects, the operator == compares only the object identity and therefore we cannot use it for content comparisons. The developer can redefine `equals()` for each class but this operator is already used in the product system—some of the container structures use the `equals()` operator, for example. Therefore, the functionality of this operator desired in the product system can be incompatible with the behavior in a test comparison. Furthermore, different individual comparisons $\equiv$ may be necessary for different tests. For example, it may be necessary to compare *vector* objects as realizations of sequences or sets, that is, with or without respecting the order; alternatively, only a subset of the attributes may be relevant for a comparison of objects in an application class. Therefore, support for a flexible definition of comparisons for object structures is desirable, with these comparisons being able to compare recursive object structures or object structures subject to cycles. If the test fails, the recursive output of the actual result $f(x)$ should also be marked with the differences to the expected result y.

Abstractions in the form $Ab(f(x)) = Ab(y)$ and $Ab(f(x)) = z$ used before the comparison include, for example, the selection of individual attributes, an individual object, or a substructure. The definition of a comparison operation described above allows us to perform the abstraction implicitly without calculating the abstracted data structure explicitly. It is therefore an efficient way of coding an abstraction intended for a comparison. In some cases, it is useful instead to delete attribute values and links that are not to be compared, or to calculate a transformation into a *normal form*. This allows us, for example, to first sort two sequences that are to be interpreted as sets and then compare them element by element.

The last two variants, the *identity check* $f^{-1}(f(x)) = x$ and the *check using an oracle function* $g(x) = f(x)$, are based on the existence of corresponding functionality. The function g represents a form of oracle. Oracles are discussed in more detail in [Bin99], which states, correctly, that a perfect oracle is not possible. At the same time, however, it provides some patterns for realizing oracles.

6.2.3 The JUnit tool

From the previous discussion in this section, we can see that it is important to provide suitable functionality to support the definition of tests. This functionality can be provided by tools such as generators or analyzers or by a framework. Ideally, it is provided by a combination of both: a generator generates test code that works together with the framework. The framework can then be regarded as a component of the UML/P runtime environment. The `JUnit` [JUn11, BG98, BG99, HL02] framework described below has a lot of the functionality required for this purpose.

Fig. 6.9 shows an excerpt of a test of the method `bid` of the class `Auction` formulated in Java and performed with JUnit.

```
public class AuctionTest {                                          Java
  @Test
  public void testBidSimple() {
    // (1) Setup of the object structure
    Auction auction = new Auction(...);
    Person  person  = new Person(...);
    Time    time    = new Time("14:42:22", "Feb 21 2000");
    Money   money   = new Money("552000", "$US", 2);
    // Further objects are required to complete the structure

    // (2) Execution of the test
    boolean result  = auction.bid(person,time,money);

    // (3) Check of results
    // result==true: Bid was successful
    assertTrue(result);

    // The bid submitted is currently the best bid
    assertEquals(money, auction.getBestBid());

    // The end of the auction was defined as time + extensiontime
    Time expectedClosingTime =
           new Time("14:45:22", "Feb 21 2000");
    assertEquals(expectedClosingTime,
                 auction.getCurrentClosingTime());
  }
  ... // E.g. constructor
}
```

Fig. 6.9. Test driver for the method `bid` in JUnit

In a setup phase (1), all objects required for the test are created. These can be complex object structures (as is the case with `Auction` and `Person`) that are also connected to one another. Therefore, this first phase is often outsourced to a separate method with the name `setUp`. In this example, the number of objects required is already relatively high. To achieve the greatest possible efficiency in the test run, according to [LF02] small object structures should be used as far as possible.

The execution of the test (2) consists of a simple method call. The results are then compared (3), whereby the methods `assert`*X* can be used to define and to report the success or failure[7] of the test. These methods are available

[7] JUnit uses the term "failure" for the failure of a test and "error" for an abnormal termination of the test execution by an exception.

as static methods from the JUnit framework or alternatively inherited from a given class `TestCase`.

Some of the strengths of JUnit lie in the management and combination of test suites. This is because JUnit provides simple mechanisms for creating *test suites* and for executing such suites individually. JUnit is a cleverly implemented framework that is demonstrated only incompletely in this example. It has only a few classes and a small API that the user has to understand. It can thus be used to define test cases effectively through clever formation of subclasses and interface implementation. JUnit is easy to handle because the implementation language and the language for defining test cases are identical. Therefore, the same development environment can be used. JUnit is a successful example of a development tool that is realized as a framework in the implementation language itself and therefore easily and elegantly creates a close connection between the tool and the implementation.

7

Model-Based Tests

Testing is an extremely creative
and intellectually challenging task.

Glenford J. Myers, [Mye01]

Building on Chapter 6, this chapter focuses on practical issues associated with implementing tests with UML. It demonstrates how we can define test cases efficiently using UML/P and which types of diagrams are suitable for doing so.

7.1 Test Data and Expected Results using Object Diagrams 186
7.2 Invariants as Code Instrumentations 189
7.3 Method Specifications 191
7.4 Sequence Diagrams 195
7.5 Statecharts 202
7.6 Summary and Open Issues Regarding Testing 212

© Springer International Publishing AG 2017
B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_7

First, the terminology and typical problems are discussed. Then, this chapter addresses several specific techniques for using UML/P notations to define and develop test cases.

The process of defining test cases in UML/P is based on the transformation of diagrams and OCL into pieces of code as described in Chapter 5. These pieces of code are then put together to create test cases and consistency checks. On this basis, the various sections in this chapter use examples to demonstrate how to model test cases with the following elements: object diagrams, Section 7.1; OCL invariants, Section 7.2; method specifications, Section 7.3; sequence diagrams, Section 7.4; and Statecharts, Section 7.5.

7.1 Test Data and Expected Results using Object Diagrams

Fig. 7.1 shows the use of two object diagrams to represent the test data and the expected result that is based on Fig. 6.7. To enable an object diagram to be used for multiple test cases, small adjustments in specific situations ma be necessary. Therefore, it is possible to add some adjusting Java code once the relevant object structure has been built up.

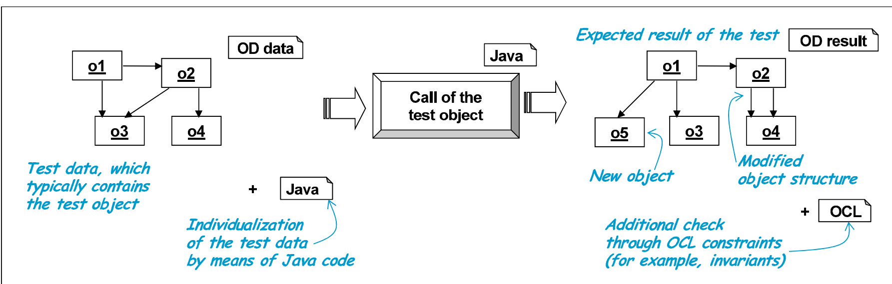


Fig. 7.1. Standard form of the test with object diagrams, OCL, and Java test driver

Once the test object has been executed, the actual resulting data structure is available. Afterwards, it is compared to the expected data structure specified by the second object diagram. Additional properties can be checked via OCL constraints. These can be OCL invariants that are generally valid and should always be checked, but can also be constraints specific to the test.

One of the typical sets of test data used in the auction system consists of the following: one object of the class `AllData`, several auctions with all dependent objects, several persons in different situations logged in, and a series of bids and other messages sent to some open and some completed auctions. In our project we needed just a handful of data records from these basic structures because by adapting them with additional Java code, we can reuse them for lots of different test cases.

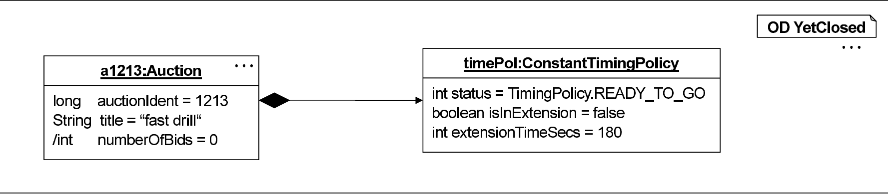


Fig. 7.2. Set of test data as an object diagram

With the set of test data represented in Fig. 7.2, the expected result (here an excerpt) in Fig. 7.3 describes the effect of the method for opening an auction (`start()`).

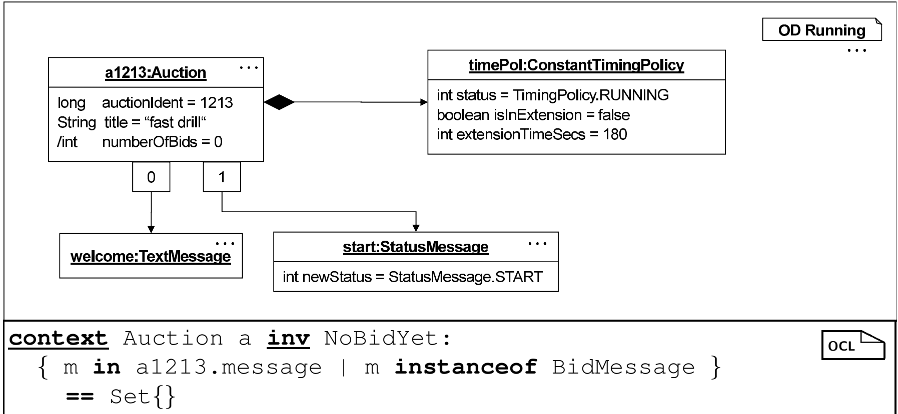


Fig. 7.3. Expected results as an object diagram

In addition to the OCL constraint, specified as `NoBidYet`, which states that no bid has been submitted since the auction was opened, there are additional invariants that have to be respected. `Bidders1`, for example, is valid for auctions generally:

```
context Auction a inv Bidders1:
  a.activeParticipants <= a.bidder.size
```

It must therefore also apply once an auction has been opened. If the diagrams described are based on code generation in accordance with Chapter 5, we can formulate a test with the Java/P extension described in Appendix B, Volume 1 as follows:

```
testStart() {
  Auction a1213 = setupYetClosed();        // Create test data
  a1213.start();                           // Execute test
```

```
  ocl isStructuredAsRunning(a1213);        // Expected result satisfied?
  ocl checkNoBidYet(a1213);                // Property NoBidYet
  ocl checkBidders1(a1213);                // Invariant Bidders1
  ocl checkTime1(a1213);                   // Further invariant Time1
}
```

UML/P offers a separate type of document that we can use to formulate these tests more compactly:

```
test Auction.start() {                                                 Test
  testdata: OD YetClosed;
  driver:   a1213.start();
  assert:   OD Running; inv NoBidYet; inv Bidders1; inv Time1;
}
```

For a good use of the JUnit framework capabilities, an adapted code generator for the Boolean query `isStructuredAs`, which is based on the object diagrams and `check`-implementations for all OCL constraints, is helpful. This method results not only in a Boolean value but, when an error occurs, also a description of the reason and the actual result like it is common for JUnit. This description ideally describes the name of the OCL constraint violated or the name and content of the objects and attributes that deviate from the expected result.

Fig. 7.3 presents the expected result in just as much detail as the test data. This does not always have to be the case. The object diagram used for the test data requires a certain level of completeness to create object structures constructively. In contrast the expected result only has to present the part of the object structure that is of interest for the test case. Individual attributes can be omitted but derived attributes can also be used. If a substructure of the objects is missing from the expected result, this does not mean (in terms of the semantics of an object diagram) that this substructure needs to be deleted in the actual result; it merely means that the current form of the substructure is of no interest.

Adding the stereotype «complete» from Table 5.15 indicates that the object diagram should be understood as a complete description of an object structure including all links. In this case, the comparison is significantly more restrictive, as described in Section 5.2. However, in this object diagram, all attribute values also have to be specified.

We can use the ability to combine object diagrams controlled by OCL logical operators, as discussed in Section 4.3, Volume 1, as an aid for modularization when modeling test data and expected results. According to the procedure discussed in Section 5.2, we can combine object diagrams and thus put the entire structure of the test data together from several individual diagrams. We can use the flexible ability to combine object diagrams using OCL operators, as discussed in Chapter 4, Volume 1, to define the expected result. For example, we can use negated object diagrams to check that a certain situation does not occur.

The procedure for modeling test cases with UML proposed here is also found in approaches such as [CCD02], where object diagrams are also used to model test data. There, multiple stereotypes are available allowing to remark the purpose in the object diagram directly. However, this reduces the ability to reuse the diagram for different purposes.

The use of object diagrams is demonstrated in more detail in Chapter 8 using test patterns.

7.2 Invariants as Code Instrumentations

One of the disadvantages of modeling test cases according to the style shown in Fig. 6.7 is the lack of opportunity to check invariants during the test run. If, for example, a complex algorithm is processed, it can be useful to formulate and check intermediate properties that apply instead of checking only the result and having to draw conclusions about internal intermediate states. For this purpose, Appendix B, Volume 1 introduced the already frequently used `ocl` statement with an OCL constraint as an assertion to be checked. This is supplemented by a `let` statement for defining local variables that can be used in later OCL invariants in order to access earlier states.[1]

The example in Fig. 7.4 demonstrates the use of `ocl` and `let` statements based on a method of the class `Auction` for sending messages.

```
class Auction {                                              Java/P
addMessage(Message m) {
  ocl !this.message.contains(m);

  let oldMessageSize = message.size;
  message.add(m);
  ocl message.size == oldMessageSize +1;

  for(Iterator<Person> ip = bidder.iterator();
                                   ip.hasNext(); ) {
    Person p = ip.next();
    p.addMessage(m);
    ocl MessagesDelivered(this,p);
  }
}}
```

Fig. 7.4. Assertions as `ocl` statements

[1] With the `assert` statement, available since version 1.4, Java offers a similar form for assertions. However, there is no analogy for the `let` statement that allows the introduction of local variables for test purposes only.

As shown in Fig. 7.4, an `ocl` statement comes with an OCL constraint, but could also ne a name of a named OCL invariant. The example refers to the following OCL invariant that describes the relation between an auction and the person receiving a message:

```
import Auction a, Person p inv MessagesDelivered:                OCL
  p in a.bidder implies
    forall m in a.message: m in p.message
```

According to Section 4.4, Volume 1, object diagrams are excellently suited for describing a state via a predicate. It follows, therefore, that we can also use object diagrams and the combination of object diagrams and OCL described in Section 4.3, Volume 1 as assertions.

We can also define local variables within loops. Although the variables introduced with `let` are not modifiable, they have the same visibility range as normal local Java variables and new values are therefore assigned to them with each iteration of the loop. This allows us, for example, to check the termination of the (not entirely trivial) loop given in Fig. 7.5.

```
int n = ...;                                                Java/P
while( n>0 ) {
  let nOld = n;
  if( n % 1 == 0 ) n = n/2; else n = n-1;
  // Use of n ...
  ocl nOld > n;          // Decreasing values
  ocl nOld > 0;          // Limitation by 0
}
ocl n<=0;                // Applicable after the loop
```

Fig. 7.5. Assertions that show the termination

To terminate the loop in Fig. 7.5, we use the constantly decreasing variable `n`, which has a lower limit of 0. The old value of this variable is buffered in `nOld`.

The addition of the two statements to Java is very much based on assertion logic that uses the same techniques to prove statements about programs. In fact, these technics allow us to do much more than merely support exemplary tests. A suitable verification calculus for Java, as discussed for example in [vO01] and [RWH01], can thus verify that invariants are always valid. However, in an object-oriented system, developing a sufficiently complete set of assertions in code to allow a verification tool to check correctness involves a great deal of effort. Nevertheless, such an approach can be interesting for specific tasks, such as complex algorithms, logs, or core elements of the security architecture.

It must be possible for the compiler to handle the instrumentation of the product code with checks for invariants flexibly: in live operation the instru-

mentation should not take place; in trial operation, an instrumentation invisible to the user but with an output in the background (log) must be possible; and in a test system, an instrumentation suitable for test cases, with output and error display via JUnit, should take place.

7.3 Method Specifications

One of the main application domains of OCL is describing the behavior of individual methods abstractly by specifying a precondition and a postcondition for the method. A method specification can act as a code instrumentation, as part of a test case, and as the starting point for deriving sets of test data.

7.3.1 Method Specification for Code Instrumentation

The pair `CC2pre/CC2post` is a typical example of a method specification. It is taken from Section 3.4.3, Volume 1 and its context is already described there:

OCL

```
context Person.changeCompany(String name)
pre  CC2pre:  company.name != name &&
              exists Company co: co.name == name
post CC2post:
  company.name           == name &&
  company.employees      == company.employees@pre     +1 &&
  company@pre.employees == company@pre.employees@pre -1
```

A typical transformation of this method specification is the instrumentation of the product code analog to the use of assertions described in Section 7.2. Fig. 7.6 shows a method body of the method `changeCompany` for one of the three cases discussed below. Here, the method body has been supplemented with `ocl` statements.

As discussed in Section 7.2, the `ocl` statements and their OCL arguments are transformed into JUnit-capable runtime checks. Instead of the existence quantification in the OCL constraint, which is a problem for the execution time, we can provide a more efficient version by redefining the method specification. Instead of using the existence quantification with the `let` construct, the `Company` object is defined directly.

7.3.2 Method Specifications for Determining Test Cases

Although the transformation described above uses the method specification to instrument the product code, it does not give rise to a test case or the operative transformation of a test case into a test driver. Developing tests from a pair consisting of a precondition and a postcondition is not generally

```
class Person {                                                    Java/P
  changeCompany(String name) {
    // pre CC2pre:
    ocl company.name != name &&
        exists Company co: co.name == name;

    // Method implementation
    Company oldCo = company;
    Company newCo = AllData.instance().getCompany(name);
    if(newCo==null) ... // Company does not exist

    company = newCo;
    newCo.employees++;
    oldCo.employees--;

    // post CC2post:
    ocl company.name == name &&
      company.employees     == company.employees@pre    +1 &&
      company@pre.employees == company@pre.employees@pre-1;
}}
```

Fig. 7.6. Instrumentation with precondition/postcondition

easy. However, for certain forms of methods, we can derive suitable sets of tests data from the structure of the specification.

Starting from the disjunctive normal form of the precondition, we can perform a partitioning that requires the definition of a test case for each satisfiable clause of the normal form. In [BW02a] and [BW02b], this was done for an example by means of transformation according to Isabelle/HOL [NPW02] in order to detect and eliminate unsatisfiable parts with a verification tool at least partially automatically. The example `changeCompany`, with its three specification parts (see Section 3.4.3, Volume 1), can also be understood as a disjunction. This method is described by three pairs of preconditions/postconditions that define three equivalence classes of inputs `CC1pre`, `CC2pre`, and `CC3pre`:

```
// List of preconditions as OCL parts                             OCL
context Person.changeCompany(String name)

pre  CC1pre:  !exists Company co: co.name == name

pre  CC2pre:  company.name != name &&
              exists Company co: co.name == name

pre  CC3pre:  company.name == name
```

These equivalence classes are pairwise disjoint and partition the entire possible input range. The disjunction of the three conditions produces:

```
CC1pre || CC2pre || CC3pre  <=>  true
```

Therefore, at least one test case should be provided for each of these three equivalence classes. Identifying equivalence classes for test data is a significant step towards developing tests systematically. Interesting test cases can be determined based on a manual or tool-supported analysis of the specification. Unfortunately, we have to accept that it is not easy to automate the generation of sets of test data (in this example, object structures in each of which one of the specified conditions is valid). For example, the constructive transformation of the existential quantifier `exists x: P`, (that is, the generation of code that creates an object `x` that satisfies the condition `P`) can only be resolved for special cases of `P`. Nevertheless, when analyzing a given test suite, it is helpful to receive corresponding information if one of the equivalence classes specified is not covered by tests.

As described in [Mye01], [Lig90], and [Bal98], we can refine the partitioning of the space for test data by analyzing the postconditions. This takes the following specification into account:

```
context int abs(int val)
pre:  true
post: if (val>=0) then result==val else result==-val
```

As the precondition is `true`, it cannot be used to partition the space for test data. In this case, however, an analysis of the postcondition can help as it allows us to easily identify two equivalence classes: `val>=0` and `val<0`.

A further possibility for defining test cases is the standard formation of equivalence classes and the consideration of borderline cases. This option is particularly suitable for the data types offered by the programming language. For integers, for example, test data from

```
Set{-n,-10,-2,-1,0,1,2,3,4,10,11,n+1}
```

for a large `n` within the value range can be used, as critical special cases are often found around `0`. We can define similar standards for container data structures, Boolean values, and floating-point numbers.

These standard cases are derived from the realization that these data types contain special values for which the implementation takes a different path compared to the neighboring values. The boundary value analysis procedure [Mye01, Bal98] explicitly restricts such value ranges and covers them with sets of test data on both sides of the boundary. The mathematical *abs* function has `0` as boundary and requires, e.g., `-1`, `0`, `+1` as test data. In most cases, however, defining the boundary values is more complex, as conditional expressions can place the parameters in relation.

In addition to developing black box tests from the specification, we should not forget to derive white box tests from a given implementation. Additional cases that may not be recognizable in the specification only become obvious when we analyze the implementation, in this case the Java

code bodies and transition actions in particular. Here we have to use classic techniques for test coverage [Bei95, Bal98].

In some cases, we cannot predict the set and type of test cases required from a specification because there are several different implementations. These include, for example, different sorting variants, such as merge sort, quick sort, or bubble sort, which each have very different functions and therefore require different sets of test data. In particular, tests for combinations require a lot of effort, especially, when bubble sort is used to presort small strings before merge sort is applied.

In general, however, it is important to develop solely test cases based on a code analysis and on the specification. This is because solely code-based tests are suitable primarily for ensuring robustness, while specification-based tests check that the implemented and the specified behavior match—that is done by checking conformity to the specification. If, for example, a specification case has been forgotten in the implementation (an *omission*), it cannot be discovered by code-based test cases.

However, we must be careful when using metrics schematically for test case coverage. On the one hand, important cases could be overlooked and on the other hand, a lot of potentially unnecessary additional effort might arise. Therefore, the optimal situation is a combination of different approaches for defining test cases that is adapted to the complexity of the test object and to the required level of system quality.

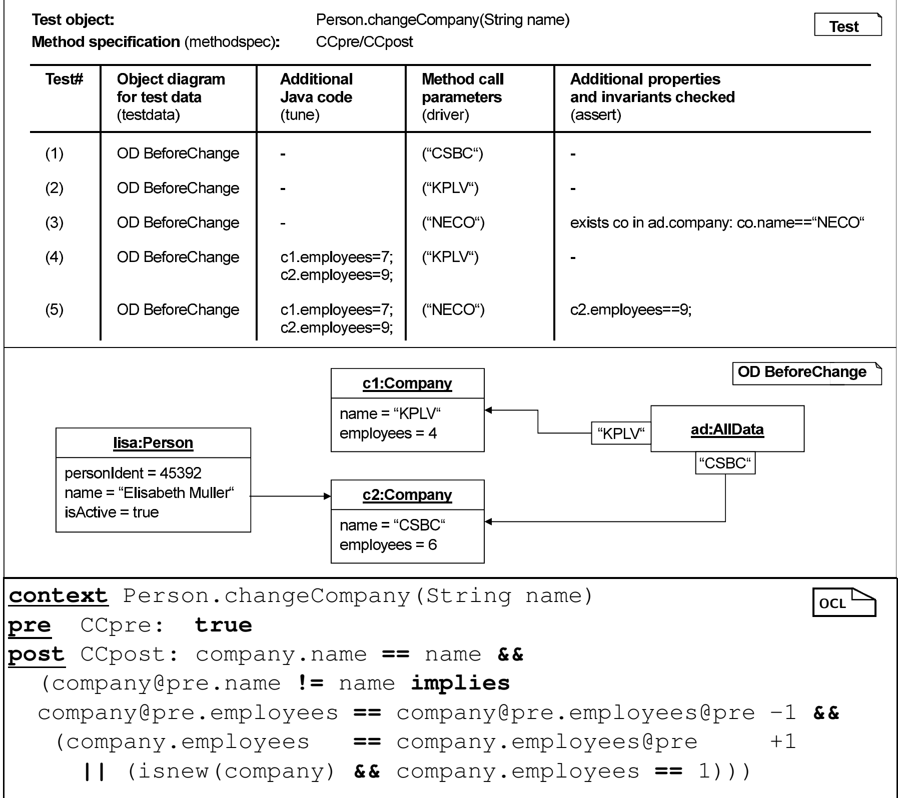
7.3.3 Defining Test Cases using Method Specifications

A method specification on its own does not represent a complete test case—it also needs a set of test data. The table in Fig. 7.7 represents a test suite consisting of five test cases that can be used to generate a test suite for JUnit automatically.

The same object diagram can be used as a set of test data for the first three cases, as all three cases can be varied through the call parameter. For cases (4) and (5), additional Java code is used. This additional code is executed after the construction of the object diagram but before the test itself. Both cases represent variants of existing cases. They primarily check that number of employees has been changed correctly. This appears to be necessary as otherwise an implementation could have a constant number of employees without this being detected.

The possibility of specifying additional OCL constraints or invariants (by name) allows us to check further properties. Thus, case (3) requires that the new company is also registered in the singleton `ad`. Case (5) describes how the company “KPLV”, which is not involved, retains the number of employees registered on the system. Both cases could also be expressed by object diagrams.

The specified object diagram `BeforeChange` is used constructively to generate the test data. Therefore, the object diagram specifies the complete



Test object: Person.changeCompany(String name)
Method specification (methodspec): CCpre/CCpost

Test#	Object diagram for test data (testdata)	Additional Java code (tune)	Method call parameters (driver)	Additional properties and invariants checked (assert)
(1)	OD BeforeChange	-	("CSBC")	-
(2)	OD BeforeChange	-	("KPLV")	-
(3)	OD BeforeChange	-	("NECO")	exists co in ad.company: co.name=="NECO"
(4)	OD BeforeChange	c1.employees=7; c2.employees=9;	("KPLV")	-
(5)	OD BeforeChange	c1.employees=7; c2.employees=9;	("NECO")	c2.employees==9;


```
context Person.changeCompany(String name)
pre  CCpre:  true
post CCpost: company.name == name &&
  (company@pre.name != name implies
  company@pre.employees == company@pre.employees@pre -1 &&
   (company.employees   == company.employees@pre     +1
     || (isnew(company) && company.employees == 1)))
```

Fig. 7.7. Definition of a test suite

environment; there are no further `Person` and `Company` objects. The value specified in the attribute `employees` thus violates invariants which must therefore not be checked in this test.

7.4 Sequence Diagrams

A test sequence is a series of inputs for the test object that this object processes during a test. For systems that are difficult to adapt for testing, there is a lot of effort involved in understanding how to design test cases to establish certain situations and test the behavior of the system in those situations. In contrast, object orientation allows to create the test data directly before the test execution instead of specifying a path from an initial state up to this dataset. There are several reasons for this. Firstly, due to more modern coding standards, in the object-oriented approach methods are typically much smaller than the procedures in the ancient imperative world. Secondly, dynamic binding and building of subclasses simplify the control and manipulation of the test object

by offering it dummies instead of the real implementation at critical points. Thirdly, it is now recognized that the code must be designed or redesigned such that it can be tested easily. For example, in the approach presented in Chapter 8, the creation of new objects is outsourced to a factory that can be replaced by a factory dummy. These measures mean that there is no need for a series of calls to establish a certain situation in the system: we can set it up directly as a set of test data. Therefore, we can control a majority of the test cases with a simple method call once we have created an adequate set of test data. This approach bears the risk, however, that the set of test data used may not appear at all in the real system.

Breaking the functionality down into a number of object-oriented methods gives rise to more complex call hierarchies. The objects thus form more complex interaction patterns that often cannot be recorded adequately using the precondition and postcondition of a method. To understand the interaction between different objects or system parts, these interaction patterns can be modeled as sequence diagrams. A sequence diagram is an exemplary representation of a possible sequence of interactions. It is therefore ideally suited for describing the internal sequence of a test. [PJH$^+$01], for example, discusses the use of an MSC dialect based on UML for the conformance tests for telecommunication protocols.

The section below uses examples to demonstrate how we can use sequence diagrams to model test cases.

7.4.1 Triggers

The example shown in Fig. 7.8 uses the stereotype ≪trigger≫ defined in Section 6.1, Volume 1 to identify the first method call as the trigger. This means that the other method calls and returns specified have to take place for the test described to be successful. We can specify additional OCL constraints within the sequence diagram in order to check intermediate states and to perform a final check of the result. The sequence diagram does not show all of the method calls that take place. It abstracts, for example, from interactions between the auction object and other persons taking part who also receive a notification. This aspect is therefore not tested by the specified sequence diagram.

To complete the test, the execution requires a set of test data in addition to the sequence diagram. We can describe this set of test data using an object diagram, for example. Using an object diagram allows us to arrange the objects specified in the sequence diagram in a link structure that cannot be represented in the sequence diagram. Due to the complexity of the structure, which has to be modeled constructively and therefore completely, in this example it is advisable to describe this structure with two object diagrams. The example assumes that a completed version of the object diagram from Fig. 5.13 is available under the name `copper912` and can therefore be used to

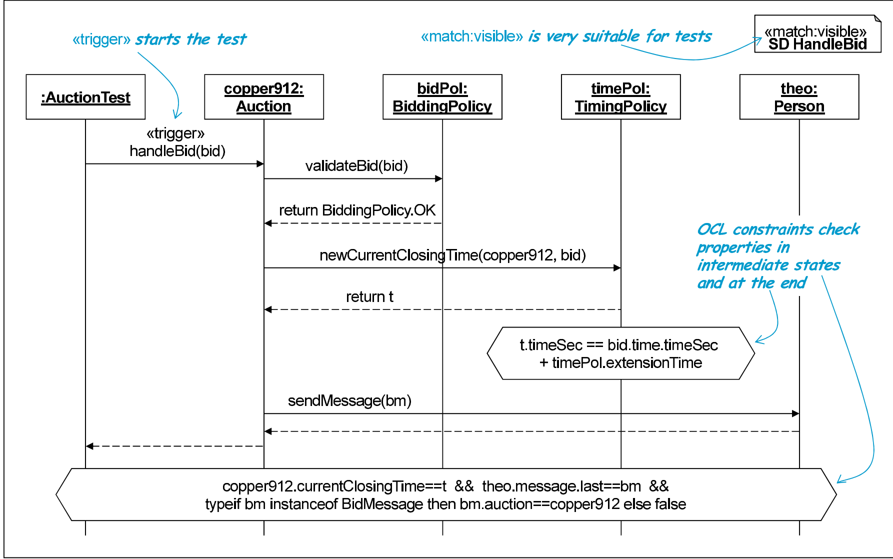


Fig. 7.8. Sequence diagram as a test case description

generate test data constructively. Fig. 7.9 thus contains the complete description of the test based on the sequence diagram `HandleBid`.

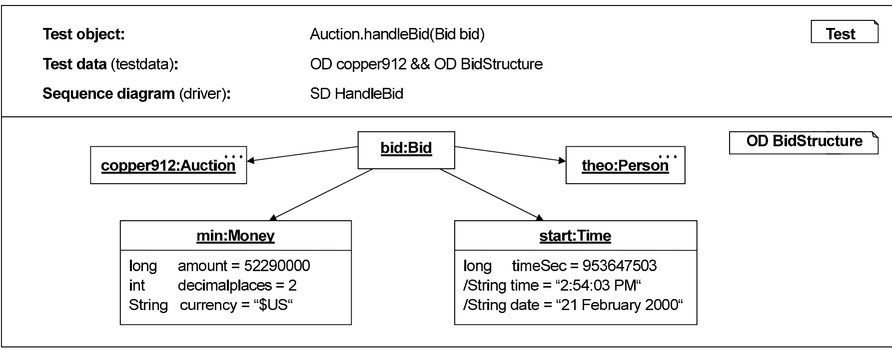


Fig. 7.9. Test that uses the sequence diagram `HandleBid`

As a sequence diagram allows us to specify OCL constraints at various points in the system sequence, we do not have to use an additional method specification although we can specify some for the test.

To allow checking of the specified interactions and the OCL constraints when a sequence diagram is processed, the code must be suitably instrumented. Therefore, as described in Section 5.5, each method call and each `return` statement has to be logged. In addition, the OCL constraints have

to be checked. The check, the construction of the test data, and the execution of the test are localized in the test driver, which is stored in the class `AuctionTest` as a method that can be called by `JUnit`.

7.4.2 Completeness and Matching

According to Section 6.3, Volume 1, there are several ways of interpreting a sequence diagram. For example, the stereotype ≪match:complete≫ defines that all interactions of the object in the described period are shown in the sequence diagram. This strong constraint, which is usually impracticable for specifications, is justifiable in a test because the objects tested were created exclusively for the purpose of the test and are used neither before nor after the test.

The stereotype ≪match:visible≫ has weaker effects. It requires only that all interactions between the objects specified in the diagram are shown, but further method calls to other objects are possible. This stereotype is therefore also a useful interpretation of sequence diagrams for tests.

The use of the stereotype ≪match:initial≫ grants further freedom for the classes tested, as the sequence diagram requires only the interactions specified after the trigger but still permits others. The interpretation with the stereotype ≪match:free≫ is not really suitable for tests if the stereotype is applied to the complete sequence diagram. However, the combined use of stereotypes, as shown in Fig. 7.10, offers an interesting technique.

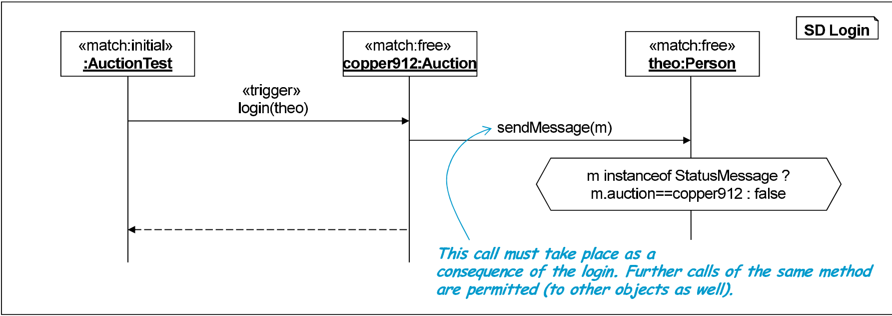


Fig. 7.10. Combined use of stereotypes

When a new person logs into an auction, all of the messages that occur are sent to the person object in succession by the method `sendMessage`. In this case, `sendMessage` is called multiple times. The sequence diagram now requires that there is a call that satisfies the properties specified before the higher level `login` method is finished.

7.4.3 Noncausal Sequence Diagrams

As described in Section 6.4, Volume 1, a sequence diagram can be incomplete and therefore noncausal. For example, we either omitted an object involved in the execution or a method call, that initiated further but observed method calls. These types of sequence diagrams are suitable for tests, as the example in Fig. 7.11 shows.

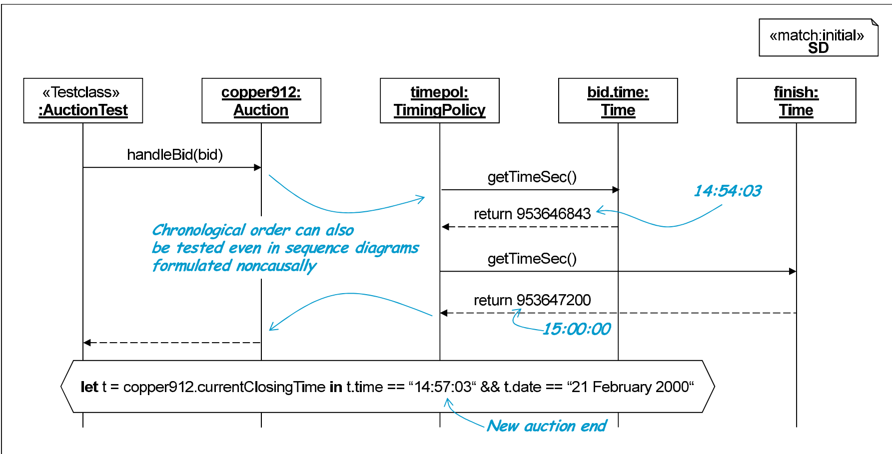


Fig. 7.11. Noncausal sequence diagram as test sequence description

The chronological order of the method calls is defined solely by their order along the time axis and this can therefore be verified. However, the causality cannot be nullified arbitrarily. Naturally, `return` arrows can only be specified if the related method call is specified.

7.4.4 Multiple Sequence Diagrams in a Single Test

As a sequence diagram describes an observation, we can use multiple sequence diagrams for different partial sequences within one test. The two diagrams from Fig. 7.8 and 7.11 therefore complement each other. The sequence diagrams are checked independently of one another as if no other sequence diagram were present. The method calls that appear in both sequence diagrams—in our example `handleBid`—are therefore satisfied by the same method call in the test run. A sequential or alternative composition or iteration of sequence diagrams, as permitted by MSCs [IT11], the UML standard [OMG10], or YAMS [Krü00], would be a possible extension but, as explained in Chapter 6, Volume 1, is not part of UML/P. Instead, the interpretation of multiple sequence diagrams is subject to a loose form of "merging" that can identify identical method calls and thus uses the sequence diagrams independently of one another.

7.4.5 Multiple Triggers in a Sequence Diagram

In accordance with the coding standards common in JUnit, test classes have the suffix "test" or, as proposed in Section 2.5.2, Volume 1, are labeled with a stereotype such as ≪Testclass≫. Because the method calls at the start of the test can only be executed by the test driver, but conversely, each method call (starting from the test class) is to be understood as a trigger, just one of the stereotypes ≪trigger≫ or ≪Testclass≫ is sufficient when defining test cases.

As Fig. 7.12 shows, a sequence diagram can also contain multiple method calls labeled with the stereotype ≪trigger≫. This type of sequence diagram describes a test driver that performs multiple method calls in succession.

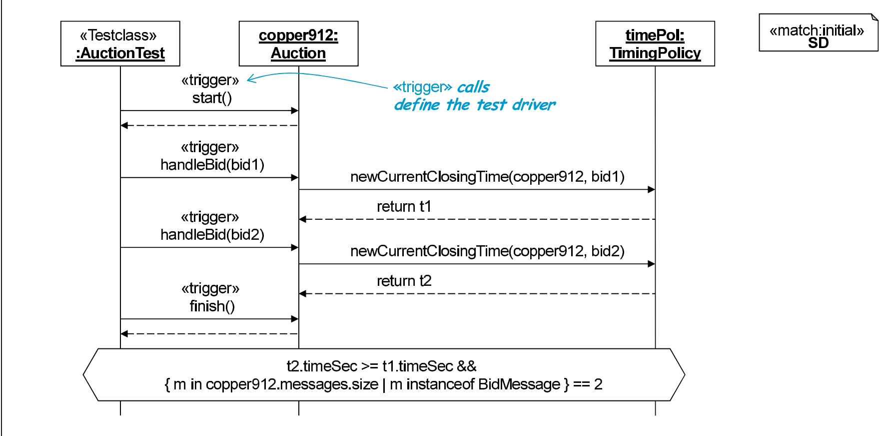


Fig. 7.12. Sequence diagram with successive triggers

7.4.6 Interaction Patterns

Using a sequence diagram to define a test case has the significant advantage that the sequence diagram explicitly represents the interaction pattern tested. By means of a comparison with an existing implementation, it is relatively easy to analyze whether and how well a given set of such test cases covers the various possible sequences in the test object.

In most cases, however, a complete coverage of all possible sequences of a test object is not possible. This is because, similarly to the path coverage test of a procedure [Bal98, Mye01], loops and recursions lead to an unlimited number of variants of possible sequences. Therefore, for practical purposes, we have to define a finite set of sequences that is as representative as possible. This set of sequences is then implemented in test cases. In the auction project, for example, these are auctions with different sets of bids (0, 1, 2, 3, 5, and a

lot) that have been tested in test cases. Such tests, which test the sequence of a sub-phase or even the entire auction, go beyond the individual method test and therefore represent a first step towards integrating different system parts. Sequence diagrams are therefore also suitable for modeling integration tests. In such cases, integration tests primarily rely on the interfaces between the system parts and respect the encapsulation of the system parts. Sequence diagrams with stereotypes such as ≪match:free≫ are suitable here.

Further problems in covering the interaction patterns result from the ever-increasing number of possible object structures that are the basis for the sequences:

- Set-valued associations generally allow an infinite number of different object structures that also require loops for processing and therefore in turn, lead to an infinite number of possible sequence structures.
- The dynamic binding of methods that results from inheritance requires that, for a given object of a class, all subclasses must be tested as well. If there are multiple objects in the set of test data, this leads to an explosion in the number of test cases.
- The modifiability of object structures through their parameters also allows to adapt the behavior of a test object when used in a test run. For example, a parameter can determine how many objects are created in a data structure or how these objects are connected by links.

For practical purposes it is unrealistic to expect a complete coverage of all possible sequences with test cases based on sequence diagrams. As the following example shows, there is also no guarantee that the statements of the methods tested are covered:

Java

```
void method(int i) {
  if(i >= 1)
    foo(i);
  else {
    attribute = i;
    foo(i+1);
  }
}
```

All sequence diagrams for testing `method` show the same sequence structure. It is incorrect, therefore, to assume that two sequence diagrams with the same sequence structure test the same sequences of the implementation.

Defining integration tests with sequence diagrams is therefore just one of several instruments for modeling test cases. We can use it together with the test cases from method specifications and the checking of invariants already discussed.

7.5 Statecharts

The Statecharts defined in Chapter 5, Volume 1 have the following principle application domains:

1. *Executable Statecharts* are transformed into code constructively and are used in the product system or as an oracle function.
2. *Statecharts that can be used for tests* are used to verify the correct execution of a test.
3. Statecharts act as generally valid *behavior specifications* and are used for manual or automated derivation of test cases.
4. *Abstract Statecharts* serve as documentation but are too abstract or too informal for tests and code generation.[2]

If we add detail to the information contained in an abstract, informal Statechart and this increase it details, we can transform the Statechart into a Statechart that can be executed or is suitable for tests. We can use constructive elements (such as actions) and descriptive elements (such as OCL postconditions) in a Statechart simultaneously in a combined form. We can therefore use a constructive part of a Statechart to generate methods and use state conditions and postconditions in parallel, for example, to support tests and generate invariants.

7.5.1 Executable Statecharts

An executable Statechart is used as a constructive model and, as described in Section 5.4, it is transformed into code directly. Executable Statecharts typically have procedural actions in the form of Java code. The Java code that arises from the transformation can be extended by state invariants entered in the Statechart. These state invariants are checked in the code at runtime during the test phase of the system. However, constructive Statecharts are not suitable as a test description since the effect of a procedural action cannot be tested; the action can only be executed.

In principle, we can also use Statecharts as test drivers. However, it is a principle of the definition of test drivers that they must be developed as simply as possible and thus essentially without any branched control structure. However, the linear structure of a test driver allows us to break down a Statechart into a linear form that can also be represented by a sequence diagram.

Using a constructive Statechart as an oracle function for the test result is interesting. This is recommended, for example, if the code `g` that can be generated by a Statechart is too slow for the product system and therefore an alternative implementation `f` was realized. In a test, the set of test data `x` is copied, the respective function is applied to the two sets of test data, and the

[2] In [Wil01], these Statecharts are referred to as "sketches" and not real models.

result is compared accordingly. As shown in Fig. 6.6, we can also define the comparison form explicitly by specifying a `comparator` entry as there are a number of comparison options, for example, depending on whether orders are important in lists.

A black box comparison of the results `f(x)=g(x)` has the advantage that the internal realization of a method can deviate significantly from the description by the Statechart and can thus be subject to a refactoring without loosing the Statechart as an oracle. In this approach, however, the OCL constraints specified in the Statechart are not checked during the sequence. To achieve such an approach, a simultaneous use of the constructive parts of the Statechart to generate product code and the descriptive parts to generate checks in tests is necessary.

When a Statechart is used as a test driver or as an oracle, nondeterminism in the Statechart is critical. Nondeterministic Statecharts, i.e., those with overlapping firing conditions, are mainly useful if the Statechart is used as a sequence description in the test. In this situation, the nondeterminism acts as an underspecification and is replaced by the actual implementation.

Fig. 7.13 illustrates three of the strategies of the auction system which, according to the description in Section D.2, Volume 1, cause different (delta) extensions when a bid is submitted depending on the time of each current bid respectively.

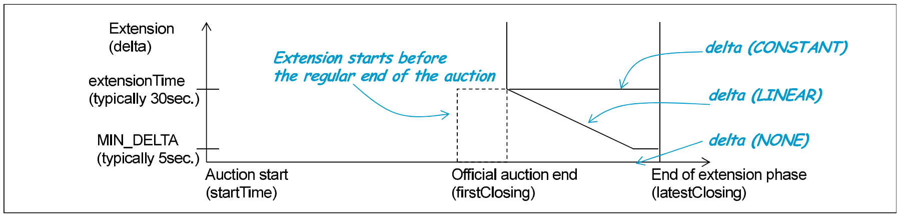


Fig. 7.13. Extension strategies for bids in the auction

The calculation structure for the new end of the auction after submission of a bid is described by the method Statechart in Fig. 7.14.[3] The structure shows the calculation for a constant extension, no extension, and a linearly decreasing extension with a minimum lower limit.[4]

In the actual auction system, the calculation has been distributed over several subclasses of `TimingPolicy`. The original modeling of the functionality shown in Fig. 7.14 is therefore not suitable as a constructive implemen-

[3] For the sake of simplicity, in this example all `Time` objects were replaced by `long` values.

[4] A Statechart developed in this way is typically the first draft and can generally be simplified, as has been done here. In this case, a tabular representation instead of a diagram is also useful.

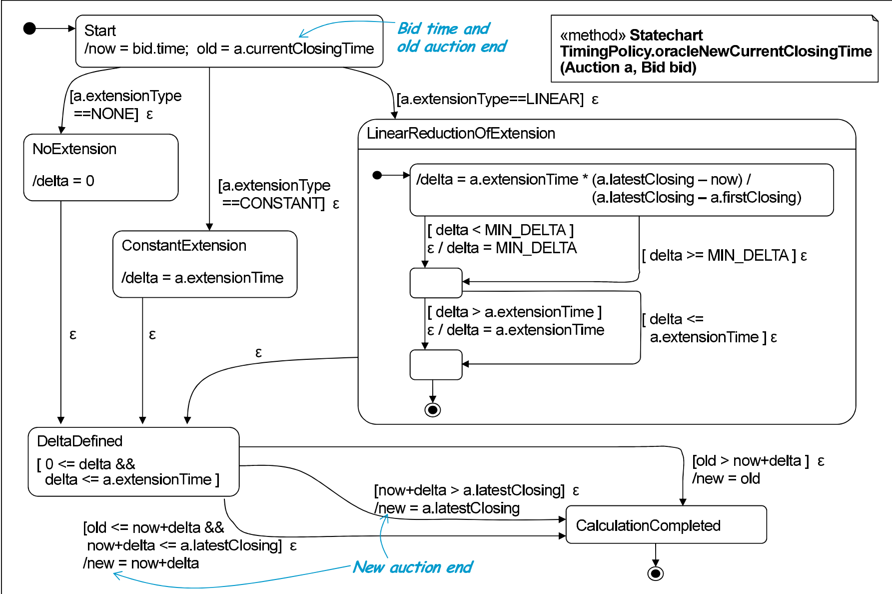


Fig. 7.14. Statechart calculates new end of auction

tation. However, due to the fact that the content of the model is correct and the model can be executed, the model can be used as an oracle function.[5]

Therefore, the modeled functionality can be used as an oracle in tests, as shown in Fig. 7.15.

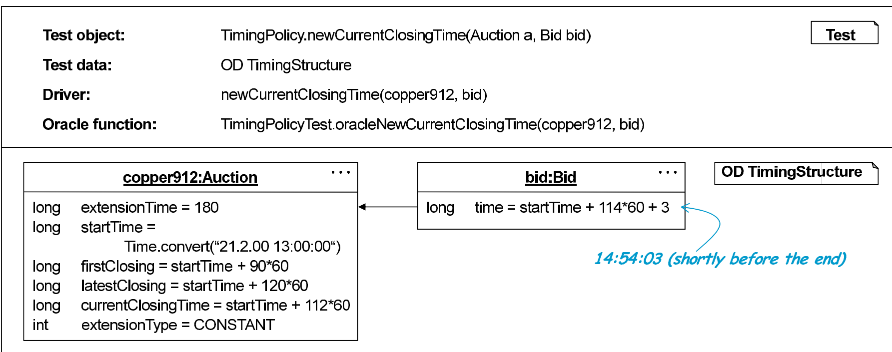


Fig. 7.15. Test with a method Statechart as an oracle function

[5] Parallel to the refactoring of the functionality of `newCurrentClosingTime` with the shift to the `TimingPolicy` classes, the attributes used must also be adapted. However, these adaptations are not shown in the example.

7.5.2 Using Statecharts to Describe Sequences

We can use a Statechart that models a life cycle, similarly to a sequence diagram, to check the accuracy of a system execution. In this situation, the Statechart is understood as a predicate for a test case that begins in a specified state and whose interactions are monitored by the Statechart. This means that in a test, a Statechart can also be used as a predicate that the test needs to fulfill. However, the Statechart itself does not represent a complete test case, as both the test data and the test driver are missing.

We typically describe a test sequence with a sequence diagram and model the objects involved with an object structure. We can now attach Statecharts, whose validity must be checked, to one or more of these objects. However, the objects do not have to be in an initial state specified by the Statechart or reach a final state during the test run. For each object, therefore, we also specify the relevant state at the beginning of the test and the permitted states at the end of the test. These states are set or checked dependent on the transformation of the states with an enumeration attribute or predicates as described in Section 5.4. As a Statechart represents a variety of paths, Statecharts can easily be reused for multiple test cases.

Fig. 7.16 shows a suitable Statechart and an associated test driver which could alternatively be represented as a sequence diagram. Further test cases can start in other states or take other paths by considering bids (`bid` calls), for example.

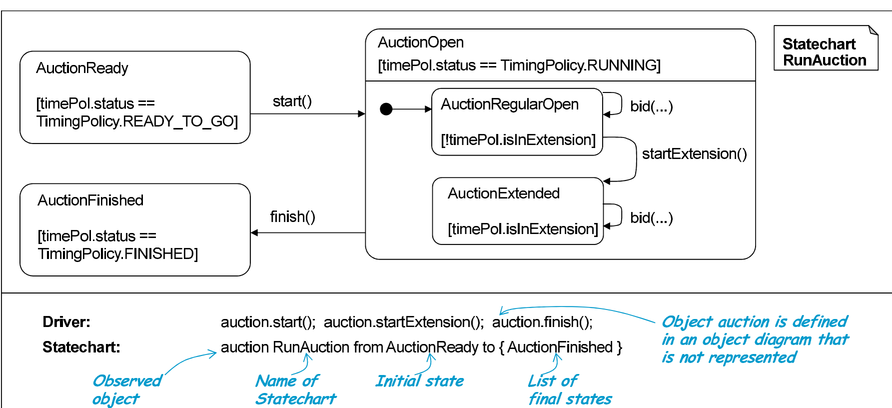


Fig. 7.16. Statechart as a life cycle description in a test

To compare the test sequence with the transitions and states of the Statechart, we need an instrumentation of the test object in a similar form to that for a sequence diagram. This means that at the beginning and end of every method, a suitable instrumentation is built in, for example, to allow us to check states, state invariants, and transition enabledness. However, not all of

the properties that can be formulated in a Statechart can be used for testing. For example, procedural actions are not suitable as test specifications and must therefore be ignored (if specified).[6]

Furthermore, the control states (stereotype ≪controlstate≫) of a method Statechart cannot be checked as they cannot be automatically connected to program points in the code. The descriptive part can only be used as assertions in the generated method if the implementation is created from the procedural part of a method Statechart constructively.

For the piecewise composed actions (stereotype ≪action:sequential≫) discussed in Section 5.5.2, Volume 1, it is also only possible to check the conditions that have become valid in between if the Statechart has been transformed constructively.

7.5.3 Statecharts used in Testing

As Statecharts describe a potentially infinite set of execution paths for an object, rather than just one single sequence, a Statechart is an excellent starting point for developing tests and for measuring the coverage of the behavior parts modeled in the Statechart. In contrast to earlier forms of model usage in test cases, in this case, a number of test cases can be derived from one diagram rather than one diagram being used for one test case.

If a Statechart is used constructively, i.e., the Statechart represents the implementation, the assertions generated from the predicative part (OCL constraints) are understood as white box tests. However, if there is an implementation that has arisen independently, the Statechart is compared with the implementation as a specification. In that case, the tests are black box tests.

A topic currently being researched is the generation of collections of sets of test data that are as compact as possible but complete, and that, from given Statechart specifications, achieve a coverage in accordance with specified coverage metrics. Procedures have already been developed for different variants of flat automata [PYvB96, RDT95]. Section 7.5.6 discusses further approaches.

As already described, it is not generally possible to decide whether an OCL constraint can be satisfied. If an unsatisfiable OCL constraint is used as the precondition for a transition, no path can be found that contains that transition. It is therefore not possible to cover all transitions. Due to these undecidability problems, for the Statecharts defined in UML/P, and for many other variants of automata, there is no automatic procedure that generally creates a complete test suite according to a certain criterion.

A further problem arises from the possible nondeterminism in a descriptive Statechart—as a result of overlapping firing conditions, for example. If

[6] It is possible to use procedural actions as an oracle function g for tests of the form $f(x) = g(x)$ of the effect of each individual transition. However, this has limited efficiency as the respective current structure has to be copied for each transition.

a Statechart has two transitions with identical firing conditions but different final states, as shown in Fig. 3.38(a), for example, it is possible that an implementation will always select the same alternative. The second transition is therefore not selected and cannot be covered by tests. The overlapping of firing conditions and the preference of one transition over another in an implementation also cannot generally be recognized automatically. In this situation, a tool is useful that, based on existing tests, measures the degree to which a Statechart was covered and indicates deficits (where applicable).

What both forms of Statecharts have in common is that the transition paths they describe represent a certain abstraction of the complete implementation:

- A complex statement, including loops and case-statements, can be embedded in an action.
- An individual diagram state can correspond to a set of object states.
- An OCL constraint can consist of several alternatively satisfiable clauses grouped in one disjunction.

These conceptual requirements should be taken into account for detailed tests. We can achieve it by combining coverage metrics for these elements with the procedure described below for Statecharts (similarly to [GH99]). This means that once we have developed the tests based on the Statechart structure, we refine them to the extent that the individual components of conditions and actions in the Statechart are also covered. An alternative is to measure the coverage based not on the Statechart but on the code generated from the Statechart.

A further aspect is the integration of the execution sequences in the called methods of the same or sub-objects. In particular, if there is a composition relationship to sub-objects, as it is the case in the auction example between the `Auction` object and its `Policy` objects, then the behavior of the dependent objects is integrated in the tests of the composite. However, this causes the number of required tests to increase significantly, as different configurations of object structures and the composed state models including all objects, must be considered. An estimate should help to quickly indicate whether an attempt at coverage can be executed from a practical perspective. Therefore, the subsystem to be tested and the desired coverage criteria must be selected appropriately. Furthermore, we have to select test cases intelligently and, therefore, manually. [Mye79] already notes that experienced testers can achieve good results without explicitly applying a systematic approach by "error guessing", that is, guessing potential error sources. [PKS02], however, demands that for many applications, or at least critical system parts, "error guessing" should be seen as an additional technique to more systematic approaches.

7.5.4 Coverage Metrics

When we at first sight ignore the complexities discussed above for actions and OCL constraints, we can identify the following metrics for the coverage of a Statechart provided by a test suite. They are already identified as control flow-based metrics in [FHNS02] and [Bal98]:

State coverage requires a test case for each state that can be reached in the diagram. Here, a test case defines a sequence of inputs that leads from an initial state to this state.

Transition coverage requires a test case for each enabled transition. The sequence of inputs describes the path from an initial state up to the execution of the transition.

Path coverage requires a test for each possible path from an initial state to a final state. This form of coverage subsumes the two previous forms. However, if there are loops in the automaton, the set of paths is infinite and thus, from a practical perspective, path coverage is not possible.

Minimal loop coverage is a reduced form of path coverage. Here, loops are not processed multiple times. However, each loop that appears in the Statechart must be processed at least once.

According to [Bal98], path coverage has no practical relevance because when loops are present, it becomes an infinite and thus no longer executable task. In contrast, the minimal loop coverage is a strong yet practicable criterion that should be used for high-quality software. For Statecharts without loops, however, both metrics are equivalent. Both metrics may count paths that are recognizable in a Statechart, but can actually not be executed in an implementation. If the precondition of a transition can never be satisfied because of its invariant, this transition cannot be executed. This is also true if the object modeled by a Statechart is embedded in a test object consisting of multiple objects and is therefore not directly accessible. In this situation, the environment can also prevent a required input sequence from occurring. Therefore, we have to select the test environment appropriately in each case.

To determine the coverage metrics for a test suite, the test executions are logged by an instrumented version of the code.

We can also apply the original coverage metrics developed for flat automata to Statecharts with a hierarchical state concept in this form. However, due to the transformations to Statecharts discussed in Section 5.6.2, Volume 1, we can also perform the tests based on the expanded hierarchy. This refines the required test coverage because, for example, when a transition to a hierarchical state is triggered, this transition is multiplied. The expansion can therefore be used to get a more detailed test coverage.

Using the Statechart from Fig. 7.14, the four metrics referred to above are shown in Fig. 7.17. The illustration uses structurally identical, iconified forms of the initial Statechart to illustrate the progression of the test through the Statechart. The three paths of the state coverage are not sufficient for the

transition coverage. A fourth sequence must be added and two of the existing sequences must be modified so that the transitions that occur at the end are also covered.

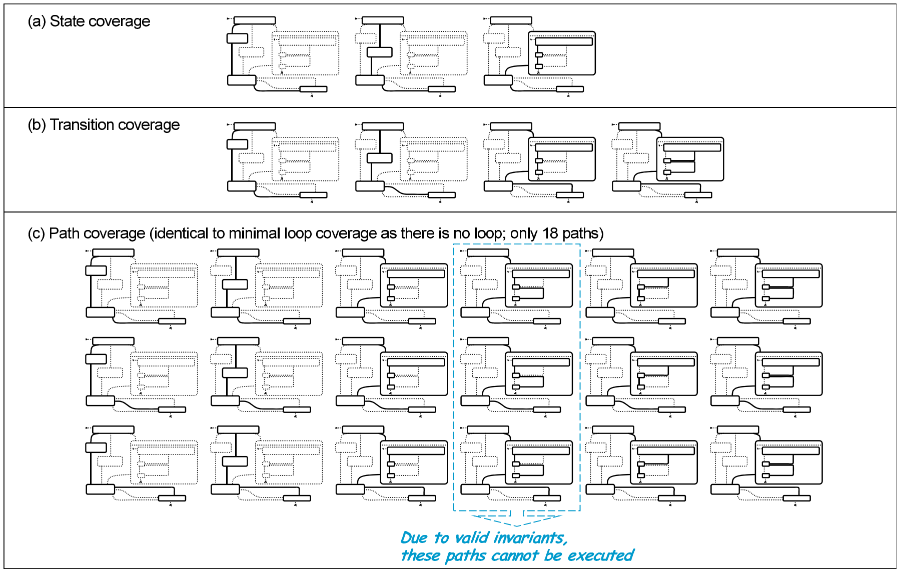
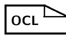


Fig. 7.17. Test case paths to satisfy the metrics

The path coverage requires 18 test cases. As the Statechart to be tested is a method Statechart, the input consists of only one method call. Where applicable, there may also be further interactions with the environment, with methods of the environment being called and further control of the transition progression enabled through `return` values. In this example, however, the progression of the test case is dependent exclusively on the initial input and the value assignment in the initial object structure. To achieve a 100% path coverage, therefore, we have to find 18 different assignments for the variables used in the Statechart. Consider the following invariant:

```
context Auction a inv:
  MIN_DELTA <= a.extensionTime
```



This invariant requires, for example, that the `extensionTime` of an auction is always greater than the minimum extension time of `MIN_DELTA` (5 seconds). Therefore, the three paths marked cannot be executed. Suitable test data can be found for the other 15 paths. Borderline cases such as the arrival of a bid at exactly the time the auction ends or just afterwards should also be covered. We can identify further borderline cases by analyzing the triggering conditions of transitions and thus derive further tests.

A second example demonstrates the metrics using the Statechart represented in Fig. 7.16. There, each transition must be controlled by a method call, such that the input is a sequence of calls.

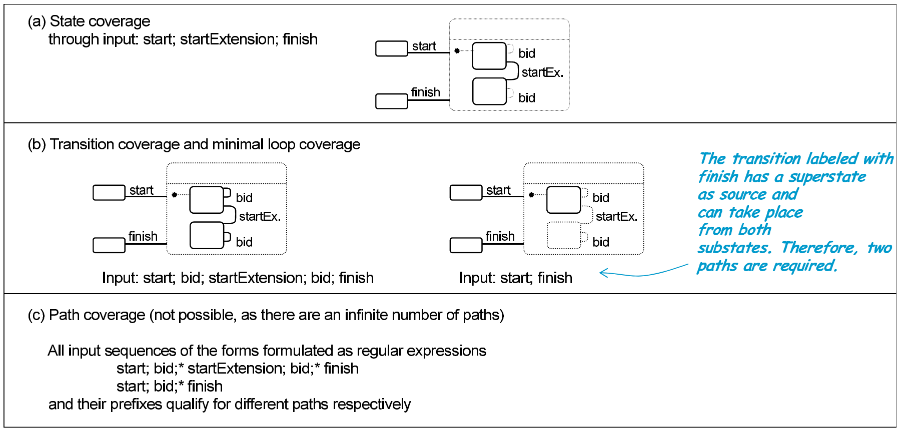


Fig. 7.18. Test cases for satisfying the metrics

The possibility of giving Statecharts different semantics with respects to completion and error situations by using different stereotypes means that we have to take these stereotypes into account when defining tests. The following situations and strategies are possible:

1. The Statechart is complete in that, for example, an error state was introduced. This means that there are implicit transitions that lead to the error state and whose firing conditions cover everything that is not covered by explicit transitions. These transitions can be included in the coverage. As Statecharts can have a second error state for handling exceptions that occur, the situation that arises here is the same. Again, we have to decide whether the processing of exceptions has to be tested and if so, to what level of detail.
2. The Statechart was completed with ≪completion:ignore≫. This gives rise to transition loops that can be tested for transition and path coverage.
3. A Statechart labeled with ≪completion:chaos≫ is based on the assumption that the Statechart defines only part of the behavior. It describes the behavior of the object up to the first time a situation occurs in which the Statechart is not enabled and the test object can take on any arbitrary behavior. This type of completion does not have to be tested.

Completion with ≪completion:ignore≫ makes it particularly obvious that testing these additional transitions is not very important for the behavior of the object described with the Statechart as these transitions are generated uniformly. Therefore, it is usually more interesting to test calling objects of

the environment to find out whether they can cope with a method call being ignored or an error state occurring. If we assume that, through its incompleteness, a Statechart describes all permitted calling sequences of an object, this gives rise to significant restrictions on the environment that must also be checked at least with tests. We can implement this, for example, with a completion transformation in which the test is reported as having failed when the error state is reached.

7.5.5 Transition Tests instead of Test Sequences

With the test sequences discussed so far, we have generally assumed that the object described is in an initial state. Therefore, to test a transition, we first have to find a path that leads to the source state of this transition.

The basic structure of test cases characterized with Fig. 6.7 shows that a test case can start from any arbitrary set of test data. This means that we can also test a transition of a Statechart by finding test data that corresponds to the source state of the transition and enables this transition. This applies above all to the life cycle of an object in which a method call corresponds to a single transition and can thus be executed in isolation. Therefore, instead of requiring a test sequence that begins with an initial state, it is sufficient to execute a simple method call to test a transition.

Even though tests are generated automatically, defining test data that corresponds to a source state of a transition circumvents the problem discussed in Section 7.4 of having to find a test sequence that leads to a certain state or to a transition. Instead, we only have to find an object structure in which the test object corresponds to a Statechart state. It is therefore essential to characterize diagram states in sufficient detail with OCL constraints. Otherwise, with this approach, it happens too often that object states that cannot be reached in the product system are used as test data. When searching for object states that correspond to a diagram state, we also have to consider the globally valid invariants.

Thus, the search for test data for transitions from Statecharts is rather equivalent to the search for test data discussed in Section 7.3.2, in which the test data has to satisfy the precondition of an OCL method specification. If we use a procedure for identifying test data that satisfies an OCL precondition, then due to the transformation of Statecharts into OCL presented in Section 5.6.2, Volume 1, we can also apply this procedure on transitions in life cycle Statecharts.

An advantage of the transition-based approach outlined here is the ease with which we can test each transition, which means that tests run more efficiently. The disadvantage is that it is not possible to recognize whether a state can actually be reached. Transitions starting from such states do not have to be tested as they do not contribute to system behavior.

The approach outlined is equivalent to transition coverage but ignores the paths from the initial state to the source state of the transition to be executed.

7.5.6 Further Approaches

As already stated, determining test cases from automaton-like description techniques with various procedures is an ongoing area of research. By way of example, the following approaches briefly outline some progress in this field.

For more simple variants of state-based systems, [GH99], for example, outlines a procedure for generating automated tests from executable automata by means of model checking. These tests check an implementation against the automata used as an oracle function. However, these automata have no state hierarchy, no nondeterminism, and, compared with the Statecharts presented here, a limited form of transition markings. The procedure achieves transition coverage for the automaton. Model checking is used primarily to find call sequences for the automata that satisfy preconditions of the respective transition to be tested. This procedure can e.g. be refined in two ways: on the one hand, existing preconditions are deconstructed from disjunctions and thus the transitions implicitly divided in order to achieve coverage of all clauses of a disjunction. On the other hand, a simple boundary value analysis is performed by deconstructing boundary comparisons such as `a>=b` into two cases `a>b` and `a==b`.

[FHNS02] describes an approach for reducing the state space from a state automaton by means of projection. This simplifies the coverage of the automata obtained as a projection according to the criteria of state and transition coverage. However, the paths through the system that are actually tested and the errors discovered depend on the selection of the projection, because paths and errors may be distributed over several of the projected automata. This procedure is helpful in situations in which we have an object with many states. In object-oriented systems, it is advisable (although not always possible) to use this projection in the design phase by outsourcing subfunctionality. This results in multiple objects whose state spaces can each correspond to a projection. The projection technique from [FHNS02] does, however, offer additional flexibility, as it allows overlapping projections and thus permits overlapping views for tests.

Similarly, the approach in [CCD02] discusses the use of a restricted form of the UML Statechart to obtain tests. Object diagrams are also used for the test data. However, the authors of this work, explain the architecture of a tool coupling based on XML rather than the approach of generating test cases from the hierarchical Statecharts.

7.6 Summary and Open Issues Regarding Testing

Summary

Quality management is not possible without developing tests systematically and in a disciplined way. However, defining test cases requires a lot of effort,

particularly as business-critical systems require good coverage by test cases. Therefore, it is very important to develop and represent *test cases* efficiently. The techniques of this book based on UML/P allow a compact and clear representation of tests by means of the combination of various diagrams for representing the *test data*, the *test driver*, and the *expected result* as well as *invariants* of the system. These diagrams and specifications can be developed independently of one another, are more compact and easier to understand than test code, and therefore simplify reuse.

In a test-first approach, we can use sequence diagrams initially to specify behavior patterns which we can then use, with additions where applicable, as test case descriptions before starting an implementation. This approach is thus an extension of the classic approach of using sequence diagrams during the early gathering of requirements only.

After an implementation, it is useful to define further test cases to test borderline and special cases. The intensity of the testing depends on the desired quality and the underlying methodology. In an agile approach, metrics for analyzing the test quality are used primarily at particularly critical and complex points but otherwise we can put some trust in the experience of the test developers.

A UML/P model can be the object of the test if we use the model *constructively* to describe the *system*. However, we can also use a model as a *testable specification* if there is already an implementation. As the procedures for code generation in Chapter 4 and Chapter 5 have shown, some concepts of the model can also be used constructively or as test code depending on the generation approach.

After introducing the terminology for test procedures and briefly describing two significant tools for testing, this chapter demonstrates how we can use UML/P diagrams to model tests.

Despite the available literature on *conformance testing* for code generated from graphically denoted implementation models, there are a number of open issues and possibilities for improvement, some of which are discussed below. This book also does not examine load tests, quality management measures such as inspections and model reviews, or the approaches for interactive tests with user participation for acceptance purposes.

Development Potential for the Generation of Test Cases

The use of UML/P diagrams and in particular OCL specifications for generating automated test cases offers great development potential. As described in Section 7.3, using an OCL method specification to generate a set of test cases that is as effective but also as small as possible, and that covers the code sufficiently, would be helpful for developing test cases more efficiently. The same applies for the Statecharts discussed in Section 7.5.

In the context of UML and OCL, there is not much known work on the topic of agile testing. One reason for this is that for a long time, the assump-

tion was that the test procedures, which had all generally been developed for procedural programming [Bei04, Lig90], simply need to be applied to object orientation. However, inheritance and the dynamic binding of methods give rise to new problems.

Another source of problems is the decoupling of the language UML and the many, to some extent very varied process models based on UML [BL02] which use models with very different levels of detail that therefore have to be tested very differently.

Nevertheless, there is now an increasing volume of work available on applying the various test procedures to UML without, however, already exhausting the potential for generating code and tests. For example, [PJH+01] addresses applying sequence diagrams for modeling test cases which we know from TTCN [ISO92]. [BL02] describes the systematic and partially automated development of test cases from UML analysis documents such as use case diagrams, sequence diagrams, communication diagrams, and the class diagrams which represent the domain model. [BB00] takes a similar approach which also uses a form of sequence diagrams to describe interactions between objects and combines this with a known procedure for category partitioning [OB88].

[PLP01] describes the application of constraint reasoning techniques for a graphical modeling language suitable for distributed systems but not based on UML to generate a set of test cases (referred to there as "*test sequences*") from an abstract *test case specification*. However, the systems examined there are static and, therefore, the transfer of algorithms to dynamic object-oriented systems is not canonical.

The publications [DN84] and [HT90] describe how the random generation of test data produces results that are equally good as those for partition-based test procedures for developing confidence in the accuracy of the implementation. If these results prove to be true for today's languages and test procedures as well, it indicates that random generation of tests can be an important help for test procedures. In particular, Statecharts and OCL postconditions are then helpful as an oracle for this type of generated test. If the generated tests are to be stored, object diagrams and sequence diagrams can be used. As noted in [HT90], for an effective test activity we must further replace the manual creation of test cases—which is very work-intensive—to automated test generation.

This chapter has discussed only a small part of the testing concepts, techniques, and approaches. Therefore, references were given to relevant literature at various points. In general, we can state that supporting the development of test cases effectively requires not only good and parameterized generators for UML, but support must also be offered for test patterns and for frameworks and components that are to be integrated.

In object-oriented and in particular, agile process models, there is a trend towards explicitly structuring the product system in such a way that simplifies testing. This means that we no longer have to define a test sequence that

tests a certain transition, statement, or method starting from an initial state. Instead, we can specify an object structure that allows us to start the test directly in the desired object state. Thus object structures are much easier to find and the test is executed more efficiently.

Symbolic Testing

One example for the potential evolution of test procedures is based on the interpretation of symbolic instead of real values. Thus, the use of abstract elements represented by symbols in object diagrams, as discussed in Section 4.2.2, Volume 1, can be a basis for an interesting extension of the use of these diagrams and this is illustrated as an outlook here. Symbolic values can be used to generalize test cases: in this approach, a symbolic value is specified in a precondition. This value changes within the method being tested or is used to calculate other attributes. The expression is not evaluated and is instead stored unevaluated as a term. In the postcondition, instead of checking a specific value, we can then check whether the term used for the calculation corresponds to the desired intention. The most simple form of comparison is syntactic equality. However, using a suitable algebraic reformulation, we can also specify a semantically equivalent term. One example is the following calculation, which produces the maximum of both arguments in a circuitous way:

Java

```
int foo(x,y) {
  int a = x;
  int b = y;
  a = a-b;
  if(x < y) b = b-a;
  return a+b;
}
```

Instead of using specific values, the calculation uses the symbolic values `x` and `y`. With the additional assumption `x<y`, the test result for the function value `y` is predefined. The symbolic interpretation can be described with the following annotation:

Java

```
int foo(x,y) {              // Value a=      b=
  int a = x;                //       x
  int b = y;                //       x       y
  a = a-b;                  //       x-y     y
  if(x < y) b = b-a;        //       x-y     y-(x-y)
  return a+b;               // Result: (x-y)+(y-(x-y))
}
```

The result `(x-y)+(y-(x-y))` is actually equivalent to `y`. We can “test” the negation of the condition `!(x<y)` symbolically in the same way. With this form of interpretation, the calculation is based at least in part on symbolic

values. This allows exemplary test cases to be generalized to form equivalence classes of test cases. In the example above, there are only two equivalence classes, described by `x<y` and `!(x<y)`, meaning that complete test coverage is possible. Symbolic testing can therefore be a mechanism for designing a finite set of generalized tests that represents coverage of the entire system functionality. This creates a bridge between exemplary test procedures and general verification techniques [Lig90]. However, in an object-oriented programming style that is laden with side effects, such procedures involve a lot of effort. When loops are involved we get infinitely many equivalence classes, such that the test coverage remains incomplete. Today, unevaluated data structures are rarely used for testing and instead, are used successfully primarily in implementations for logical programming languages such as Prolog [Llo87], functional programming languages with a "lazy" evaluation such as Gofer [Jon96], mathematical, algebraic systems such as Maple [Viv01] or Mathematica [Wol99], term rewriting systems such as OBJ [GWM+92], or verification systems with term rewriting [NPW02, Pau94]. They are also used for more complex tasks, such as verification.

Statistical, Application-Based Tests

A special form of test development can be found in the *cleanroom* approach [PTLP98] to software development, for example. This approach uses statistical Markov models to describe the probabilities of use cases. As cleanroom works primarily with input and output sequences described by state transition models, the statistical model weights the potential input sequences according to probabilities. This enables more intensive testing of the most probable applications. In the context of UML, this procedure can be applied to Statecharts. For this, the transitions are weighted according to the expected frequency of occurrence. This means that we develop a model of the class being tested. The model describes which method calls and messages occur with what probability. The test paths for the Statechart are then selected based on this weighting.

Alternatively, weighting the states according to the expected frequency of occurrence would also be interesting, although this technique is not propagated in the cleanroom approach itself. Furthermore, the same procedure could also be applied profitably for class diagrams to achieve a weighting of object structures to be used as sets of test data.

In the cleanroom approach, the use of statistical procedures allows us to predict average error frequencies dependent on the number of tests performed and the errors found in those tests. In particular, this allows us to develop a precise criterion for the end of the test dependent on the desired quality (error frequency). One of the main basic prerequisites for this, however, is the gathering of a meaningful set of data that indicates how well the tests developed in this way actually discover errors. Such data is dependent on the programming language used and can only be collected in suitable field trials.

8

Design Patterns for Testing

> The two grand tyrants of the earth:
> time and chance.
>
> Johann Gottfried von Herder

To supplement the description of the theory of and the general approach for developing tests, this chapter demonstrates how to use UML/P based test patterns. We here use these patterns to illustrate how to define test dummies and how to design functional tests for concurrent and distributed systems, for example.

8.1 Dummies **219**
8.2 Designing Testable Programs **224**
8.3 Handling of Time **232**
8.4 Concurrency with Threads **237**
8.5 Distribution and Communication **245**
8.6 Summary **253**

© Springer International Publishing AG 2017

B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_8

The collection of design patterns [GHJV94] is one of the first works to show that proven structures and techniques can be developed, documented, and thus reused in software development projects. For test purposes, we can also identify patterns that are valid generally or for specific topics. This section describes some example test patterns to show how we can use UML/P to implement such test patterns. The test patterns covered in this section concentrate on the following topics:

- Dummies[1]
- Designing programs that can be tested in terms of the use of static variables, object instantiation, and the use of predefined frameworks
- Simulating time
- Concurrency
- Distribution and communication

The test patterns described here are used primarily to prepare the test object and its necessary environment so that we can test the functionality effectively. We use dummies, for example, to simulate the environment of the test object and catch side effects or provide predefined returns.

We can also inject the concepts covered by the test patterns retrospectively into existing software. Embedding these test patterns in corresponding rules for refactoring models, as discussed in Chapter 10, is a suitable way of injecting the test patterns.

[Bin99] contains a recommended collection of test patterns which discusses tests that cover the full range from single methods up to complete systems. It discusses, for example, test strategies for covering control flows, integration tests, or regression tests. [LF02] complements this discussion by examining technology-specific topics such as persistence using databases, communication with CORBA [OH98], the use of frameworks such as Enterprise JavaBeans [MH00], or technologies such as Java Server Pages [FK00]. [PKS02] is dedicated to improving the test process and uses checklists to optimize the test process on a project-specific basis, for example.

In analogy to the design patterns in [GHJV94], a *test pattern* is a generic description of a recurring design problem that specifically supports the ability to test the system. A test pattern addresses the form how to define the tests, the structural changes required to get testable code, and the strategy for executing the tests. However, because this strategy is driven primarily by the test-first approach (see Section 2.3.2 and [Rum04]) in the sense of an agile methodology, in contrast to [Bin99], the test patterns discussed here focus primarily on supporting the execution of efficient tests. This requires a system architecture that is suitable for tests. The focus is also on the implementation of these tests using predefined frameworks and components. The idea that a system should be designed such that it can be controlled and its internals observed for tests has meanwhile been accepted [Bin94, HBG01].

[1] The term "*mock*" is also commonly used for "*dummy*".

A part of this chapter is dedicated to functional testing of distributed systems that communicate internally and with the environment asynchronously or have multiple concurrent threads. The chapter thus supplements the collections of test patterns mentioned above. Distributed systems are typically nondeterministic. For example, to enable a better responsiveness, the processing of user inputs from the interface is handled with different threads. Internet queries are also processed concurrently and thus concurrency occurs in almost every system today. The auction system in our examples is also distributed across a number of clients and servers via the Internet but also has multiple threads within each of these system parts.

Nondeterministic effects are generally critical for automated tests as they make it more difficult to evaluate or repeat the results. In principle, we can handle nondeterministic results in test executions by using OCL constraints that allow to specify a range of outputs as test success. However, this does mean that a significant criterion for automated tests, namely repeatability, is lost. It is often difficult for the developer to identify errors when a test fails only sporadically.

For functional tests, therefore, nondeterministic effects should be completely suppressed. We can do this via clever use of *dummies* and by explicitly defining parallel executions with a controlled scheduling. This is explained in the following sections.

The schema for presenting patterns, as introduced in [GHJV94] and continued for many types of patterns since, can also be used for test patterns. This section uses only a subset of the schema from [Bin99], as we use it primarily for a summary at the end of a discussion.

Section 8.1 discusses the introduction of dummies to define the environment of a test object. Section 8.2 describes general problems that arise from the underlying language during testing of object-oriented systems and provides guidelines for structuring the product system so that it is more easily accessible for tests. This includes, for example, encapsulating static variables and making object instantiation dynamic via the factory design pattern. Section 8.3 discusses the simulation of time in a functional, repeatable test that is thus designed in a deterministic form. Sections 8.4 and 8.5 discuss concurrency, distribution, and communication aspects that also increase the difficulty to define functional tests beyond the boundaries of individual processes. These sections introduce several test patterns that significantly improve or even enable testing of the functional properties of such systems.

8.1 Dummies

A dummy is an object that simulates part of the environment of the test object and thus provides the test object with an environment in which its behavior can be tested. Object-oriented programming has made it much easier to use dummies. This is because inheritance and the dynamic redefinition of

methods, as well as the implementation of interfaces, make it relatively easy to inject dummy objects into the system without making the product code dependent on these objects as a result of syntactic (for example, `import`) relationships. Fig. 8.1 shows how dummy classes are typically modelled. By convention, these classes receive the suffix "dummy" but can also be labeled with a suitable stereotype ≪dummy≫ and are available only in the test system.

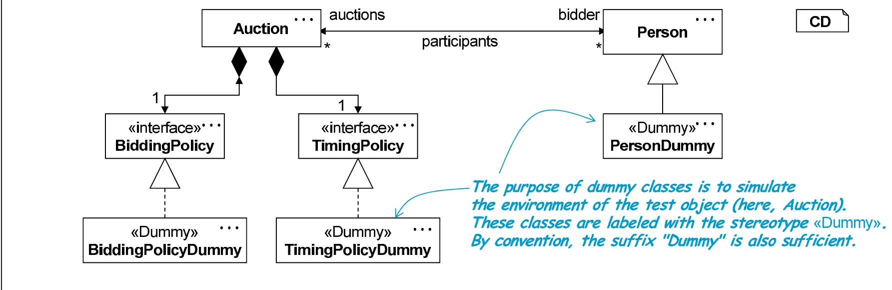


Fig. 8.1. Dummy classes overwrite methods

This allows us to set up test cases for the class `Auction`, for example. Fig. 8.2 shows an excerpt of an object diagram that is used to set up such test cases.

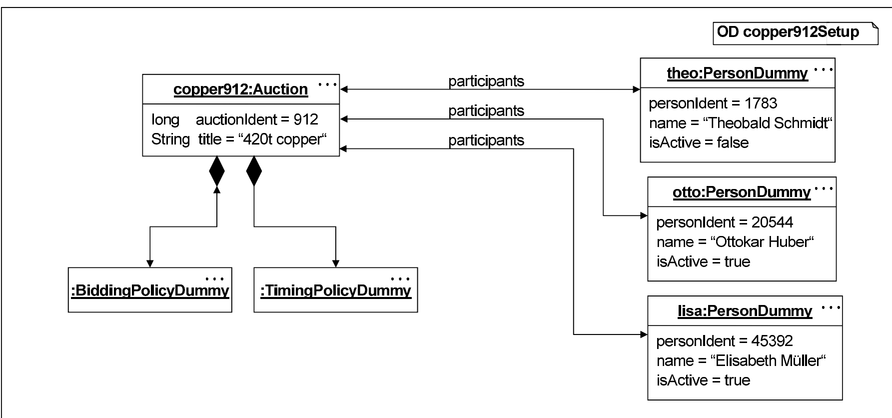


Fig. 8.2. Test situation for an `Auction` object

In the `dummy` classes, methods used during the test can be suitably overwritten. A simple dummy method returns a predefined, constant result or reads results from a list in order. In the auction project, for example, the

dummy from Fig. 8.3, which is special for a certain test, is used to simulate requests of the current time.

```
class TimingPolicyDummy implements TimingPolicy {              Java/P
  Time[] timelist = new Time[] {
        new Time("14:42:22", "Feb 21 2000"),
        new Time("14:42:23", "Feb 21 2000"),
        new Time("14:44:18", "Feb 21 2000"),
        new Time("14:59:59", "Feb 21 2000") };
  int count = 0;

  public Time newCurrentClosingTime(Auction a, Bid b) {
    ocl count < timelist.length;
    return timelist[count++];
}}
```

Fig. 8.3. Dummy with results for four calls

Dummies are typically not necessary for testing simple classes with little functionality. For other classes, multiple different dummies may be necessary depending on the purpose of the test. Multiple classes or one parameterized dummy class are realized accordingly. For example, `TimingPolicyDummy` can offer a constructor with the list of `time` objects as a parameter, or alternatively, can have stored internally multiple lists, one of which is selected.

8.1.1 Dummies for Layers of the Architecture

Depending on the goal of the test, we can test groups of objects instead of individual objects. For example, we can test the functionality of the server by having both the policy and the person objects coming from the product system and only their environment, such as persistence mechanisms, logging, and Internet connection, replaced by dummies. We can generally group tests for different environments. [Bin99] Thus proposes tests for individual layers of an architecture according to the pattern shown in Fig. 8.4. In each case, one layer is tested and the underlying layer is replaced by dummies (a), (b). For more extensive integration tests (c), we can also couple layers. As a layer itself consists of a group of objects, conversely, we can deconstruct these groups and, as already shown, test individual objects using class and method tests.

If the method to be tested calls other methods of the same object, it can be useful to form a subclass of the class with the test object and in this subclass, replace the methods called with dummy methods. The test object and the dummy are then the same object.

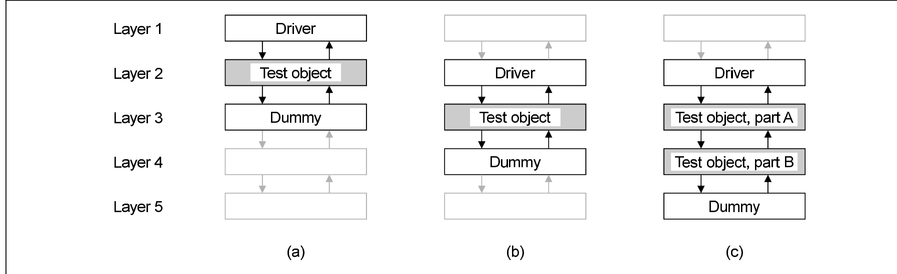


Fig. 8.4. Test environments in the layer architecture

8.1.2 Dummies with a Memory

We can refine the test environment by having it log the form in which it is used by the test object and, where applicable, use this memory to calculate the return results. In the Extreme Programming approach [MFC01], these dummies are also called *mock* objects, and in approaches similar to those for telecommunications, they are called *stubs*. A simple example is shown in Fig. 8.3. We can use the `dummy` object after completion of the test to check how often the method `newCurrentClosingTime` was called. To do this, the counter status is read.

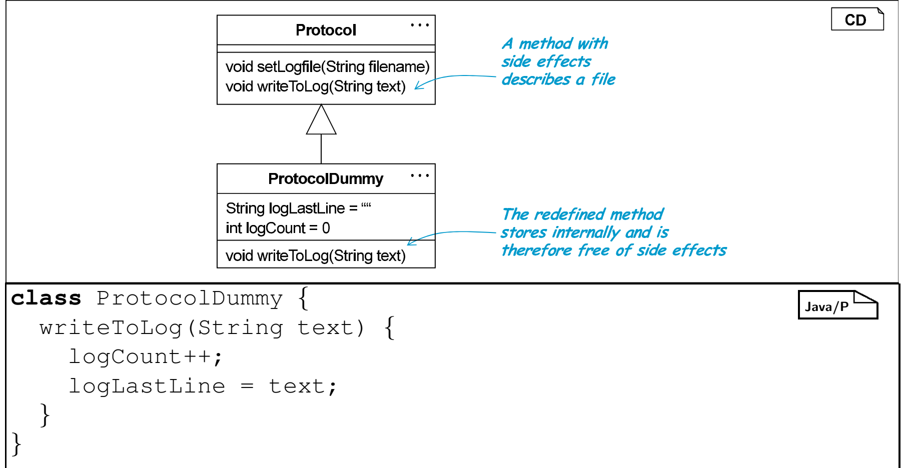


Fig. 8.5. ProtocolDummy allows the capture of side effects

A further example of a dummy with a retrospective query function is a log object that is used to log significant events in the auction system. These include, for example, notifications of the submission of new bids. Therefore, in addition to the actual test data, a dummy object of the class `ProtocolDummy`

is realized. Fig. 8.5 shows the class diagram for a simplified form of the logging. When using the class `ProtocolDummy`, the log can also be tested. To do so, for example, the object diagram `ProtBefore` from Fig. 8.6 can be used to set the test data and `ProtAfter` for the expected result.[2]

The dummy presented here stores only the last line and the total number of messages that have occurred. In some cases, it can be useful to store all messages in a list or to store messages selectively according to certain patterns in a string.

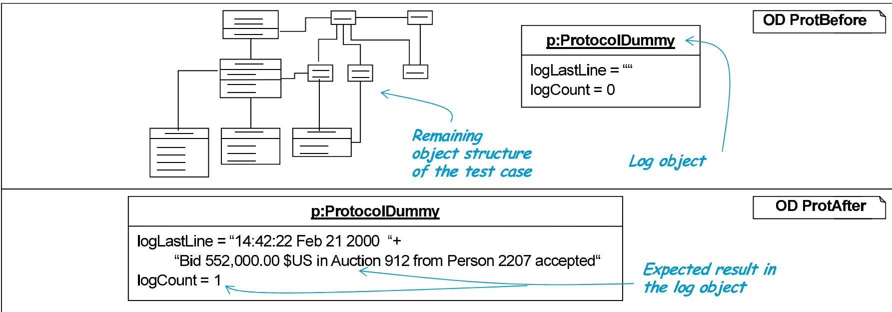


Fig. 8.6. The ProtocolDummy object describes the expected result

8.1.3 Using a Sequence Diagram instead of Memory

To enable us to draw conclusions about the interactions performed during the test run based on the data stored, `ProtocolDummy` requires a memory. Because sequence diagrams are suitable for specifying interactions, we can use them as a replacement for dummies that store data. Fig. 8.7 describes the log aspect of the test execution from Fig. 7.12 by specifying how often and with which arguments the method `writeToLog` is called.

The method `writeToLog` in the dummy for class `Protocol` does not need to store anything. It is sufficient to use an empty body to ensure that no side effects occur. To do this, we use the automatically generated class `ProtocolSimpleDummy`.

Because of the changes that frequently occur when texts are produced, it is often better to require only certain properties instead of a complete specification of the log text. The method call can thus be modeled with `writeToLog(s)` and the string `s` can be checked whether it contains the desired text: `s.indexOf("552,000.00 $US")>=0`.

[2] In practice, an expansion of the class `Protocol` to differentiate between warnings, errors, and notifications as well as the configurability of the output verbosity and the debug level is useful.

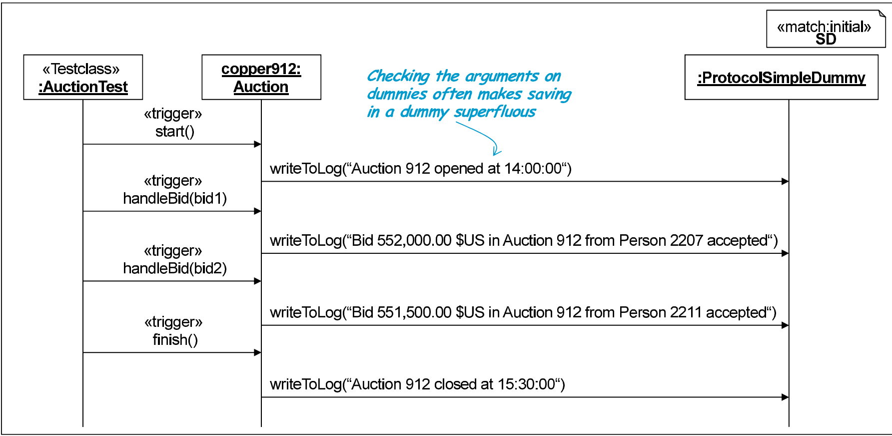


Fig. 8.7. Sequence diagram checks log behavior

In addition to the driver object, the example in Fig. 8.7 contains a dummy. This shows that we can use dummies like normal objects in sequence diagrams and we can thus model the entire interaction in the test run. This does mean, however, that dummies also have to be instrumented by the generator to enable an observation.

8.1.4 Catching Side Effects

The procedure applied in the previous section demonstrates how a test object can be embedded in a test environment from "above" by the test driver and from "below" by dummies. In both cases, the data and interactions relevant for checking the test success are available on all sides. In particular, this allows us to prevent all side effects of the test object that interact with the system environment. Preventing side effects is an essential prerequisite for fast and efficient tests.

However, we also have to test the log class itself. In general, to test the last layer above a file system, a database, or a communication layer, we have to invest a certain amount of effort to ensure a defined initial situation of the environment, to enable access to the test results after the end of the test, and for cleaning up. For example, we have to set up log files or the database accordingly. [LF02] contains detailed descriptions of this scenario. Section 8.5 of this book covers only simulating distributed communication without actually performing the distribution.

8.2 Designing Testable Programs

One of the advantages of object-oriented systems is their improved testability. This is based on forming subclasses of the environment of the tested

objects and by redefining their dynamic methods. Nevertheless, there are still some problems with object-oriented systems that make it more difficult or even impossible to define test environments, to perform tests, or to determine the test success. These problems are generally the result of the use of static attributes and methods, object instantiation, and the use of predefined frameworks or components. We will discuss these problems and how to eliminate them below. To do so, we will use further test patterns with UML diagrams. Interestingly, although polymorphic and dynamic binding increase test complexity, due to the flexibility they provide, they are also important elements for defining test cases.

The following sections propose a number of structural modifications for making a system *testable*. These modifications either enable or at least simplify simulation of a system environment. In addition, the test object is instrumented for the test execution. Both types of modifications change the system. The proposed structural modifications are permanent and therefore require acceptance by the developers. Object-oriented methods tend to support this approach much more than earlier paradigms did. Furthermore, in an agile approach, it is an advantage that the tests are developed before the system is completed and can therefore influence the system structure a priori.

In contrast, the instrumentation of the tests objects is not permanent and therefore requires a great deal of caution, as it must not change the functionality of the test objects. Instrumentations performed correctly by the code generator modify the runtimes at least slightly and thus potentially falsify runtime measurements but not functional tests.

8.2.1 Static Variables and Methods

A frequently recurring problem is the use of static variables and methods. As far as possible, such static elements should not be used. When static elements cannot be avoided, an object that encapsulates the actual variable should be used instead of direct variable access. As an example in the auction project, the singleton object `AllData` shown in Fig. 3.7 was used to obtain access to all auctions, persons, and further data structures. In the UML model, this object is specified to be accessible via the static attribute `AllData.ad` which has the access right `readonly`. During the code generation, a suitable access method is created from this attribute, although it is not a method for modifying the attribute value itself. This means that the singleton is accessible from outside but protected against being replaced. Nevertheless, below we will introduce a pattern that fully encapsulates static variables and allow to access static methods during tests.

The singleton object for logging is stored in the attribute `prot` of the class `Protocol`. As Fig. 8.8 shows, the attribute is also encapsulated in a static method. A message can then be created in the log with:

```
Protocol.writeToLog("message")
```

Java

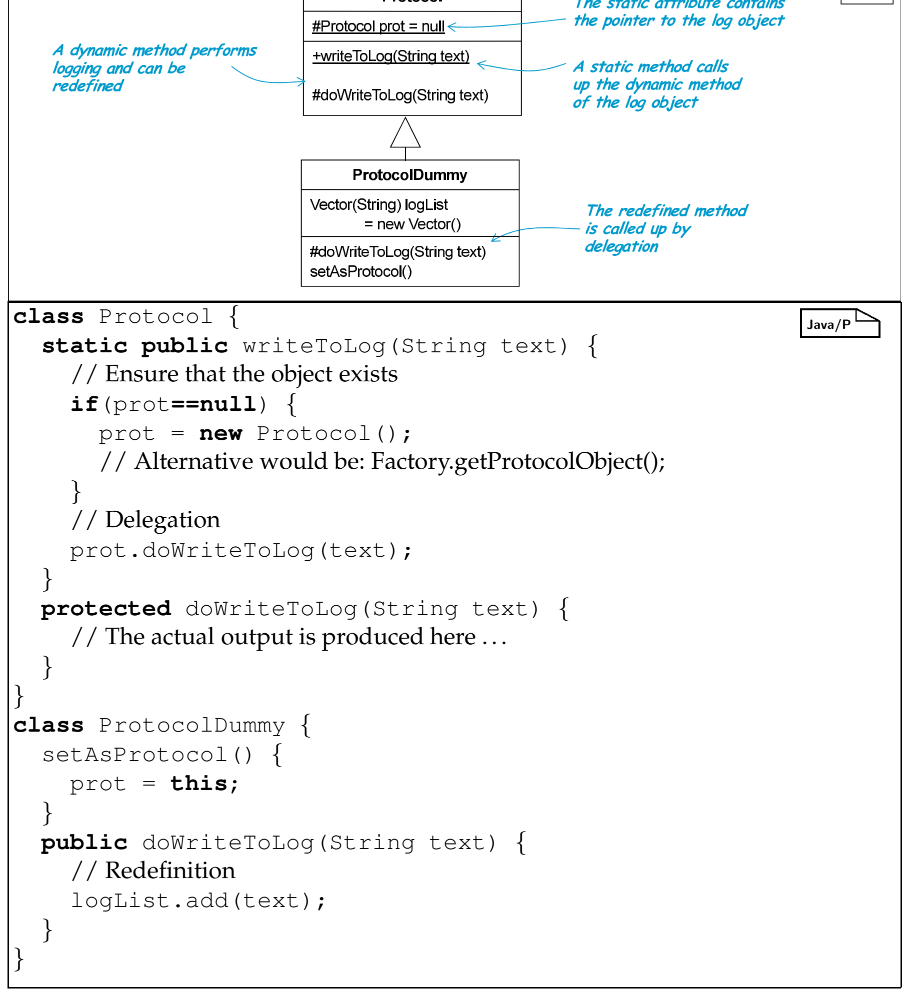


Fig. 8.8. Global log object with replacement option

To ensure that an initial value is assigned to the attribute `prot`, the method checks the static attribute and assigns a value where necessary but then executes only a delegation to this object.

It is now easy to replace the `Protocol` object with a dummy. The method `doWriteToLog` is overwritten in the dummy and the dummy object becomes the log recipient when we set it by using `setAsProtocol()`.

This procedure is summarized in Table 8.9 as a pattern. Furthermore, Section 10.1.5 specifies a refactoring rule that introduces this test pattern into an existing model of the structure of a system.

Pattern: Singleton behind static methods

Intention

On the one hand, the pattern enables compact access to a singleton object which for tests, can be replaced by a dummy; but on the other hand, it avoids a publicly accessible static variable for this object.

Motivation

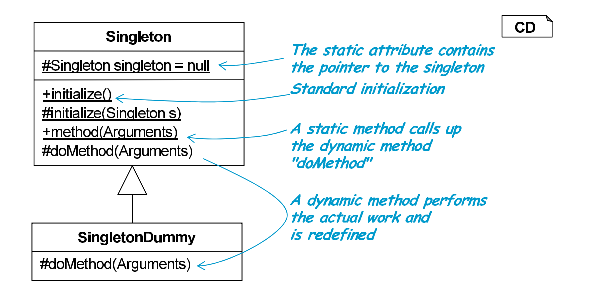
See the previous discussion on the testability of code with static variables. Examples are objects that realize log outputs, time queries, or a factory mechanism.

Application

It is useful to apply this pattern in the following cases:

- There is only one singleton of a class but this is used in many places
- A compact access in the form `Singleton.method()` is desired
- The variable storing the singleton should remain hidden
- The singleton is to be replaced by a dummy in tests

Structure



Singleton implementation

Java/P

```
class Singleton {
  static initialize() {
             initialize(new Singleton( ... )); }
  static initialize(Singleton s) { singleton=s; }

  static method(Arguments) {
    // Separate initialization if necessary
    if(singleton==null) initialize();
    // Delegation:
    return singleton.doMethod(Arguments);
  }

  doMethod(Arguments) {
    // Work is performed here ...
  }
}
```

(continued on the next page)

(continues Table 8.9.: Pattern: Singleton behind static methods)

Dummy implementation	Java/P
	`class SingletonDummy {`
	`  setAsSingleton() { initialize(this); }`
	`  doMethod(Arguments) {`
	`    // Work is simulated here ...`
	`  }`
	`}`

Access	Access to the singleton is via the expression `Singleton.method(`*Arguments*`)`. A prior initialization is not necessary.
Noteworthy	The problem of incomplete initialization is eliminated by an object of the class itself being created by default. A more restrictive form could create an error message here, as experience shows that for tests in particular, adequate assignment of values to the singleton is often overlooked.

Table 8.9. *Pattern:* Singleton behind static methods

8.2.2 Side Effects in Constructors

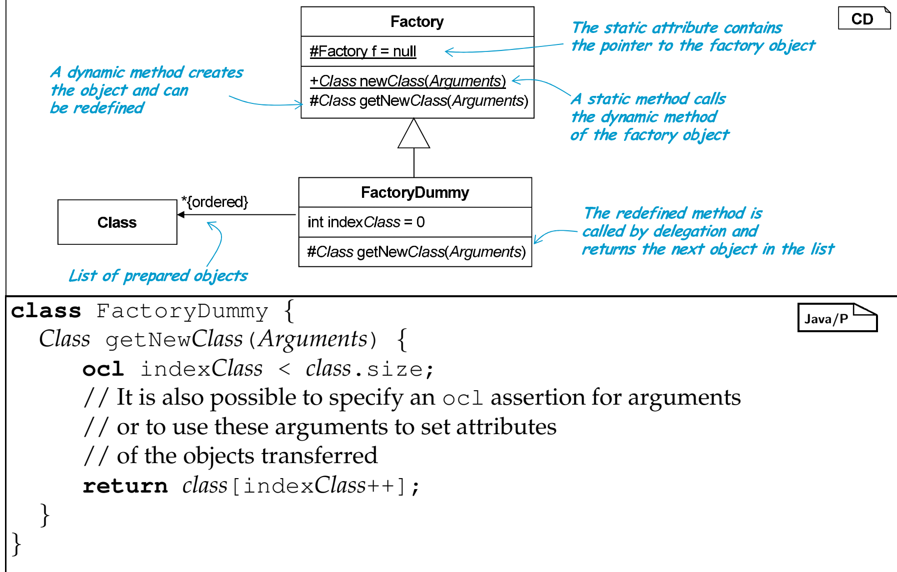
One of the main problems with the procedure shown is that Java requires for objects from subclasses to call a constructor of the superclass. If this constructor causes side effects—in the example it opens the log file—it is no longer possible to define dummies for this class without any side effects. This is why constructors should contain relatively little functionality and, where applicable, additional functions that perform such initializations should be offered. They can be called from the constructors but overwritten in dummy subclasses.

8.2.3 Object Instantiation

A similar problem is the instantiation of new objects. A command in the form `new Class()` in a test object precisely defines the class of the object that is created. In this case, we cannot use a suitable dummy in test runs instead of the object specified. This leads to code that is difficult to test. We can eliminate this problem by using a factory [GHJV94] or a builder in the product code.

As described in Section 5.1.7, we can create a factory from the given UML model by creating corresponding methods of the factory for all constructors that occur. In the same way, the generator replaces all constructor calls in the Java code bodies with factory calls. The factory is itself a singleton that is typically stored in a static variable. We can therefore make it dynamic with the

pattern already used for logs and prepare it for tests with a `FactoryDummy` object. Fig. 8.10 shows an approach that goes even further. The factory that this approach specifies is a dummy object that provides multiple prepared dummy objects for the actual test run.




```
class FactoryDummy {
  Class getNewClass(Arguments) {
      ocl indexClass < class.size;
      // It is also possible to specify an ocl assertion for arguments
      // or to use these arguments to set attributes
      // of the objects transferred
      return class[indexClass++];
  }
}
```

Fig. 8.10. Factory prepares objects of the test run that are to be created

The objects required in the test run are therefore no longer generated during the test and are instead created in advance and then merely transferred. The factory can therefore be initialized by an object diagram as shown in Fig. 8.11, for example. This allows us to determine exactly which classes are used and, if it is useful, which default values should be assigned to the attributes.

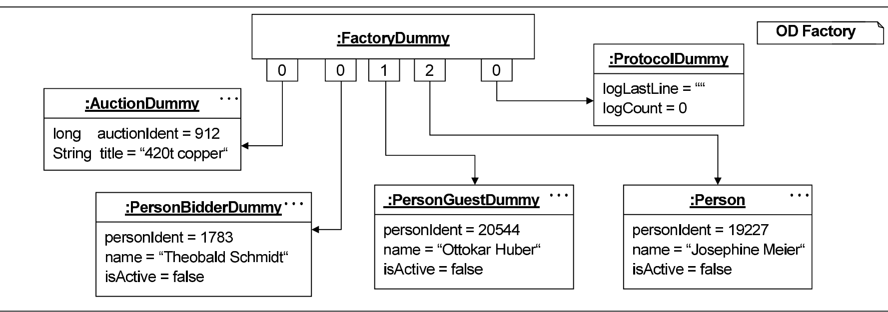


Fig. 8.11. Factory object prepares objects of the test run that are to be created

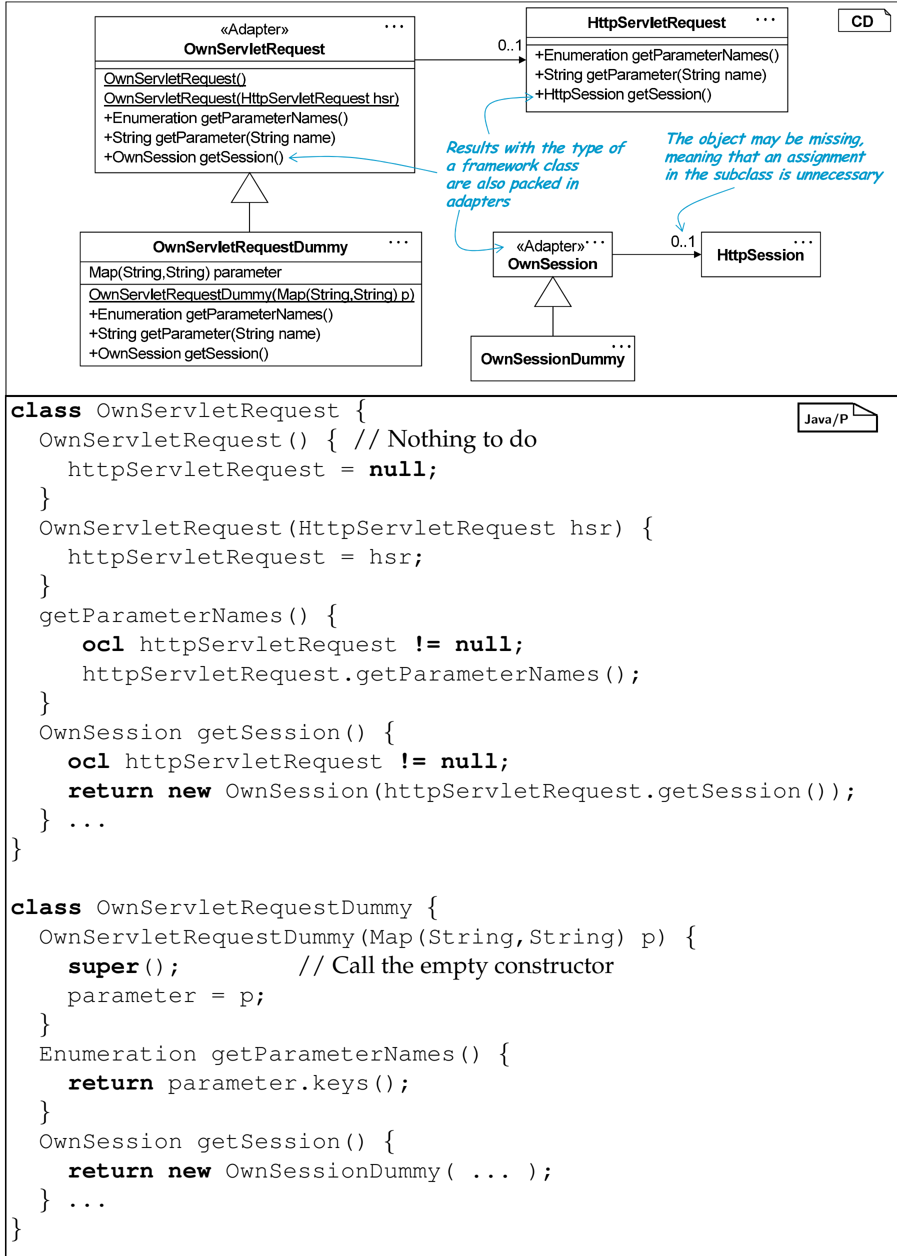
8.2.4 Predefined Frameworks and Components

Unfortunately, we cannot apply the transformations described in the previous sections to use predefined frameworks, class libraries, or components, because these cannot be modified. The goal of a test here is not the predefined frameworks themselves, but rather the classes developed whose functionality builds on the frameworks and therefore requires parts of the framework in the test environment. For example, according to [LF02], it is difficult to integrate Enterprise JavaBeans (EJB) [MH00] in tests. There are generally several potential reasons for this:

- The control flow can be defined by the framework. The "Don't call us, we'll call you" principle [FPR01], which is common in frameworks, only allows the test to take over control with a great deal of effort.
- The creation of new objects is already fixed in the framework; injection of a factory is not possible.
- Static variables, particularly if they are encapsulated, cannot be controlled sufficiently in a test and cannot be assigned suitable values.
- Encapsulated object states do not allow access for evaluating the test success.
- No subclasses of the classes available can be formed, and therefore no dummies, because (1) the class or a method it contains is declared as `final`, (2) there is no public constructor, (3) constructors have undesirable side effects, or (4) the internal control flow is unknown.
- The classes cannot be instrumented, which means, for example, that the information required for testing invariants and sequence diagrams is not accessible.

In order to still be able to test software despite all of the issues listed above, the application logic must be separated from such frameworks or components. We can generally use the *adapter* design pattern [GHJV94] to do this. [SD00] describes this separation as important to allow the application logic to be reused independently from the technical code, but also for improving maintainability. A further positive effect of this separation is the improved testability. Fig. 8.12 shows an adapter for Java Server Pages (JSP). The class diagram describes a separation of the processing of datasets entered via the web and the actual storage in `HttpServletRequest` objects provided by the JSP [FK00]. These request objects contain the data entered by the user via a web form and can be queried, for example, via the list of parameters using `getParameterNames` and the reading out of individual parameter values using `getParameter`. Further methods, such as `getSession`, deliver the context of the session to which the web form belongs, for example.

The interaction mechanism, which JSP needs to read the input data, for example, makes the complex adaptation necessary. Where applicable, parameters and results have to be repacked and unpacked again respectively.




```
class OwnServletRequest {
  OwnServletRequest() { // Nothing to do
    httpServletRequest = null;
  }
  OwnServletRequest(HttpServletRequest hsr) {
    httpServletRequest = hsr;
  }
  getParameterNames() {
     ocl httpServletRequest != null;
     httpServletRequest.getParameterNames();
  }
  OwnSession getSession() {
    ocl httpServletRequest != null;
    return new OwnSession(httpServletRequest.getSession());
  } ...
}

class OwnServletRequestDummy {
  OwnServletRequestDummy(Map(String,String) p) {
    super();           // Call the empty constructor
    parameter = p;
  }
  Enumeration getParameterNames() {
    return parameter.keys();
  }
  OwnSession getSession() {
    return new OwnSessionDummy( ... );
  } ...
}
```

Fig. 8.12. Adapter for `Request` objects

In the auction system, this mechanism was used to separate the JSP interface from the application core.[3]

[3] Information about whether an `OwnSession` object has already been assigned to a `Session` object is also stored. This is necessary to ensure that a session is unambiguous in the application.

There are two primary variants for separating the developed code from used frameworks and components. Firstly, we can offer a complete collection of adapters for all classes of the framework. Or secondly, we can create a minimal version of the classes currently required and the methods of these classes used.

The minimal version corresponds to the idea that as little effort as possible should be invested in such technical definitions; however, it does have the disadvantage that due to the necessity for further methods, the adapter layer has to be expanded iteratively. On the other hand, this restriction also has the advantage of making it easier to adapt a software system that uses this framework to a new version of the framework and to migrate it to another framework.

In contrast, a complete adapter layer has the advantage of greater reusability. Unfortunately, as the method `getSession` shows, this adapter layer cannot be generated fully automatically. It is therefore an advantage if the framework itself already has such adapters or is encapsulated by interfaces and factory objects such that dummy objects can be used directly. Ideally, the framework will also have a number of dummy classes in an additional package which can be used for various test purposes. This would simplify developing tests in framework-dependent projects.

Conversely, when publishing a component or a framework, it is useful to also publish a test suite that demonstrates that the component or the framework behaves in accordance with a given specification. This also increases the confidence of anyone using the component and explains the application to the users with examples.

8.3 Handling of Time

In distributed real-time systems, the continuous progression of time is important. In the auction system, for example, the current time is used to decide the state to which an auction should proceed, whether a bid is received or rejected, and whether notifications are sent to the bidders.

In Java, we can use `System.currentTimeMillis()` to determine the current time in milliseconds. Using this within a test, times and time differences vary, which influences the test result and makes it dependent on the execution time of the test. For example, the log output differs depending on the period in which a test is executed.

Including the current time means that the determinism of the test run required in Chapter 6 is lost. We therefore have to define the test driver to control the time prevailing during the test run. As a very helpful side effect, tests that extend over several hours in a real time execution, such as an entire auction, can be executed effectively in a few milliseconds.

8.3.1 Simulating Time in a Dummy

Due to the afore-mentioned considerations, we locate the method call requesting the current time in a separate class and control it via a dummy object that can be preconfigured. Fig. 8.13 describes this construction with excerpts from the dummy class. In the auction project, the pattern for singletons behind static methods from Table 8.9 was also applied.

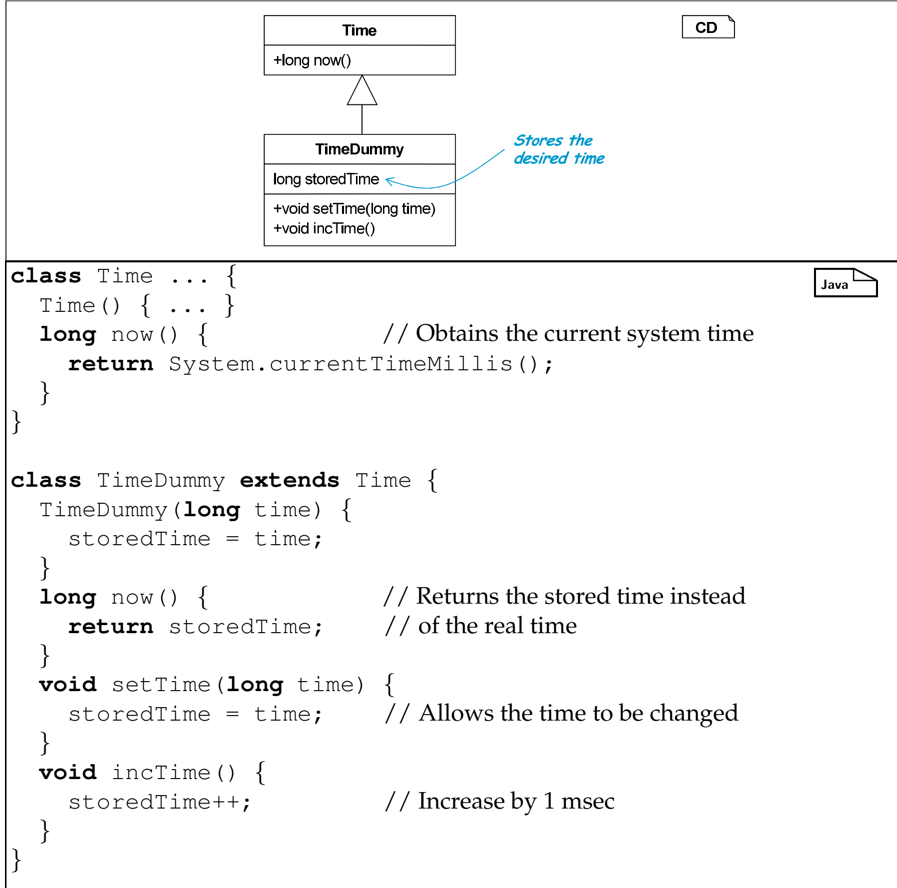


Fig. 8.13. Simulation of time with `TimeDummy`

The function `setTime` allows test drivers to set the time arbitrarily. The time remains constant during the execution of a test, however. This corresponds to an idealizing assumption that no time actually passes during the calculation of a reaction. This assumption is applied in languages for embedded systems such as Esterel [Hal93], for example. In fact, the proposed test pattern requires some restrictions on the code with regards to the use of the

requested time. Ideally, only one time query that is then used as the reference time during the calculation should take place. If it is necessary for time to progress within the body of a method, a progression can be simulated by calling `incTime()` for each query of `now()`.

8.3.2 A Variable Time Setting in a Sequence Diagram

In parallel to the use of the time simulation, we can use a sequence diagram to adjust the clock each time the test object is called. For example, we can extend the sequence diagram from Fig. 7.12 such that the time is set explicitly in each case. This produces the sequence diagram shown in Fig. 8.14.

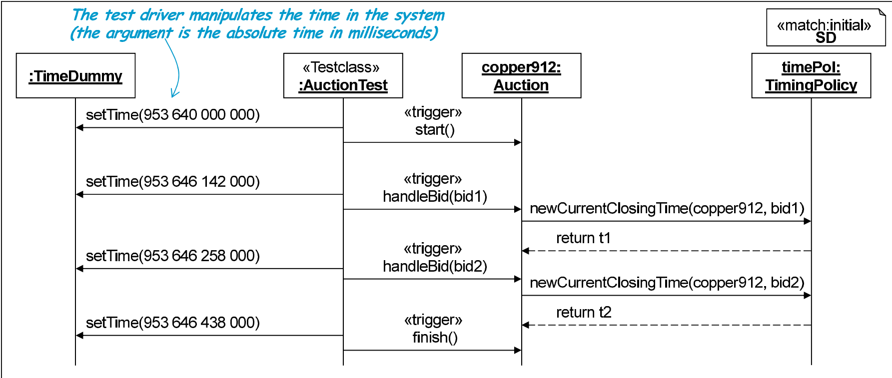


Fig. 8.14. Assignment of the time for each call of the test object

We can use tags for method calls to model these time specifications more compactly. In a test system, these time-based tags are interpreted constructively: they are not measured, and are instead used as a specification for the respectively valid system time. This allows us to represent a form of the auction run annotated with system times as shown in Fig. 8.15.

At the same time, the instrumentation of the test object for observing method calls and returns allows the respective current time to be adapted. Table 8.16 describes the application of the tag `{time}` and its additive form `{time+}`.

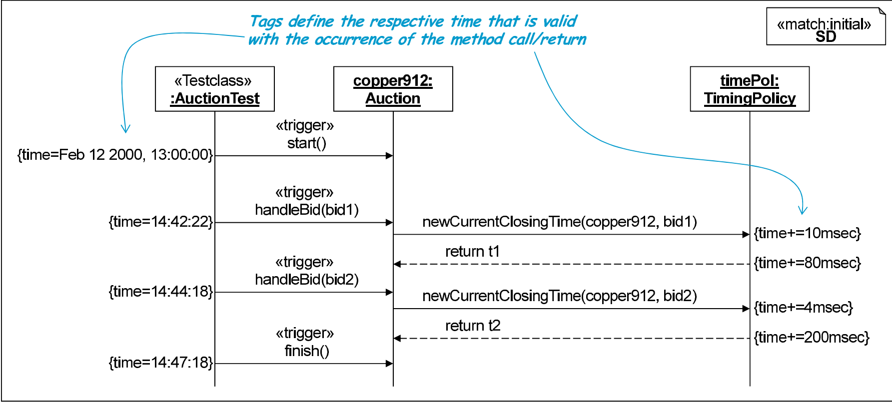


Fig. 8.15. Annotation as time specification

Tag {time}	
Model element	Interactions in a sequence diagram. SD oneObject furtherObject {time=13:00:00} method() {time=Feb 21 2000,13:00:00} return {time+=2h 7min 6sec} method() {time+=13msec} return
Motivation	In a sequence diagram, the tag {time} specifies at which time the call of a method or a return happens. This allows the system time to be simulated.
Usage condition	To implement the time specification, the sequence diagram must be used as a test driver in an environment with simulated time. The times must increase as the timeline descends. In a constructive sequence diagram in the interpretation with the stereotype ≪match:free≫, the specification of times is prohibited.
Effect	The instrumented product code sets the respective specified time when the corresponding call or return takes place. As defined in the pattern in Table 8.17 for simulating time, this time remains constant until the next tag.

(continued on the next page)

(continues Table 8.16.: Tag {time})

	The tag {`time`} allows different date and time formats as arguments that apply for the description of the time to be set. If the date is missing, the tag that is already valid is retained. It is possible to not only specify constant values, but also to include variable and attribute values in calculations. The variant {`time+`} allows relative time specifications that can also be given in different formats.
Example	Fig. 8.15 uses these tags.

Table 8.16. Tag {`time`}

8.3.3 Patterns for Simulating Time

The pattern discussed for handling the time problem is summarized in Table 8.17.

Pattern: Simulation of time	
Intention	To ensure deterministic results for tests of time-dependent behavior, the time is simulated. The driver can manipulate the time during the test run.
Motivation	Time-dependent behavior can thus be simulated and is repeatably deterministic.
Application	It is useful to apply this pattern if the system behavior is dependent on the current time, as is the case, for example, when logging operations or for tests of interaction patterns that have timeouts.
Structure	See Fig. 8.13. To apply the pattern, sequence diagrams such as those shown in Fig. 8.15 are available, for example. They use tags to define the current time.
Implementation	See Fig. 8.13. This is typically combined with the pattern described in Table 8.9 for encapsulating a singleton behind static methods so that `Time.now()` can be used as a call.
Noteworthy	An implementation must not assume that the time progresses during its activity. In particular, no waiting loops in the form `t=Time.now(); while(Time.now()<t+1000);` may be used.

Table 8.17. *Pattern:* Simulation of time

A refinement of this concept was used in the auction project: in test runs, several clients and the server run together in a single process space. As clients have different system times, and signal delays in the Internet should not be

underestimated, for each client a separate time was simulated by exchanging the `Time` objects each time the activity switched between clients. The pattern 8.25 discussed in Section 8.5 was used to do this.

This allows us to model relativistic phenomena and to test, for example, phenomena of the software in the case of (often occurring) insufficient time synchronization between clients and the server. We need to do this because one of the critical problems in the Internet results from the significant and, to some extent, very different signal runtimes. These become significant in a real-time application, such as the auction system.

8.3.4 Timers

Monitoring of timeouts is closely related to querying the current time. Both can generally be realized with the same techniques. Typically, new timers are created for each task, which is why a timer factory is realized as a global singleton. Timeouts can thus be produced or prevented as desired and transmitted by timers or directly by the test driver.

8.4 Concurrency with Threads

Concurrency always occurs when system parts that act independent of each another can perform activities at the same time [Bro98]. For example, multiple threads within a system can execute different tasks. Typically, external events such as user input, TCP/IP communication, and printing and loading operations are handled in separate threads. Thus in the Abstract Window Toolkit (AWT), events are processed in a separate thread. This does normally not increase the overall speed of the system but does use waiting times caused by the environment efficiently and improves the response behavior of the graphical user interface. Threads are lightweight processes that all run in the same memory space. In contrast, there are heavyweight processes that the operating system manages and that do not use any common memory. Thirdly, processes can be distributed over different processors via the Internet, for example.

Concurrency occurs in all of these cases and can lead to nondeterministic results and thus also to sporadic errors. It is therefore important for conformance tests to suppress this form of nondeterminism and execute tests in all variations of interactions. In the following, we will discuss a concept that allows us to control the nondeterminism arising from threads within a process space.

The principle behind this approach for testing distributed systems can be characterized as follows: the basic functionality is initially tested using a deterministic test run to ensure that the methods do work together correctly. The test run brings all of the activities required for the test into a deterministic order suitable for the test. This now acts as initial proof of existence of

correct test runs. Naturally, it does not allow a statement that is valid for all tests. Therefore, alternative orders are checked in further tests. This *interleaving at method level* creates further security about the correctness of the concurrent interaction. The tests, however, do not ensure absence of undesired *interactions between threads on shared data*; this is ensured instead by the adequate use of synchronization mechanisms. In high-level languages such as Java, the intensive use of synchronization means that the problems coming from concurrency are significantly less than in hardware-based, embedded systems, for example, where interleaving occurs at machine instruction level.

The technique proposed in [LF02], for example, of allowing nondeterministic test cases to run several times to test different variants of executions can be seen as a supplement. [Bin99] also offers a test pattern for integrating distributed systems. However, this test pattern does not contain any functional simulation within a process space and should therefore also be seen as a supplement.

8.4.1 Separate Scheduling

Deterministic test results can generally only be achieved if the concurrency is fully controlled by the test driver. To do this, a test driver must be able to either intervene in the scheduler of the Java virtual machine or switch off its scheduling. As both the product code and the test code should be able to function on several platforms or at least with different versions of Java implementations, an adaptation of the Java virtual machine is only useful to a limited extent.

However, a relatively elegant and stable solution is to model the scheduling in a simple form directly in the test driver. As a test driver is formulated for only one test run, it is relatively easy to describe under the following assumptions:

1. Each thread consists of one or more regularly recurring activities which each require relatively little time.
2. The parallel execution of individual activities has no interferences that are explicitly required for the program execution.[4]
3. Thread security and deadlock freedom are already tested with other test and inspection procedures.

According to 1., a thread therefore has a form similar to that shown in Fig. 8.18.

The use of other mechanisms, such as Java´s `TimerTask` and the associated possibility of defining an explicit scheduling for method calls, does

[4] For example, a data exchange from threads via common variables or busy-wait loops would be problematic.

```java
class OwnThread extends Thread {
  protected Workingclass client;
  public OwnThread(Workingclass client) {
    this.client = client;
    // The thread is not started directly when the thread is created,
    // but afterwards by the object creating it!
  }
  public void run() {
    // Constant repetition:
    while (true) {
      client.workingmethod(arguments);
      try {
        sleep(sleepPeriod);
      } catch (SomeException e) { }
    }
  }
}
```

Fig. 8.18. Typical appearance of a testable thread

not change this principle. Instead, the use of this mechanism actually also requires the separation of thread management and application functionality. If necessary, the product code must be refactored by, for example, outsourcing the functionality hidden in the method `run` into a separate method. If the calculation of a termination condition for the `while` loop or the `sleepPeriod` is more complex, these two calculations are also outsourced to separate methods. This allows us to test these individual functionalities and the basic functionality of the thread is simple and therefore straightforward.

This principle can also be achieved if the actual thread is hidden in an external component or in a framework and only *callbacks*, that is, calls from the framework to the own code developed take place. In some circumstances, however, the arguments transferred have to be packed in adapters. This ensures that there are no dependencies between the application code and the framework that prevent the application code being embedded in a test system. This is demonstrated in Section 8.2 using the example of the `Request` classes from JSP.

Another set of problems results from the fact that there are threads that can be structured according to Fig. 8.18 but the methods regularly called have to perform *blocking calls* themselves. For example, the communication via `Socket` classes in Java retrieves incoming data using a blocking `read()` call. A `Socket` dummy that already has a set of input data can be used to bypass this blockade.

8.4.2 Sequence Diagrams as Scheduling Models

Based on the thread structured according to Fig. 8.18, it is feasible to model a test scheduling with a sequence diagram. Fig. 8.19 describes a test driver that executes multiple methods whose calls to the auction system take place in different threads.

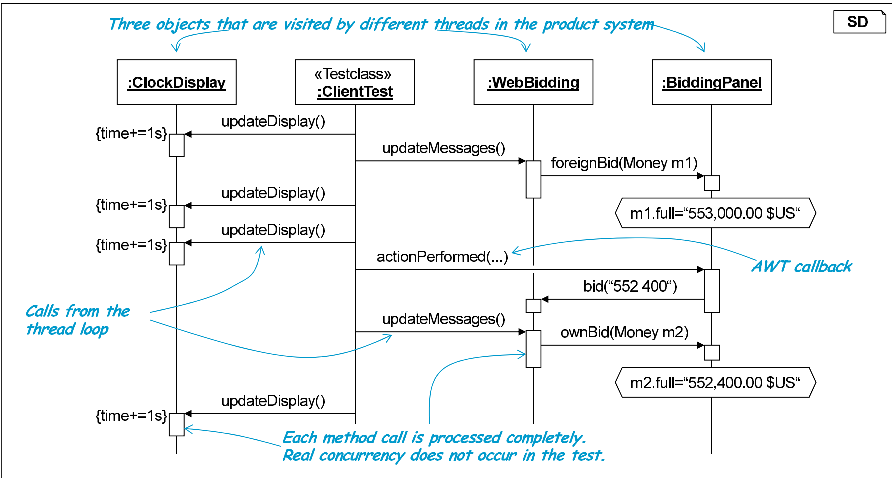


Fig. 8.19. Scheduling at the level of individual functions

The three objects of the classes `ClockDisplay`, `WebBidding`, and `BiddingPanel` belong to three different threads. The first is responsible for updating the time display in the Client GUI; the second is responsible for regularly requesting new information from the server for the auctions currently being observed; and the third is part of the graphical interface that waits for user actions. The object `:BiddingPanel` is therefore addressed by callbacks from the AWT framework. The example tests a large part of the client system as it integrates not only the application core, but also the processing of the graphical interface and the communication with the server. Dummies are therefore required at multiple points to simulate the effects of the environment used. In particular, using AWT classes to transfer events is critical for the reasons described in Section 8.2. It is therefore advisable to test one layer lower or to place an adapter layer between the AWT and the application code. In the example, pressing the return key in the input field for bids is interpreted as a bid submission and leads to `bid(String inputtext)` being called. This method can also be called up directly and therefore independently of the AWT framework. Fig. 8.20 shows one of multiple scenarios used in the auction system to test the application core in the client, whereby again there is no complete definition of the underlying object structure.

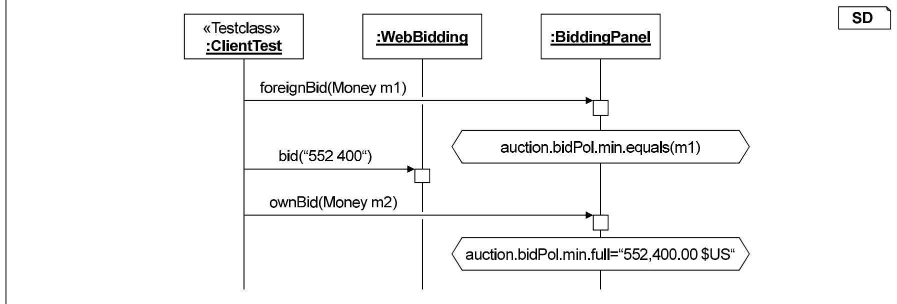


Fig. 8.20. Scheduling solely in the application core

8.4.3 Handling Threads

The previous modeling of concurrency assumes that the respective threads already exist and act independently of one another.

However, there are several situations in which threads influence each other mutually. For example, a new thread can be created at any time. As the test driver has to take over the scheduling of the threads completely, the creation of a new thread must be simulated by applying a factory for creating new objects and a dummy thread that does not lead to a new thread actually starting. However, the simulation of a forced termination or interruption of another thread is then also not a problem. Therefore, in a dummy class for threads, methods such as `start()`, `interrupt()`, or `destroy()` can be replaced by empty methods.

If the method `join()` is used, however, the scheduling procedure used must be extended such that the `join()` continues to call methods of other threads until the threads there can be deemed finished. Like other methods, the method `join()` also cannot be redefined in the class `Thread`. Therefore, an adapter is required. A class `OwnThread` can be formed according to the pattern in Section 8.2.4. `ThreadDummy` is the subclass from this pattern that is used for the test.

Using a suitable code generator, from the sequence diagram in Fig. 8.21, we can generate a test case that partially integrates the functionality of the test driver in the class `ThreadDummy`.

Methods such as `sleep()` should be redefined in a test to simulate time in the thread dummy without actually waiting.

8.4.4 A Pattern for Handling Threads

The discussion for handling threads can be summarized with the pattern presented in Table 8.22.

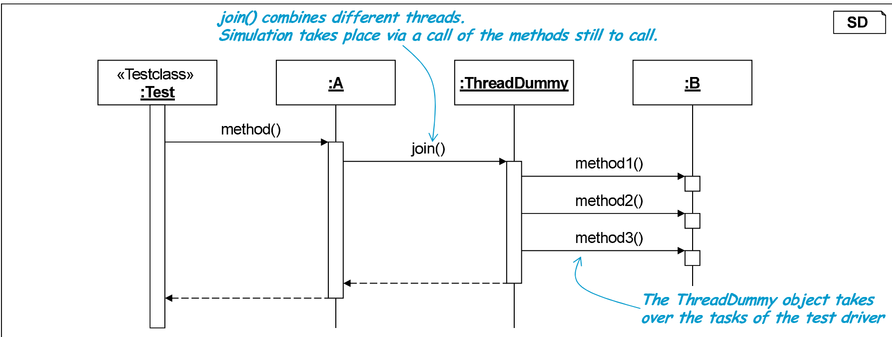


Fig. 8.21. Thread dummy takes over scheduling tasks

Pattern: Handling threads	
Intention	Threads can generally cause nondeterministic system executions and must therefore be suitably adapted for tests.
Motivation	The processing of threads and therefore of concurrency within a process must be simulated with a distinct scheduler that produces determined test results.
Application	It is useful to apply this pattern in the following cases: • Multiple threads run in parallel within one process. • There are interactions between the threads that need to be tested. • The threads are defined according to the structure described in Fig. 8.18 or can be restructured accordingly; this involves a thread performing regularly recurring, relatively short activities.
Structure	The pattern consists of two parts. A `ThreadDummy` is used if threads are controlled explicitly in the test object. `ThreadDummy` is the subclass of an adapter for the class `Thread`. In the product system the adapter is used. Threads that run in parallel and that are assumed to be initialized are tested via a scheduler formulated for the test run. This scheduler calls the individual activities of the different threads in order and allows each of these activities to run to completion before the next activity is executed. A type of cooperative multitasking is thus implemented.
Implementation	We can model this type of scheduler using a sequence diagram such as the one shown in Fig. 8.19. If this pattern is used in combination with the simulation of time from Table 8.17, we can use the tag {`time`} to model the advancing time.

(continued on the next page)

(continues Table 8.22.: Pattern: Handling threads)

Noteworthy	If a method called by the scheduling uses special functions such as `yield()`, `sleep()`, etc., these must be redefined suitably in a `ThreadDummy`.

Table 8.22. *Pattern:* Handling threads

This pattern describes a first part that has to be generated and a structural, reusable second part. The latter belongs to the runtime environment of UML/P which provides adapter `OwnThread` and its subclass `ThreadDummy` as a standard. As described, the subclass ignores all method calls. Separate subclasses can be defined from this class that perform the scheduling described in Fig. 8.21, for example.

A generator that is suitable for thread scheduling can generate test drivers (from sequence diagrams) that take over the scheduling as subclasses of `ThreadDummy`.

8.4.5 The Problems of Forcing Sequential Tests

We remember that the method described here for scheduling tests does not test the concurrency but rather excludes it explicitly in order to run sequentialized conformance tests. The test results do therefore not reflect all situations in a concurrent product system. These test runs are only some possible executions. However, if each critical method is synchronized, as required in the Java coding standard, then at least fine grained concurrency can be prevented. This simplifies the development of tests as well as to cover all potential behavior with these tests.

The following piece of code belongs to a program that is difficult to test and that permits "race conditions":

Java

```
class X {
  int a = 0;
  int loopa() {
    for(int i=0; i<100000; i++) a = (a+1) % 99;
  }
  int setAndReada() {
    a = 0;
    return a;
  }
}
```

With a scheduling that calls the methods in a random order but one after the other, the following OCL constraint is always tested successfully:

OCL

```
context X.setAndReada()
pre:                true
post SetAndReadA:   result==0
```

The reason for this is that `setAndReada()` runs with the exclusion of true parallelism. However, if real concurrency occurs in a product system, the result of `setAndReada()` is not determined. There are two ways of addressing this: (1) The constraint `SetAndReadA` is defined too narrowly. Thus `0<=result && result<99` must be used instead. (2) The methods are implemented incorrectly; they should be protected against such surprises by means of synchronization.

Simulating the unprotected functions with the proposed procedure means that we have to reorganize the parallel functions to allow a more detailed scheduling. For example, we can split the method `setAndReada()` into two parts (one modifying method and one query) and move the body of `loopa()` to a separate method. This results in the methods that, provided a test driver in the form `seta(); loopaBody(); reada();`, immediately cause a recognizable violation of the invariant:

Java

```
class X {
  int a = 0;
  int loopa() {
    for(int i=0; i<100000; i++) loopaBody();
  }
  int  loopaBody() { a = (a+1) % 99; }
  void seta()      { a = 0; }
  int  reada()     { return a; }
}
```

An alternative approach could use a scheduling in which running functions interrupt themselves with an appropriate method call.[5] However, this does require a greater level of effort to manage the scheduling.

The biggest problem is still that the number of quasi-parallel executions increases significantly with this very detailed scheduling. Developers have to anticipate the potential sources of danger accordingly so that these can be anticipated through specific tests. On the other hand, this does allow to test unclear or suspectedly erroneous situations.

The literature available on the topic of testing contains various proposals for testing concurrent programs. It is a best practice to make programs *thread-safe* and establishing a type of *pseudo-determinism* [LF02] by making concurrent program parts as independent as possible. Another option is to run as many automated test runs as possible with the real threads to thus capture at least sporadically occurring errors [LF02]. However, the confidence in the robustness of the system remains limited even with this method, especially, if the product system has a different hardware, operating system, compiler version, system load, etc. as the test system.

[5] The first implementations of the Java virtual machine required this, for example, through calls of `yield()`, as they only realized a cooperative multitasking. A similar principle would have to be applied here.

8.5 Distribution and Communication

Truly distributed programs differ from only concurrent programs by a spatial or at least conceptional distribution of their subsystems. This results in separate storage that prevent more than one process working on the same object and that enforce explicit communication forms [Bro98, Bog99].

Systems, which are distributed in the Internet, such as the auction system, can communicate in various ways. Several frameworks and technologies, such as RMI, Rest, or CORBA, offer different levels of support. Increasingly many frameworks even allow to use synchronous method calls that are then encoded as asynchronous communication via the Internet.

8.5.1 Simulating the Distribution

Distributed systems are also concurrent systems and are therefore inherently nondeterministic, unless a fully deterministic communication and scheduling is explicitly added. Furthermore, in contrast to the usual thread programming, a communication partner can fail relatively often because, for example, WLAN is fragile, the computer is turned off, or the client was terminated. State-based communication, such as protocols, where the state of the communication is remembered by the partners, tests must therefore also integrate spontaneous state transitions to mimic *protocol resets*.

The principle that can generally be used for testing distributed systems is based on the scheduling of concurrent threads. The pattern for simulating concurrency from Table 8.22 is therefore also generally suitable for simulating distributed systems. However, some additional problems must be considered.

Threads that are designed for different process spaces can interact in an undesired way in a test within one joint process space, for example, because all threads suddenly share the same static variables in a test run. This problem can be overcome by exchanging values of the static variables on each change of the thread context. It is even better to encapsulate static access in an interface that allows internal redirection. The singleton pattern from Table 8.9 is ideally suited to this purpose. In this pattern, the realization of the `do`...method delegates to the object that resembles the currently active context. This concept is similar to the use of `Session` objects in JSP [FK00], which define the context of a thread that is currently active. However, in the approach presented here, the use is not visible to a user due to the additional encapsulation in a static method.

Fig. 8.23 shows a simple scheduler that stores a message on a server and then triggers two clients to retrieve the message.

This sequence diagram does not contain the actual communication. This is dealt with later below. The objects involved in the test are parameterized with the tag {`location`} introduced in Table 8.24. The value of the tag describes the respective process space. Before the respective method call is exe-

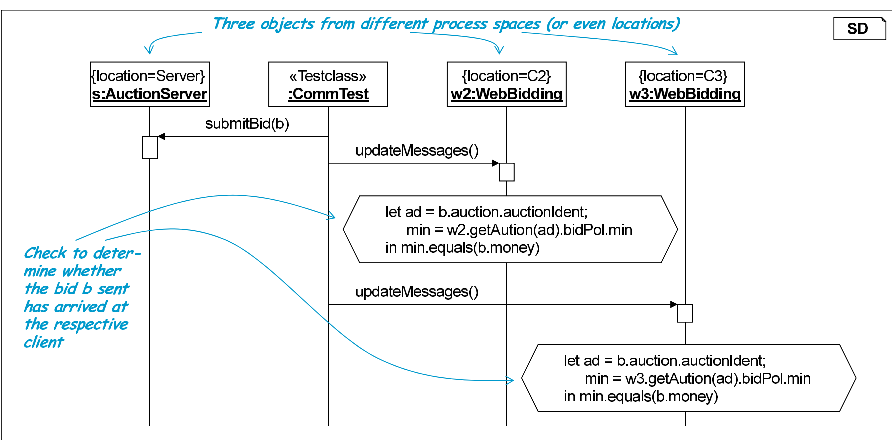


Fig. 8.23. Simulation of a distributed system

cuted, the process context is switched such that each activity triggered finds its natural environment.

The implementation of location specifications is a very pragmatic form of the use of locations for modeling systems. Locations are also used in the ambient calculus [CG98] for modeling dynamic systems. In the UML standard, the deployment diagrams, which are not examined in detail in this book, can also be used to describe similar effects.

Tag {`location`}	
Model element	Objects in an object diagram and a sequence diagram. {location=Server} oneObject {location=Client} furtherObject SD *and* OD
Motivation	Describes the physical or conceptual distribution of the marked objects in different process spaces.
Glossary	The position specification described with the tag is called *location*.
Usage condition	As process spaces are distributed physically or conceptually, links and calls between objects in different process spaces are usually not possible. However, an exception is made for test drivers and communication elements. Objects that are created as a result of a sequence diagram belong to the same process space as the creating object.

(continued on the next page)

(continues Table 8.24.: Tag {location})

Effect	The physical distribution can be simulated in a test case. Each location has access to a separate environment that prevents, for example, the use of static variables leading to undesired interactions. The argument for the tag {`location`} is an identifier of the type `String` which represents the name of the process space.
Example(s)	The sequence diagram in Fig. 8.23 uses these tags.

Table 8.24. Tag {`location`}

8.5.2 Simulating a Singleton

Simulating a distributed real-time system usually requires the use of different system times for each individual location. For this purpose, the pattern in Table 8.17 for simulating time can be extended using the same technique as for other singletons. The pattern in Table 8.25 describes how a singleton can be managed individually for each location.

Pattern: Individual singleton for each location	
Intention	If distribution is simulated in the test system, this pattern enables a separate singleton to be assigned to each location. The pattern builds on the singleton pattern from Table 8.9.
Application	When this pattern is applied, the fact that a distribution is only simulated remains hidden to the product code. From a virtual perspective, each location has its own static variables.
Singleton implementation	(code below, Java/P)

```
class Singleton {
  static String location = "";
  static Map(String,Singleton) singletonMap
                = new HashMap();

  static public void setLocation(String l)
                { location = l; }

  static public void initialize(Singleton s) {
    singletonMap.put(location,s);
  }
  static public void initialize() {
    initialize(new Singleton(...));
  }
```

(continued on the next page)

(continues Table 8.25.: Pattern: Individual singleton for each location)

<table>
<tr><td></td><td>

```
Java/P
  static method(Arguments) {
    // Selection of the singleton
    Singleton singleton
                  = singletonMap.get(location);
    // Separate initialization if necessary
    if(singleton==null) {
        initialize();
        singleton = singletonMap.get(location);
    }
    // Delegation:
    return singleton.doMethod(Arguments);
  }
  doMethod(Arguments) {
    // Work is performed here ...
  }
}
```

For the simulation, only the class `Singleton` is adapted using subclasses like `SingletonDummy`, which are defined precisely as they are described in Table 8.9.</td></tr>
<tr><td>Noteworthy</td><td>• As multiple "singletons" are used in the same process space of the test, we can only refer to virtual singletons. In fact, multiple instances exist simultaneously.
• It is therefore important for the simulation that the singleton encapsulated by this technique also only accesses its static attributes using the mechanism presented here.</td></tr>
</table>

Table 8.25. *Pattern:* Individual singleton for each location

8.5.3 OCL Constraints across Several Locations

As a result of the combination of multiple virtual process spaces used in the test, the singleton is only unique within its location. As a result, for example, the auction identifier `auctionIdent` is not unique across multiple locations. Therefore, when simulating distribution, OCL invariants have to be interpreted differently to the form of Chapter 3, Volume 1. In principle, expressions such as the following are valid for each location but not for the entire test case:

```
OCL
context AllData ad inv AllDataIsSingleton:
  AllData == {ad}
```

or

```
OCL
context Auction a, Auction b inv AuctionIdentIsUnique:
  a.auctionIdent == b.auctionIdent implies a == b
```

Such expressions are not valid for the entire test case because they implicitly use quantifiers over the set of all available objects.[6] Therefore, the OCL constraints used in such test cases have to be interpreted locally on all objects of a class. On the one hand, in tests the management of the extension of a class, which is created by the code generation, requires an additional differentiation according to the locations to which these objects reside.

On the other hand, as the OCL constraints in Fig. 8.23 show, when simulating distributed processes, it is interesting to evaluate constraints that are formulated across process boundaries. This allows us to formulate global properties: for example, that after the messages present on the server have been retrieved, the value of the last bid `b` is present on the client as the lowest bid. The `Money` objects compared in this process belong to different locations.

If it is not obvious that an OCL constraint is to be interpreted globally, the use of a tag in the form {`global`} is proposed for OCL invariants. In the same way, we can restrict the validity of an OCL invariant at each of the locations individually using the tag {`local`} and at a specific location only by naming these explicitly:

```
{location=server}                                                    OCL
context Auction a, Auction b inv AuctionIdentIsUnique:
  a.auctionIdent == b.auctionIdent implies a == b
```

8.5.4 Communication Simulates Distributed Processes

There are several communication mechanisms available that distributed processes in product systems can use. In a test run with simulated distribution, real communication via a network is not necessary. Therefore, the objects used for communication must be replaced by dummies which transfer the information to be transmitted in another way.

In the auction system, the communication is realized via multiple self-designed layers based on direct HTTP queries. This has led to a very efficient and configurable system which can easily switch encryption and the management of communication statuses in "sessions" on and off as well as handle different types of firewalls and cache systems. The basic structure of the communication is represented in Fig. 8.26 with abstraction of details such as sessions and encryptions. It describes only that part of the communication that is used for submitting bids.

In the client, the call hierarchy corresponds to the layering, hence `asp` calls `mhp` which uses an existing connection or creates a new one, transforms the data into an encrypted string, and then transfers it. On the server, the activity starts from a set of existing threads that is not shown here. At the socket, this set of threads checks whether a query is present and, if this is

[6] `context Auction a` is a universal quantification over all objects of the class `Auction`.

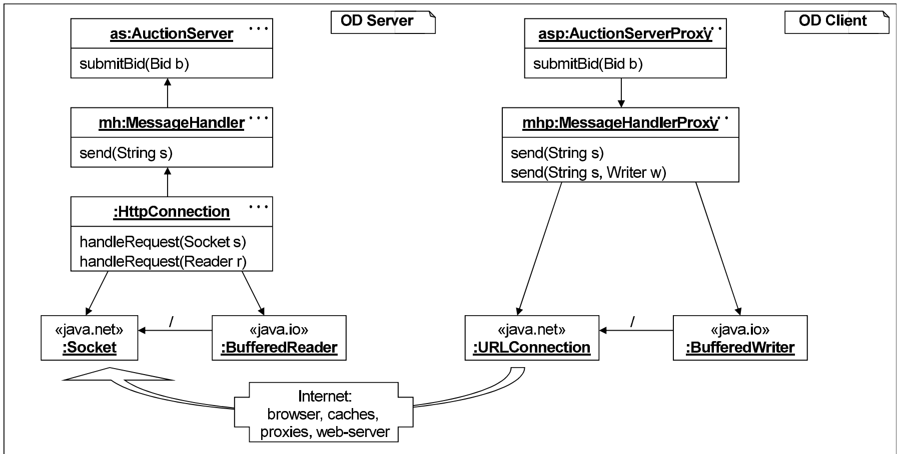


Fig. 8.26. Communication layers in the auction system

the case, allows the query to be processed in separate instances of the class `HttpConnection` respectively. Therefore, on the server, the activity starts from the class `HttpConnection`.

As discussed in Section 8.1, in distributed systems we can also test individual layers or combinations of layers. To enable this, in each case we have to design a suitable configuration using dummies and factories. We can represent the configuration of simulated distributed systems with an object diagram. For example, we can achieve a very simple configuration that hides all communication aspects using the object diagram shown in Fig. 8.27. In this case, we use a dummy for delegation so that a copy of the transferred objects can be created and no common objects are used in different locations.

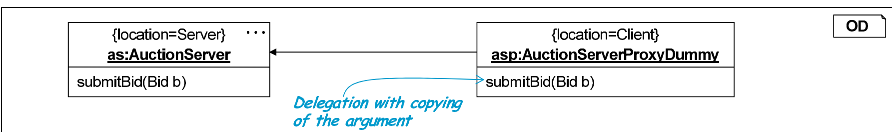


Fig. 8.27. Connection in the uppermost layer

The configuration shown in Fig. 8.28 also allows us to check whether the transferred bids have been correctly converted into strings ("marshalling"). The layers have each been designed such that the original can be inserted instead of the proxy. The shown configuration does not even require to build copies for a transfer between the locations since strings are unmodifiable objects.

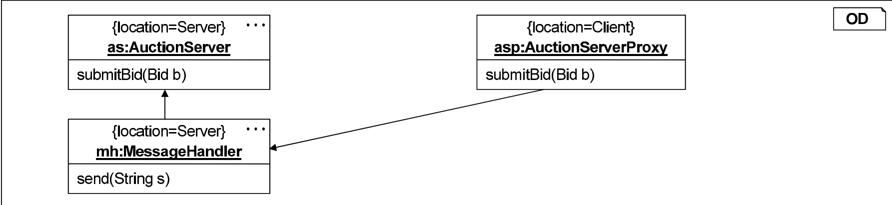


Fig. 8.28. Connecting the handler in the second layer

Further configurations are possible by replacing the `Socket` and `URLConnection` objects with dummies or, as shown in Fig. 8.29, replacing the reader and writer with a pipe.

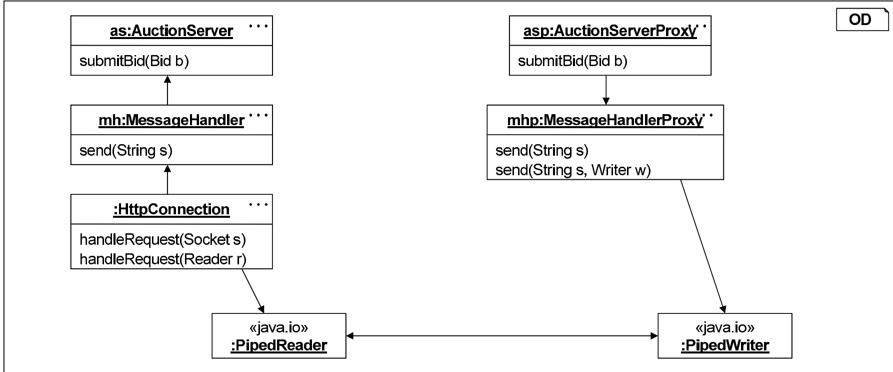


Fig. 8.29. Connecting the handler in the third layer

In some cases, we can reuse test drivers for different configurations but we may have to adapt them in some circumstances. Thus, in the configuration in Fig. 8.29, we must ensure that the data stored in the pipe is retrieved again. This means that `HttpConnection` must be activated with an appropriate frequency. In the configuration shown in Fig. 8.27, this additional effort for the driver is not necessary.

8.5.5 Pattern for Distribution and Communication

The simulation of distribution and the resulting effort for static variables and communication can be summarized in the pattern given in Table 8.30.

Pattern: Distribution and communication	
Intention	Real distribution is very difficult to test and distributed test runs are time-consuming. For reasons of efficiency distribution should be simulated in one process space.
Application	It is useful to apply this pattern in the following cases: • Distributed systems are tested; and • Global system states have to be tested across distributed subsystems; or • The communication or partial aspects of the communication, such as encryption or communication states, are to be tested.
Structure	Locations (Table 8.24) model the physical distribution. Static variables and factories are individualized for the locations using the pattern from Fig. 8.25. In the implementation, information about the currently active location is hidden. The communication is set up according to the class structure below. Different configurations allow flexible use in the product system and in test cases. «interface» Handler; CD; {location=Server} HandlerServer; {location=Client} HandlerProxyDummy; {location=Client} HandlerProxy
Dummy implementation	The dummy delegates directly to the server. Under some circumstances, object structures specified as arguments have to be copied or the duplicates that already exist in the respective location have to be used. The dummy acts as an interface between different locations. It is responsible for switching the process context when the server is called and returned.
Noteworthy	The server does not necessarily have to realize the same interface as the proxy. If it does not, the effort for implementation in the dummy is correspondingly greater.

Table 8.30. *Pattern:* Distribution and communication

8.6 Summary

This chapter discussed how we can use UML/P models in a practical way to model test cases. It also described test patterns suitable for testing functional properties of a distributed or concurrent system. We can thus prepare the test object and test environment for functional, automated tests. While the use of dummies is proposed as standard in the literature available on the topic of testing [Bin99, LF02], the patterns for simulating time, communication, and distribution for functional tests are new. A discussion of the problems of using frameworks and components is also rarely found, with the exception of [SD00]. Frameworks and components are not systematically replaced by simulations with adapters anywhere else. One clear consequence of this chapter is to realize *testability* of the system already in the architectural design phase.

9

Refactoring as a Model Transformation

All the forces in the world are not so powerful
as an idea whose time has come.

Victor Hugo

Refactoring means applying *systematic and controllable transformation rules* to *improve the system design* while *retaining the externally observable behavior*. This chapter first discusses the method of applying refactoring rules and the concepts of *model transformations*. On this basis, the chapter then develops a *notion of observation* which is based on UML/P test cases and practically usable. Building on this, the following chapter then discusses a collection of refactoring rules for UML/P.

9.1 Introductory Examples for Transformations 256
9.2 The Methodology of Refactoring 261
9.3 Model Transformations 265

© Springer International Publishing AG 2017
B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_9

The requirements for a software system in use change due to changes in the business model as well as due to the increasing demands of the users in terms of the software functionality. The technological basis of an application, such as the underlying operating system, frameworks used, or neighboring systems, is constantly evolving.

During a project, users may come up with new requirements and we have to introduce these requirements into the project flexibly. Furthermore, in most cases, software today is so complex that it is not possible to develop a system architecture that meets all potential requirements from the very beginning.

Therefore, we have to learn techniques for developing software further and for adapting software to changing requirements and a new technical environment. One technique that we can use is the *refactoring* of existing software [Fow99, Opd92]. This technique consists of a collection of targeted and systematically applicable transformations that we can use to significantly and consistently improve a system architecture. This improvement is then the basis for extending the system.

Refactoring techniques [MT04] are increasing in popularity. This is why there are now adaptations for special languages such as Ruby [FHFB09], HTML [Har08], or UML [SPTJ01, Dob10]; refactoring is also being applied to large and complex software systems [RL04, Mar09]; it is also being used to introduce design patterns [Ker04], or for certain tasks in software development, such as software reorganization [Küb09].

The goal of this chapter is to sketch a refactoring approach by using examples. The chapter then focuses on embedding refactoring techniques in the world of model transformations. As a first step, Section 9.1 presents some examples of different types of refactoring applications. Section 9.2 classifies the refactoring techniques in the software development process methodologically. Section 9.3 then explains the concepts of model transformation, of which refactoring is a special case. The section increases the precision of the semantics of transformation rules and defines a suitable notion of observation for the approach presented in [Rum16].

9.1 Introductory Examples for Transformations

This section demonstrates the types of refactoring steps available by applying them on some examples. As already discussed in [Fow99], the principle of refactoring can be demonstrated with very small examples, although these small examples do not exhibit the reasons for a refactoring. [Fow99] therefore uses a larger example covering 50 pages to show that we can control complexity in evolution using refactoring techniques. The small examples below therefore show only the possible uses of refactoring rather than explaining the necessity for using refactoring.

Fig. 9.1 contains terminology definitions from literature sufficient for the subsequent examples. Building on this basis, Fig. 10.2 in the following chapter contains a more detailed definition of terminology.

Definitions of the term "refactoring" from literature

- "*Refactoring*" can be characterized as an operation for restructuring a program that supports the design, evolution, and the reuse of object-oriented frameworks. [Opd92]

The work that is most quoted on the topic of refactoring offers the following definitions:

- "*Refactoring* (noun): A change to the internal structure of a software to make it easier to understand and change without changing its observed behavior." [Fow99]
- "*Refactoring* (verb): to restructure software by applying a series of refactorings without changing its observable behavior." [Fow99]

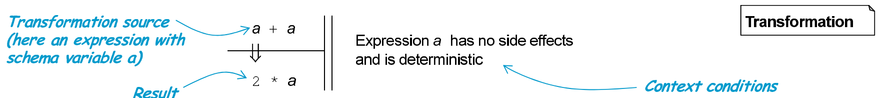
Fig. 9.1. Terminology definitions for "refactoring"

[Opd92] also refers to refactoring steps as software reorganization plans that permit modifications at a "mid-size" level. In [Opd92], modifications to individual code lines are referred to as "low-level" modifications; adapting entire functionalities visible for the user is referred to as a "high-level" modification. [Opd92] sees refactoring primarily as a technique for developing frameworks further. The use of refactoring to evolve architecture within a project was first proposed in [Fow99] and is used, for example, in the Extreme Programming approach (see Section 2.2).

In this book refactoring is defined as a transformation that transforms a given model and the parts of code it contains into a new equivalent model according to a suitable notion of observation by applying one or more steps.

Algebraic Transformations

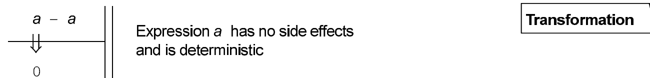
The simplest form of refactoring is given by the well known algebraic transformations of expressions. It uses the mathematical equations that can be applied, for example, to simplify mathematical expressions. One example of this type of rule is as follows:



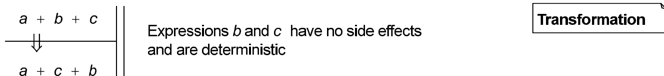
This rule can be applied to math as well as to Java expressions and to OCL. It uses the *schema variables* introduced in Chapter 4 as a placeholder for other expressions. It is important that *a* represents arbitrary expressions of

the base language and not only variables. Although this rule appears to be very simple, when used in Java it has context conditions. Here, the expression inserted for *a* must not have any side effects. For example, after the application of the transformation to `(i++)+(i++)`, the variable `i` would have a different content and a different result would be produced.

a must also be deterministic. It would not be suitable, for example, to query the time with the static query `Time.now()` for *a*. This would be particularly noticeable if the following rule were to be applied:



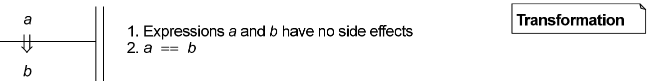
Further effects can occur with algebraic transformations and these must be noted. For example, swapping an addition according to the rule



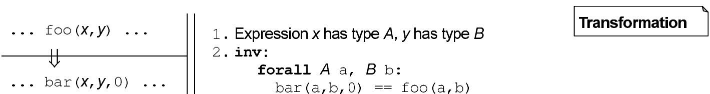
can lead to an arithmetic overflow that only occurs in the new calculation if *a* and *c* are very large and *b* is negative. Conversely, we can use an algebraic transformation to increase robustness of the system against arithmetic exceptions. We can also influence the accuracy of the result with suitable transformations of numerical calculations.

However, algebraic transformations are in no way limited to numerical calculations; we can also apply them to other primitive data types and containers. These include simplifications of Boolean calculations, transformations for strings, the utilization of commutativity and other laws for sets, etc. Note, however, that often side effects are forbidden and the existing identity of, for example, containers, must be observed.

The *substitution of equal expressions* also familiar from mathematics can be formulated as follows:



We can apply the following rule to replace a method call with another, more general call (`0` can also be a different constant), for example:



This allows us to introduce a new, more general method `bar` and to eliminate the old method `foo`. In its context conditions, this rule also requires an OCL invariant to be valid.

Algebraic transformations are familiar from mathematics and algebraic specification techniques [BFG+93, EM85] but are not seen as a core of the refactoring techniques developed for object-oriented languages. They are, however, a basis for performing the transformations of code bodies and invariants are often required.

Expanding Methods

Another example of refactoring is the expansion of a method:

```
                                 1. a is of type A                    Transformation
3 + a.getX()                     2. class A { ...
     ⇓                                int getX() {
                                          return 2 * getY();
3 + 2 * a.getY()                     }}
                                 3. getX() is not redefined in any subclass
```

Sometimes more complex method bodies require modifications of the expanded method body:

```
                                 1. a is of type A                    Transformation
m = 3 + a.foo(7+k)               2. class A { ...
     ⇓                                int foo(int x) {
                                          int c = bar(x);
int c = bar(7+k);                         return 2 * c;
m = 3 + 2 * c                        }}
                                 3. foo(int x) is not redefined in any subclass
```

These rule applications use the same principle that is executed by optimizing compilers for "inlining" methods, for example. However, a generalized formulation of the expansion rules has a lot of context conditions because, for example, variables are expanded into a different binding area and inadvertent identical names are not permitted. When such a rule is used for refactoring, the goal is often to use expansion to perform a subsequent algebraic simplification or to refactor the resulting piece of code again with a different method call.

We can also use the expansion rules outlined above in the reverse direction as extraction rules[1] in order to factor out subfunctionality. This simplifies reuse of subfunctionality and it can be tested separately. It also simplifies redefinition of the subfunctionality in subclasses.

The application of the expansion rule is also subject to context conditions and usually also requires additional modifications for adapting the expanded code. If the expanded method is from a different class, then a context condition must require that no attribute of this class is used, for example. We can achieve this with a preparatory transformation by encapsulating all attributes used with method calls.

[1] In other transformation approaches, the rule for extraction is known as "folding".

A result of the expansion is that the dynamic binding of the method is lost. Therefore, the expanded method must not have any redefinition in any subclass. Alternatively, we could use data flow analyses to ensure that the object in *a* is actually precisely from class *A*.

The context conditions for factoring a piece of code into a submethod are typically complex. However, we can verify most of these context conditions with *syntactic analyses*, meaning that the correctness of this type of rule can be checked automatically, quite like in the expansion case. Relatively often we can replace undecidable context conditions by using (somewhat stricter) syntactic conditions that are relatively easy to check to ensure decidability. One example is ensuring that the private method `foo` is only applied to its own object `self`: while this is undecidable in *a*`.foo()` for arbitrary expressions *a*, the restriction to `self.foo()` makes it trivial.

Restructuring the Class Hierarchy

Other refactoring techniques modify the system structure specified by class diagrams. We can factor out new abstractions as new superclasses or shift attributes and methods along the class hierarchy. Fig. 9.2 demonstrates the introduction of a new class in the middle of the class hierarchy. This class receives the common implementation of the method `validateBid()` extracted from the subclasses. To enable this, class-specific differences are factored in separate methods. For example, the new method `setNewBestBid()` contains the comparison, specific for the subclasses, of whether a bid qualifies as a new best bid.

The refactoring shown has already been specialized for the application to be processed and shows several steps simultaneously. Using the schema variables introduced in Section 4.4.2, we can represent the introduction of the new class more generally by using placeholders that can be filled out instead of specific class and method names.[2]

We can shift attributes and methods along associations if the corresponding association and the classes involved in it satisfy certain conditions. For example, we can usually no longer modify an established link of such an association in order to keep the access to an outsourced attribute. However, a static analysis is usually not able to recognize these time-based context conditions; instead, the conditions require additional measures, such as the use of suitable invariants and tests.

Signature Modification

Refactoring steps can cause internal modifications or modifications to the signature of system parts. For example, the removal of a local variable that is

[2] We can also use schema variables as placeholders for lists of attributes or subclasses but this requires an intuitive graphical representation that is not addressed in this book.

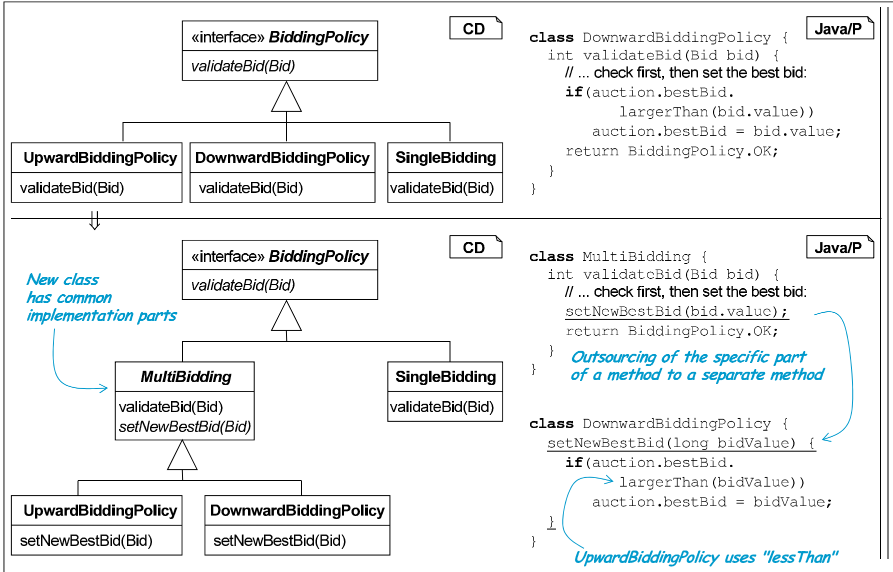


Fig. 9.2. Introduction of a new superclass

no longer required is not critical. The removal of a method is problematic if this method has been *published* in the signature of the class or the subsystem and it is possible that other developers are using the method intended for deletion. Therefore, it is often necessary to process the system in its entirety as far as possible instead of applying refactoring to locally restricted, open subsystems.

The modification of signatures also shows that the notion of observation is very important here. Beyond the behavioral equivalence of the entire system that is required in Fig. 9.1, interfaces within the system and to other system parts offer additional observation points that we have to take into account when applying refactoring techniques.

9.2 The Methodology of Refactoring

9.2.1 Technical and Methodological Prerequisites for Refactoring

Like many other elements in the portfolio of an agile methodology, refactoring is particularly successful in combination with other techniques and concepts:

- Object orientation, in particular inheritance and polymorphism
- Automated tests
- Common model and code ownership

- No or hardly any additional documentation
- Modeling and coding standards

Object-oriented concepts were already seen as helpful for reusing software components as early as [Mey97]. In particular, the formation of subclasses, the dynamic configurability of object structures, the dynamic binding of methods, and the resulting possibility to partially redefine behavior are recognized as factors which allow better reuse. The test patterns presented in Chapter 8 show that object-oriented concepts can also be used beneficially to define tests.

However, *automated tests* are a significant cornerstone for the success of refactoring techniques. Automated tests allow us to efficiently verify whether the functionality not directly affected by a refactoring still fulfills its tasks when the refactoring is performed. If no tests are available for system parts that are to be subjected to a refactoring, it is advisable to first develop suitable tests. Furthermore, as discussed in Section 9.3.3, the acceptance tests specified by the users of the system being developed define a notion of observation for refactoring steps.

If each artifact is owned by a developer and this user is the only person permitted to modify it, refactoring is rather doomed to fail. The coordination effort required between the developers to modify multiple artifacts in parallel (and, due to many small iterations, almost simultaneously) cannot be realized practically. Furthermore, owners of an artifact refuse the additional workload if a refactoring is not an advantage for them but only for the other developers. In such cases, the result is often not the best solution, but rather the solution that involves the least effort for the winner of a discussion. *Common model ownership* allows individual developers to perform refactoring across and beyond interfaces and artifacts efficiently and to commit only the modified and completely executable system into the repository. If automated tests are available, the coordination effort between the developers becomes a "coordination effort" between the developer (or pair of developers) and the automated tests.

Refactoring modifies the structure of a system. If the structure is described not only by the models used for code generation but also by further documentation, additional effort has to be invested in updating these documents. Unfortunately, this effort can easily reach and exceed the workload of the intended refactoring. The effort involved in finding the places that have to be changed in a document is often rather high. When refactoring an implementation, the according documents need to be kept synchronized. Otherwise, developers cannot be confident that the documents are up to date, which makes the documents rather worthless. Therefore, refactoring can be used most effectively if no additional, detailed documentation exists. However, this requires adherence to precise modeling standards to simplify developer access to the models. Approaches such as "literate programming", with an intensive interweaving of the model and the documentation, are

also not particularly suitable because the informal documentation cannot be adapted automatically.

9.2.2 The Quality of the Design

As outlined in Fig. 9.3, refactoring is orthogonal to the development of new functionality. In normal development, new functionality is added and the fact that the quality of the design is thus reduced is accepted; with refactoring, the functionality is retained and the quality of the design is usually improved.

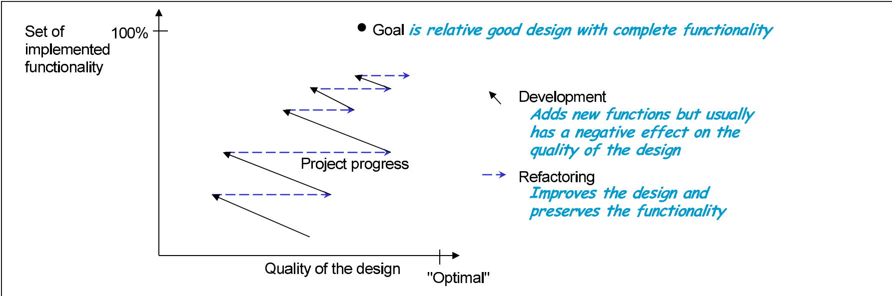


Fig. 9.3. Refactoring and development supplement each other

There are ways of measuring the functionality of a system, such as the Function Point Method [AG83], or the object-oriented adaptation of this method [Sne96]. In the simplest case, the proportion of requirements already implemented can be used as a unit of measure for the implemented functionality.

In contrast, there is currently no generally recognized procedure for measuring the quality of a design objectively. Criteria developed for programming languages to measure the "optimality" of a design are dependent on the programming language and the experience of the developers. For example, different criteria are proposed for object-oriented languages than were discussed for procedural languages 10-20 years ago. Important points include keeping the *coupling* of classes relatively low, not making individual classes too large or too small, limiting the *depth of an inheritance hierarchy*, etc. Deficits are referred to as "bad smells" in [Fow99], for example, where 22 deficits are listed.

However, it is not only refactoring steps to improve the design in terms of given metrics that are useful; above all, refactoring should be used to simplify addition of new functionality in a subsequent step. Goal-oriented refactoring may therefore initially make a design worse in order to later add new functionality or to perform further refactoring steps. A typical example is the

splitting of one class into two classes connected by a `1-to-1` association and the subsequent generalization of the association in the form `1-to-*`.

Here, for example, it is helpful to use metrics to identify deficits. A proposal for eliminating these deficits with suitable refactoring steps is also useful. However, it is the user who is familiar with the design and the underlying motivation who has to decide whether a refactoring proposal is applied.

Some typical measured metrics on programming languages (such as the inheritance hierarchy or the coupling of classes) can largely be taken over unmodified. On the other hand, the fact that UML is much more compact than Java allows a greater density of functionality within a class and therefore a reduction in the number of classes required. After all, a part of the standard functionality (such as `get` and `set` methods for attributes, factories, technical methods for saving etc.) is added by the code generator and is therefore no longer visible in the model for the developer.

In parallel, further criteria for good design can be specified for the UML/P notations. For example, an object diagram that contains a lot of objects should be split into multiple object diagrams. These individual object diagrams are then combined using the logic for object diagrams introduced in Section 4.3, Volume 1. The size and form of Statecharts, sequence and class diagrams, and OCL constraints can also be subject to suitable metrics which have to be grounded on empirical studies, however. For example, the rule familiar from other domains, that an observer can only capture up to 5 ± 2 elements simultaneously, can be used here.

9.2.3 Refactoring, Evolution, and Reuse

A refactoring of the internal structures of a system does not directly lead to any visible improvement in the system´s functionality for the users and customers. Therefore, there must be good reasons for performing a refactoring.

From an economic perspective, refactoring is useful if the goal justifies the workload involved. As discussed in Fig. 9.3, the system created does not need to be optimally designed. However, during the entire development process, the design must be good enough to enable adding more and more functionality. In order to achieve this, the quality of the design should also be good at the end of the project.

Therefore, the benefits of refactoring must always be weighed up against the efforts required. In particular, for large modifications that modify the technical or application architecture of the system by, for example, replacing the communication infrastructure (middleware) or modifying core parts of the data structure, the effort involved must be evaluated in advance. Based on previous practical experience, we can state that refactoring steps are applied consistently—even through the relatively restricted support provided by the search and replacement functionality of a development environment and the existence of a test suite—progress is significantly faster than is often estimated.

We can use refactoring steps not only within a software development project, but also for the purpose originally intended in [Opd92]: the evolution and reuse of frameworks in different projects.

A framework is normally developed further as a stand-alone artifact: in each application, new functionality is added to the framework and any necessary restructuring is performed. Special attention is given to preserving the compatibility of the framework with the original applications so that they can also be developed further. In an agile project, these considerations are irrelevant when the principle of *simplicity* is applied. Extreme Programming is therefore not directly suitable for developing frameworks, but could be adapted for this purpose. To increase the reuse of frameworks, we have to take additional methodological precautions to allow the framework to be shielded against arbitrary modifications. In this case, we have to combine agile and framework-based methods, as outlined in [FPR01], for example.

9.3 Model Transformations

This section analyzes the concepts for the definition of refactoring shown in Fig. 9.1 in more detail. It discusses the nature of model transformations, a notion of observation, a transformation calculus, and the interaction with the semantics of the modeling language.

9.3.1 Forms of Model Transformations

In principle, a model transformation is a mapping starting from a syntactically well-formed expression of the source language UML/P. In contrast to code generation, it is not Java that is used for the target language but again, the UML/P. In principle, however, the meaning, that is, the semantics of a model transformation, remains the same. Building on Fig. 4.9, Fig. 9.4 describes the basic pattern of a parameterizable model transformation.

Because model transformations are mathematical functions, we can investigate their functional properties. A function can be only *partially* defined because the prerequisites for the applicability of the transformation are not satisfied or the new model created is not well-defined. For example, a subclass can only be added if the superclass exists. On the other hand, if we add a class that already exists, this results in a model that is not well-defined. Just like in these two examples, many of these prerequisites can be checked automatically. However, this is not the case with all prerequisites.

A model transformation can be *injective*. An interesting situation also arises when a model transformation is not injective, as it is obvious then that different models are initially mapped to the same new model and information-bearing details are lost. Such transformations typically have the

The following definitions are used to **formalize the concept of model transformation**:

- The modeling language (UML/P) defines a set UML of syntactically well-formed expressions, and
- Accordingly, a model transformation is a mapping $T : UML \rightarrow UML$.
- If the model transformation is parameterized with a script, this corresponds to a mapping $T_p : S \times UML \rightarrow UML$.

T_p is thus a type of interpreter of the script language S for model transformations. Based on the lifted but mathematically equivalent version $T'_p : S \rightarrow UML \rightarrow UML$, for a script $s \in S$, each transformation is of the form $T'_p(s)$.

Fig. 9.4. The principle of a model transformation

character of an abstraction and do not necessarily *preserve semantics* in all details. One example is the removal of all attributes from a class diagram. This results in a more abstract representation with less detail information.

Model transformations are normally not *surjective*, meaning that not every model can be created by using transformations. For a calculus, however, it is worth investigating whether the entirety of the model transformations and their combination is surjective. This would allow us to derive each model from a generic start—for example, an "empty" model—using transformation steps. In the delta modeling approach [CHS10, HKR$^+$11a, HKR$^+$11b], for example, chains of transformations are used to set up product variabilities from an initial small model.

9.3.2 The Semantics of a Model Transformation

The Principle of defining Semantics for a Transformation

The use of a functional mapping instead of a relation to explain model transformations indicates the constructive nature of the transformation. A relation—for example, an abstraction[3] or a refinement relation—can be used when defining semantics. Based on Fig. 9.4, Fig. 9.5 discusses the principle for defining the semantics of a model transformation.

The semantics of a model transformation as described in Fig. 9.5 can be understood such that for any transformation, the graphically represented commutativity must be proven. In practice, however, and in particular for

[3] An abstract superclass represents an *abstraction* within the model. This must be differentiated from a model transformation which performs an *abstraction* between models. The introduction of a new superclass in a class diagram is not a *model abstraction* in this sense; rather, it adds new information to the model. However, this generally does not lead to different behavior being observable externally and therefore is a refactoring.

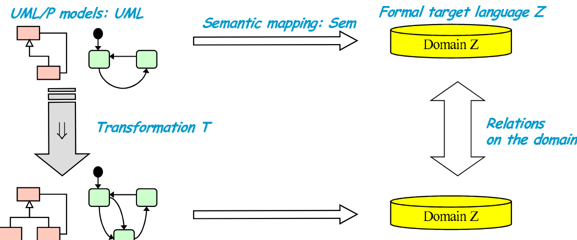
To define the **semantics of a model transformation**, building on Fig. 9.4, the following definitions are required:

- A suitable formal target language with the vocabulary Z
- A formal semantics $Sem : UML \rightarrow \mathbb{P}(Z)$ which maps each element $u \in UML$ a set $Sem(u)$ of elements of the target language
- Relations $R \subseteq \mathbb{P}(Z) \times \mathbb{P}(Z)$ which represent relationships such as abstraction, change of signature, refinements, etc. between elements of the target language

A *model transformation satisfies such a relation* R *for model* $u \in UML$ if, for the transformed model u, the following applies:

$$(Sem(u), Sem(T(u))) \;\; \in \;\; R$$

This condition can be illustrated graphically with a commutative diagram:



If this relationship applies for all models $u \in UML$ for which the transformed model is well-defined (that is, u satisfies the context conditions of the transformation T and $T(u) \in UML$), then *the model transformation satisfies the relation* R or *is an operative implementation of* R.
Typically, there are a lot of different model transformations that satisfy the same relation. If the transformation is described by a script $s \in S$, then *a script satisfies a relation* R if the transformation $T'_p(s)$ satisfies the relation R.

Fig. 9.5. Model transformation and model semantics

larger transformations, this cannot be achieved easily. Instead, in larger transformations, an explicit discussion of special cases is necessary and test suites checking invariants provide an informal justification for the transformation. A formal proof of correctness, however, is left to the person performing the concrete transformation (where applicable). With the level of granularity of, for example, the refactoring rules from [Fow99] discussed in Section 10.1.2, and with the rules shown here, for the purposes of formalization, we would first have to define the context conditions and the special cases much more detailed. We can then think about proving the correctness of refactoring rules based on the resulting more precise descriptions.

Examples for Semantics and Relations

An essential element for the semantics of a model transformation in Fig. 9.5 is the presence of relations R on the target language Z. Relations with different levels of strength are available depending on how well the theory under the target language used for the semantics is set up. An example of a mature theory are the streams in Focus [BS01b], which offer a wide variety of different types of abstractions and refinement relations, from a signature relation to an interaction and time refinement for distributed systems.

A further example of a mature theory are the finite state machines (automata) introduced in Section 5.2, Volume 1, with semantics in the form of sets of recognized words over the input alphabet. If we assume the recognized sets of words to be an observation, then, for example, we can understand the transformations (familiar from automata theory [HU90, Bra84]) for calculating a deterministic automaton variant or for minimization as refactorings. Most of the transformation rules for Statecharts from Section 5.6.2, Volume 1 also modify only the structure of a Statechart; they do not affect the behavior of the Statechart that is visible externally.

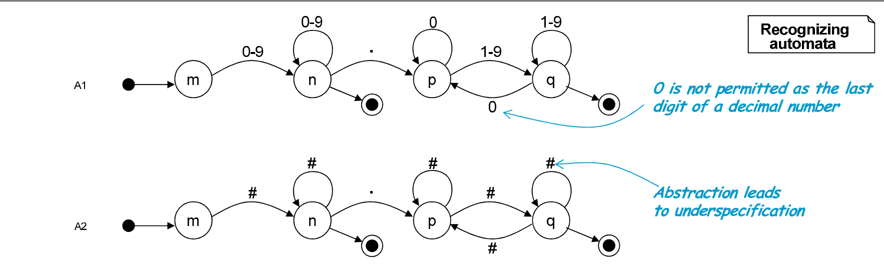
For an automaton, the replacement of an input character corresponds to a transformation to *rename the signature*. An *abstraction at signature level* is when, for example, a group of input characters is replaced by a single character. Fig. 9.6 shows the effect of an abstraction on a recognizing automaton.

A (real) abstraction looses information from the detailed initial model. One feature of an abstraction is therefore that it maps multiple initial models to the same target model. Abstractions are therefore not normally surjective. Accordingly, they are also not suitable for developing a system further. However, they are suitable for analysis purposes or for defining tests that can be derived more easily from the more abstract model.

It is also worth noting that *abstraction* on a syntactic level means that elements or information details are removed from the model and the model thus becomes "smaller". In contrast, due to the form of the definition of semantics Sem, which, as discussed in Section 4.1.1, can also be referred to as *loose semantics*, the set of possible implementations increases in size. The abstraction relation is thus represented as a set inclusion (formally: $R = \{(X,Y) | X \subseteq Y\}$).

The inverse relation is called a *refinement*. It enables us to get from an abstract model to a more specific, more detailed, and thus more complete model by adding information.

As discussed below, the purpose of the refactoring techniques is to preserve semantics. Hence they are neither abstractions nor refinements of the initial models; instead, they merely represent a restructuring that cannot be observed from outside. The associated relation will therefore be an equivalence which acts as an identity based on the notion of observation introduced later.



Assume that $L(A_i)$ is the set of words recognized by the automaton A_i. Automaton A_1 describes a number format that does not permit `0` as the last digit after the decimal point.
In an abstraction, the individual numbers are replaced by a character `#`. On the alphabet, the abstraction relation is:

$$\phi = \{(n, '\#') \mid n \in ['0', \ldots, '9']\}$$

This is extended to words on a pointwise basis. Example: $('17.33', '\#\#.\#\#') \in \phi$.
A replacement of the figures in automaton A_1 according to ϕ leads to automaton A_2.
For every word w_1 recognized by A_1, the abstraction w_2 is recognized accordingly by A_2:

$$\forall w_1, w_2 : w_1 \in L(A_1) \wedge (w_1, w_2) \in \phi \Rightarrow w_2 \in L(A_2)$$

The inverse situation does not apply, however, as the example `'17.30'` shows.
As is often the case with abstractions, information is lost. For example, the transitions labeled only with the figure `0` can no longer be represented in sufficient detail. A representation which is *underspecified* compared to the original automaton arises; it still contains important information but no longer has as much detail.

Fig. 9.6. Abstraction using an example of a recognizing automaton

The examples shown here originate from the relatively simple and well-understood theory of finite automata as this is the easiest way to explain the principle. The same principle applies for Statecharts, which are much more syntactically comprehensive; however, the usage conditions that have to be complied with are much more complex. For Statecharts, it is also important that they have an output which can be observed in addition to the input. The resulting semantics-preserving transformations for Statecharts were discussed in Section 5.6.2, Volume 1.

Categories of Semantics Relations

The examples shown present three semantics relations. Categorizing them shows that due to the semantics selected, only a few relations are necessary. We can identify the following categories:

1. *Abstraction*, which abstracts from details. The resulting model has a superset of the initial model as semantics.
2. *Refinement*, in which details are added. Refinement is therefore the opposite of abstraction.
3. *Refactoring*, as a semantics-preserving transformation with regard to a predefined observation. Refactoring can equally be seen as a special case of refinement and at the same time abstraction (with regard to the observation).
4. The *change of signature* by *renaming* syntactic elements that are visible in interfaces.

We can transfer the categorization of semantics relations directly into a categorization of transformation rules. There are therefore transformation rules for *abstraction*, *refinement*, *refactoring*, and *change of signature*, whereby these categories are not disjoint. For example, a change of signature can be recorded as a refactoring when based on an observation that does not use that signature.

The replacement of numbers with the character '#' in the example in Fig. 9.6 is an abstraction. Other forms of abstraction include removing classes from a class diagram, objects from an object diagram, etc. In each case, the resulting model is less expressive. Conversely, in a refinement new classes can be added. However, whether a refinement is a real refinement depends on the notion of observation discussed below.

Transformations to Sets of Artifacts

In Section 1.4.4, Volume 1, we defined that the term "model" includes both an individual artifact—for example, a class diagram or a Statechart—and a collection of these artifacts for describing several aspects. Therefore, in Fig. 9.7, model transformations are expanded to sets of processed artifacts and a form parameterized with scripts can be defined in the same way. This means, for example, that we can rename a method consistently wherever it is used.

The motivation for the definition of semantics used in Fig. 9.7 is that a model u_1 can be seen as a specification constraint for a system that has to be satisfied. There is therefore a set $Sem(u_1)$ of systems that satisfies the constraint specified. If a second model u_2 is defined as an additional constraint, the set of systems that now are realizations is $Sem(\{u_1, u_2\}) = Sem(u_1) \cap Sem(u_2)$, that is, precisely those systems that satisfy both constraints.

Open and Closed System Specifications

A common form of applying of model transformations today, for both CASE tools for DSL or UML models and for development environments with refactoring support, is the assumption of a *closed system specification*.

The expansion of the **model transformations to sets** is defined based on Fig. 9.5. A model transformation is extended to a mapping $T : \mathbb{P}(UML) \rightarrow \mathbb{P}(UML)$ to sets in which each artifact of the set $M \subseteq UML$ is transformed individually:

$$T(M) = \{T(u) | u \in M\}$$

The semantics definition is extended to $Sem : \mathbb{P}(UML) \rightarrow \mathbb{P}(Z)$ as follows: for sets of models $M \subseteq UML$, as is common for loose semantics, the following is defined:

$$Sem(M) = \bigcap_{u \in M} Sem(u)$$

Fig. 9.7. Model transformation to sets of artifacts

A model is called *open* if it models an *open system*, that is, has explicit interfaces to the system environment whose signature is given. Furthermore, the model expects certain behavior of the system environment. This environment includes neighboring systems modified by colleagues, predefined frameworks, the operating system, or middleware components. A model is *closed* if there are no such interfaces.

A closed model typically arises if the model explicitly contains the system environment. Closed models are easier to modify and adapt than open models because in an open model, the assertions of the system promised to the environment must be ensured. Interfaces and interaction patterns must not be modified in relation to the environment.

Although today's systems almost always contain interfaces to the environment due to the existence of frameworks, for example, developers generally work as far as possible as though the system is described in a closed form. The approach propagated in [SD00] and used in Chapter 8 for separating the application core from external components with adapters gives rise to a closed form as a side effect because interfaces to the environment are encapsulated. The adapters are then treated as part of the system and the effect of a closed system that can be controlled and manipulated arises.

The common ownership of models for developers that stems from the methodological approach of this book and that is discussed in Section 9.2 has the same effect. Boundaries between the artifacts created by a developer and those of team colleagues are thus removed. The environment of the domain primarily developed by one developer is thus accessible to and modifiable by other developers, as in a closed system.

A System Model as a Semantic Domain

Fig. 9.4 and 9.5 introduced the abstract sets UML and Z for the syntax and the semantic domain. While the syntactic form of UML/P was discussed in detail in Volume 1 and is represented in Appendix C, Volume 1 by EBNF

productions and syntax class diagrams, we have not yet characterized the domain for formalizing the semantics further.

For semantic domains, there are a number of proposals that are used to formalize parts of UML. [HR00] and [HR04] contain an overview of these proposals. In particular, different logics and mathematical formalisms are used and in most cases, are extended by specific constructs. This includes, for example, [BHH+97], which discusses a formalization of larger parts of UML based on a distributed, asynchronously communicating formalism, or [FELR98b], in which UML is transformed into the formal language Z [Spi88].

This book, which aims to provide a methodological application of UML, does not cover the formal definition of a semantic domain and a mapping of semantics based on that definition. Nevertheless, this section discusses the basic appearance of such a semantic domain in order to improve precision of the notion of observation in the next section.

In order to be able to give a set of different UML diagrams precise semantics, all essential aspects of a system must be represented in the semantics. It is not enough to model individual aspects such as the input/output behavior or the contents of individual objects in the semantic domain. A *closed* form of a system and explicit representation of the structure and time in that system is preferable. The form of representation of distribution, concurrency, and asynchronous communication influences the semantic domain significantly. In the approach discussed in this book, such aspects are covered only marginally. The tests for distributed systems and parallel threads defined in Chapter 8 also replace real concurrency with simulated scheduling and thus a sequential, deterministic execution.

[Huß97] and [Rum96] describe the basic structure of this type of semantic domain. The structure is also referred to as a *system model*, that is, an abstract representation of the structure and the behavior of systems. A system model for UML is described in detail in [BCGR09a] and [BCGR09b]. In principle, from a content perspective, this type of mathematically formal representation is very similar to the definition of a virtual machine, as discussed in Section 4.2 for UML. However, a constructive, operational description is given for the definition of a virtual machine, while a system model can generally be defined more compactly as a form of denotational semantics.

A system model that is adequate for UML/P describes a system via a set of *system sequences* which on their part characterize *interactions*, *snapshots*, and the *objects* contained therein. As a simplification of [BCGR09a] and [BCGR09b], Fig. 9.8 characterizes essential elements, whereby for simplification purposes, infinite sequences are omitted and the stacks actually assigned to the threads are distributed to the objects.

9.3.3 The Concept of Observation

According to the characterization of a refactoring rule, the *observation* of the behavior to be obtained is fundamental. However, the two most respected

The **system model** describes a set of executions of a system. The following definitions are introduced for this purpose, whereby basic elements such as the set of attribute names $VarName$ or the values $Value$ are not defined in further detail.

- An *object* $o = (ident, attr, stack)$ is defined by a unique identifier $ident$, at each point in its lifetime an assignment $attr : VarName \to Value$, and the part assigned to it from the stack. Obj is assumed as the set of objects.
- A *snapshot* $sn \subseteq Obj$ is a collection of objects that exists at a certain point in time. The links between objects are represented by the unique identifiers in the snapshot. SN is assumed as the set of snapshots.
- The set of actions Act describes method calls, returns, attribute assignments, etc.
- A snapshot knows which object is currently active and thus which action will be executed next. The program counter can therefore be reconstructed from the $stack$ of objects. This defines the next action of a snapshot that will take place: $act : SN \to Act$.
- An execution $run \in SN^*$ is a series of chronologically sequential snapshots.
- The system model $SM \subseteq SN^*$ consists of a set of executions.

A number of additional conditions is necessary in order to restrict SM to the executions that can actually occur. For example, the control flow must be preserved and only attributes that exist in the object must be assigned values.
In Fig. 9.5, Z was introduced as an abstract target language for a definition of semantics. We can now set $Z = SM$. Each UML artifact $u \in UML$ then receives its semantics in the form of a subset of system executions from SM that satisfy the properties described.

Fig. 9.8. Principle of a system model as a semantic domain

works on the subject of refactoring [Fow99, Opd92] do not define the notion of observation precisely. In [Opd92, p. 28], the notion of observation is reduced to the relation between input and output without considering interactions. [Fow99] appeals to the intuitive image of observation by the user. In XP projects, *observation* is understood primarily as the behavior that can be observed at the user interface. However, under some circumstances, the interfaces to other systems, databases, frameworks, etc. are also part of the "externally observable behavior".

Although there is no precise definition of the notion of observation in the XP approach, we can define observations to check the correctness of a refactoring of a system. To do so, the XP approach uses test case definitions formulated in Java that are largely limited to checking the result in the form of a postcondition.

The tests discussed in Chapters 6 and 8, and in particular their descriptive components, such as oracle functions, invariants, interaction patterns, and postconditions, are an ideal mechanism for representing observations. Therefore, observations for refactorings are formulated in UML/P test cases. From the list of potential forms of observations, it is obvious that an obser-

vation formulated with UML/P does not have to be restricted to interfaces; it can also "observe" internal interaction patterns, intermediate states, and the final state of the test object. Fig. 9.9 illustrates this based on a system sequence that consists of a sequence of snapshots.

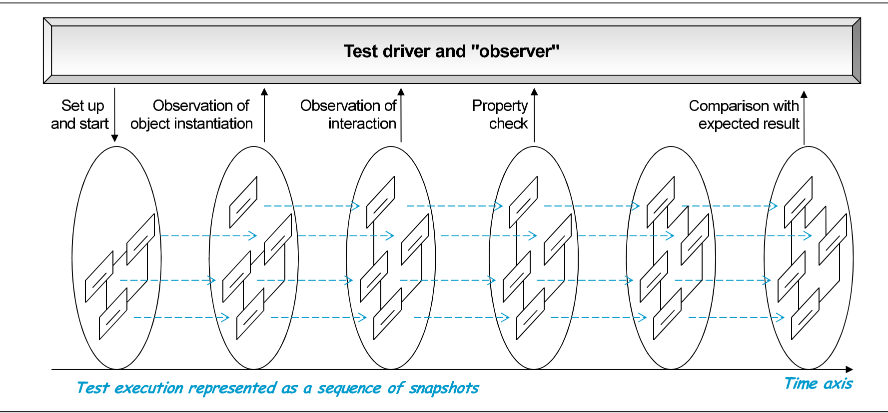


Fig. 9.9. Test as observation of a system sequence

The ability to circumvent any encapsulation of a test object in a test bears significant advantages when modeling tests: the state of the test object is directly accessible and we do not have to draw conclusions about the state of the test object based on the output behavior. Modeling interaction patterns within a subcomponent that consists of multiple objects also circumvents the encapsulation of the component. The disadvantage of such tests is, however, that they also register small changes to the test object, either because the tests can then no longer be compiled (for example, in the event of a signature change), or they indicate a failure of the test run (for example, in the event of a change to the interaction pattern). This means additional effort is required for refactoring, as these tests also have to be adapted.

Therefore, when defining tests, we have to carefully weigh up whether to use internal elements of the test object or a more abstract programming interface such as public methods. In practice, we can roughly identify two classes of tests. The *unit tests* for individual methods and classes will normally access internal elements and have to be modified together with the code. In contrast, automated tests, formulated by the user and realized with help from developers, are part of the observation by a user. This type of test should not be impaired by refactorings. This means, however, that such a test is defined against an abstract interface as far as possible, with the interface then being retained if internal parts of the system are modified. In detail, this means that for acceptance tests, it is better, as far as possible:

- To use query methods instead of direct attribute accesses
- To use OCL properties that allow certain freedom of formulation instead of specifying attribute values precisely
- To ignore uninteresting objects and attributes in the expected result and to concentrate on the significant results
- To observe only significant interactions

In this way, observations defined by the tests are *abstractions* and thus allow different sequences for satisfying the tests. We can therefore modify the system, or rather its model, within certain limits without changing the abstract observation. This can be illustrated as shown in Fig. 9.10.

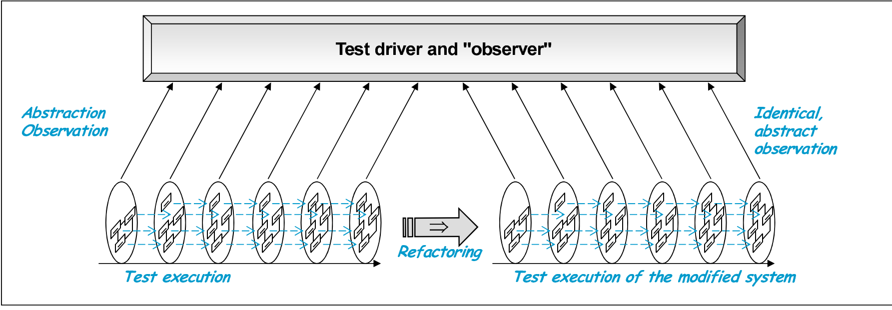


Fig. 9.10. Refactoring allows system executions to stay invariant under observation

However, a side effect of the use of abstraction during the observation of a test execution is that a test does not check all aspects of the test object. Thus, a piece of code may have been executed but its effect ignored by the test. The consequence of this is that the power of expression of coverage metrics for tests based on code that has run through a check is potentially restricted. Here, mutation tests [Voa95, KCM00, Moo01] are helpful—they check whether an existing test suite recognizes modified behavior as a result of a modification of the code. This measures the extent to which the abstracting observations of a test suite capture the observable behavior.

After the discussion on the practical representation of observations, which are illustrated graphically, in Fig. 9.11 we will now add a notion of observation to the more detailed definition of model transformations. This figure also introduces a context for transformations. The context consists of models that the transformed model builds on, for example, by using the types defined in it, but without changing the models themselves. One example is a class diagram that is the basis for an object diagram that is to be transformed.

The third point of the definition in Table 9.11 expresses that an observation in the form of a test forces the test object into certain system executions and only checks test conformity there. $Sem(u) \cap Sem(b)$ represents these sequences, which can be executed by both models, accordingly. Correspond-

An **observation** considers only some aspects of a system execution. An observation is therefore an abstraction:

- An *observation* consists of one or more artifacts of UML/P.
- The semantics of an observation is the subset of the executions of the system Z allowed by the observation and thus also defined by $Sem: UML \to \mathbb{P}(Z)$.
- We assume that $u \in UML$ is a model which is used for constructive code generation and is observed by a property model $b \in UML$. The observation of u refers then only to the common executions represented by $Sem(u) \cap Sem(b)$.
- A model transformation $T(u)$ on the model $u \in UML$ is *observation-invariant* with repects to an observation $b \in UML$ if the observed subset is identical:

$$Sem(u) \cap Sem(b) \;=\; Sem(T(u)) \cap Sem(b)$$

- If the model transformation is applied to multiple artifacts $A \subseteq UML$ and in the context of other models $K \subseteq UML$, the *observation invariance* of a set of observations $B \in UML$ holds if the following is true:

$$Sem(A) \cap Sem(K) \cap Sem(B) \;=\; Sem(T(A)) \cap Sem(K) \cap Sem(B)$$

In the same way, the *refinement* discussed in Section 9.3.1 can be defined precisely in a context and relative to an observation:

$$Sem(A) \cap Sem(K) \cap Sem(B) \;\supseteq\; Sem(T(A)) \cap Sem(K) \cap Sem(B)$$

For an *abstraction* T, the inverse relation $\subseteq$ is used.

Fig. 9.11. The concept of observation

ingly, the *observation invariance* is only necessary for the sequences described by $Sem(b)$. This gives the transformation the freedom for unobserved (that is, typically internal) modifications.

Alternatively, the semantics of observations can also be described by the introduction of a semantic domain B and an abstraction function $\beta: Z \to B$. An observation is then represented by $b \in B$ and characterizes an equivalence class of sequences $\{z \in Z | \beta(z) = b\}$. Therefore, if a refactoring changes an execution z to z', the observable part must still remain the same: $\beta(z') = \beta(z)$. A disadvantage of this approach is the necessity of defining an adequate set of observations B.

As the last point in Fig. 9.11 shows, there is no formal difference between the context and the observations. This type of context can consist, for example, of unmodifiable interfaces to the GUI, databases, etc. However, it can also contain the models not affected by the current modifications.

An example for an observation is a method specification formulated in OCL. Its semantics is a predicate for the descriptions over system executions *SM* from Fig. 9.8. An execution satisfies the method specification precisely

when the postcondition applies at the end of every method call for which the precondition was satisfied at the beginning.[4]

A complete test, consisting of a set of test data, test drivers, etc. also represents a predicate over a system execution. A system execution satisfies a test precisely when the test was performed successfully. This means that in the execution, there is a snapshot that corresponds to the initial set of test data, then the test object is executed, and the required conditions and interactions are satisfied during or after the test.

9.3.4 Transformation Rules

In theory, a transformation rule is explained as a mapping $UML \rightarrow UML$ which, when necessary, can be extended to sets of UML artifacts for a simultaneous semantics and syntactic correctness preserving application. In practice, however, a wide variety of forms of mappings can exist. The number of possible rules depends on the syntactic richness of the underlying language. UML/P has considerably fewer language concepts than the UML standard [OMG10] but it is still a rather large modeling language. Based on experiences with transformation calculi for other languages, we can expect that the number of possible and useful rules will grow more than linearly with the size of the language. One reason for this is that many rules deal with the interaction of several language concepts. From a practical perspective, it is therefore impossible to identify a complete rule calculus for UML/P. This is not necessary for practical use, however. It is more important that an applicable set of compact and simple rules is available.

In many cases, the number and specific form of transformation rules depends on the form of the underlying language. However, there are also general principles for transformation rules that can be applied to many languages. These include, for example, the expansion of methods or the migration of attributes that can also be applied to functions and `struct` entries for procedural languages such as C. However, the applications of the rules differ in technical details. In the case of method expansion, in Java, special cases for abstract methods, dynamic binding, static methods, or constructors have to be taken into account or exceptions have to be handled separately.

The rules should be as simple and general as possible to ensure maximum applicability. The expressiveness of a rule calculus consists to a large extent of the composability of rules as they are applied in succession. Using *goal-oriented tactics*, we can compose complex transformations from simple rules. This allows us, for example, to explain the introduction of a design pattern from [GHJV94] as a series of simple transformation rules [TB01].

Here, it is essential that tool support is available for the basic transformation rules—support that also checks the context conditions and ensures

[4] For a precise description of the semantics of a method specification, see Section 3.4.3, Volume 1.

that the result is well-formed. Goal-oriented tactics can be realized as scripts analog to the form discussed for code generation in Section 4.2.3.

There are also a number of transformation rules that can only be applied when certain stereotypes are present. Volume 1 demonstrates, for example, how we can prioritize transitions with overlapping firing conditions and thus apply different transformation rules for code synthesis. However, because the stereotypes available in UML/P can be freely extended with the mechanism described in Section 2.5.3, Volume 1, we have to define corresponding transformation rules and scripts that allow a specialized handling of models with specific stereotypes.

9.3.5 The Correctness of Transformation Rules

As illustrated with examples in the previous sections, a transformation rule usually has context conditions that have to be satisfied to ensure that a transformation is correct. These context conditions can take various forms, which is why we now classify transformation rules based on the types of the context conditions.

The context conditions range from simple and, in most cases, syntactically verifiable restrictions up to complex invariants that can no longer be decided automatically.

1. In the most simple case, a transformation does not have any context conditions.
2. Simple syntactical conditions—for example, that an expression to be replaced does not use a variable—can be automated via corresponding checks. This form of context condition occurs frequently and can be checked efficiently with good tool support using the syntax tree.
3. More complex context conditions such as the *type correctness* can also be decided based on the syntax but require considerably more effort. Further examples of such conditions are *control flow analyses*—for example, to identify that states in a Statechart or statements in a method body cannot be reached—or *data flow analyses* to ensure that values are assigned to variables before they are used, which is a typical part of modern compilers, for example.
4. Conditions that cannot be verified automatically are usually more complex relationships between modelling elements. For example, multiple attributes of a class can be in an internal relationship (e.g. `b==2*a`) that can be formulated with an OCL constraint. Based on this relationship, where applicable, an attribute `b` can be replaced by an expression over another attribute `a`. However, normally it is not possible to verify automatically that OCL constraints are correct; instead, specific tests or interactive verification are required.

In compiler construction, the category of syntactically verifiable context conditions is typically divided into the subcategories of (simple, context-free)

syntactical and (more complex, performed later) semantic analysis. However, both forms are performed based on the syntax and are executed automatically.

For some context conditions, there is only a semi-decidable procedure in which under some circumstances, the validity of the context condition cannot be decided. In this case, the context condition is rejected if it cannot be decided positively. This regularly happens when decidability has an exponential complexity or a property is indeed undecidable.

There are several frequently occurring usage conditions that are already familiar from other approaches for transformational software development:

Termination: A context condition can require that the calculation of an expression always *terminates*. Here an exception is also count as termination. As discussed in Chapter 3, Volume 1, the termination can be verified relatively easily by tests, particularly with given loop invariants and termination conditions, even though it is undecidable in principle. However, for transformations that are relevant from a practical perspective, we can assume that each expression terminates or the tests available discover a nontermination.

Definedness: A calculation is *defined* if it always terminates and produces a normal result—that is, not an exception.

Determinism: A calculation is *determined* if it always terminates and produces the same unique result. This result can be an exception. It is important that no random element occurs during the calculation or at least has no effect on the result. This condition can be violated by the integration of time queries, for example. As discussed in Section 3.3.4, Volume 1 with the conversion of sets into lists with the OCL operator `asList`, the result of this operator may well be unique but the result is not known to the developer a priori. The advantage of this definition is that on the one hand, an expression of the form `set.asList` is determined; but on the other hand, the specific implementation is the responsibility of an OCL interpreter.

Side effect-freedom: A side effect of a calculation is a permanent change of the state of an existing object structure as a result of a change to a local variable, an attribute, or a link. As discussed in Section 3.4.1, Volume 1, the creation of new objects is only a side effect if these objects are accessible from the original object structure.

Context conditions that contain semantic equivalences are often undecidable. For example, when one expression is replaced by another, as in the example rule in Section 9.1, the equality of the two expressions cannot be tested automatically. This type of property is represented by OCL constraints, for example. Therefore, we can only use these conditions for *tests* or for *verification*. Although tests do not provide full certainty about the correctness of an OCL constraint, they can be executed more efficiently than a real verification.

To verify this type of invariant, in most cases we have to augment the code with further invariants and, similar to the Hoare logic, verify all individual steps (a Java variant can be found in [vO01], for example). For systems of the highest quality or particularly critical areas, this can be useful particularly in combination with tests. The tests first detect and eliminate any errors that are present. Verification techniques are then used to prove the complete correctness and any residual risk of a defective invariant is eliminated. By applying tests first, we can save verification effort for a lot of the incorrect conditions and thus progress more efficiently.

However, for many systems, the use of tests will be sufficient. As shown in Section 10.2 using an approach for changing a data structure, we can use tests not only as an indicator that a system is correct, but also to indicate that a transformation is correct.

9.3.6 Transformational Software Development Approaches

[Pep84, Chapter 5] contains some transformational software development approaches as well as an interesting discussion about their effects. Amongst other things, it discusses the semantics, the benefits of top-down representations of a software development, and the language independence of transformation techniques.

Transformational Software Development

In theoretical information technology in particular, a number of transformational approaches based on refinement concepts have already been presented. These include, for example, work from Dijkstra [Dij76], Wirth [Wir71], Bauer [BW82], Back [BvW98], and Hoare [HHJ^{+}87]. CIP-L [BBB^{+}85], for example, is a language that combines an algebraic specification style with functional, algorithmic, and procedural programming styles and, via numerous transformational steps, can go from one language style to the next. The basic methodology of this top-down transformation approach is that firstly, we first model in an abstract specification language; this is then transformed into a functional language; finally, it is optimized towards a mapping into a procedural implementation language. Elements of this transformational software development approach can also be found in today's refactoring, which is part of many incremental, iterative development approaches. This includes, for example, the concept of optimizing the code as late as possible only after the correctness has been clarified and the stability of the functionality ensured. In [BBB^{+}85], as in related approaches, verification was used as a mechanism for ensuring that a transformation is correct. In the approach proposed here, verification is replaced by tests that require less effort.

The approach for specifying and transforming programs given in [Par90] also contains a detailed collection of rules for transformational software development for abstract data types defined by algebraic laws, functional and

imperative programs, and data structures. This work discusses, for example, the removal of superfluous assignments and variables, the reorganization of statements, the handling of control structures, or different variants of the composition of functions that can be found to some extent in direct correlation in [Fow99]. Furthermore, [Par90] and [BBB+85] offer techniques for refining and transforming a specification incrementally towards an operational implementation.

Algebraic Specification

In the world of algebraic specification techniques, one important step was to introduce the definition of explicit observations. By *hiding types*, OBJ [FGJM85], for example, offers a mechanism for encapsulating details of an algebraically specified abstract data type and providing an explicitly usable and observable interface. Further algebraic approaches [ST87, BHW95, GR99, BFG+93] demonstrate that it is possible to define *externally visible behavior* explicitly and to apply rigorous methods of proof to this behavior. [BBK91] contains a general overview of the approaches for defining observability in algebraic specifications.

As already discussed, a disadvantage of the test approach used here, even though it is very flexible, is that the observation and thus also the observed interface are defined only implicitly by the test and are therefore defined differently for each test. The test developer must be very disciplined, particularly in acceptance tests, and access only stable and, as for the algebraic specifications, explicitly "published" interfaces as far as possible.

Refactoring and Verification

In [Sou01], refactoring and verification are combined: in a first step, a collection of proofs is used to ensure that a system behaves correctly. Developing the system further using small, systematic steps then leads to adapting the proofs accordingly. This ensures the system as well as the proofs are still correct and can automatically and repeatedly be checked. With adequate tool support, as offered, for example, by Isabelle [NPW02] and their proof tactics, which are very strong in some cases, we can perform this type of evolutionary development by reusing verification parts.

Transformation of Graphical Specifications

A number of works show that transformational techniques can also be developed for graphical specifications, regardless of whether they represent structure views, behavior views, or interaction views of a system. The *Focus* approach described in [BS01b] demonstrates how we can combine formal, text-based specification techniques with a graphical representation of

the distributed interaction of components. This approach contains precise techniques for the decomposition of components and channels and for the refinement of behavior and interfaces. The effects of these techniques can be designed and studied on graphical models.

In this context, [PR97] and [PR99] presented a technique for a glass box transformation of the internal structure of a distributed system whose observable behavior at the interface remains equivalent or is refined during the transformation. One part of this transformation technique is based on the behavior refinement of internal components which, for example, can be described with state automata [Rum96].

[RT98] shows that these forms of transformation are suitable for more than massively distributed or hardware-like systems by optimizing business processes with structural transformations.

Summary

To summarize, we can observe that the semantics-preserving *refactoring* is based on a *notion of observation* that is based on a *test suite*. Two generalizations of the transformation of models—*abstraction*, and in particular the *refinement*—could be used not only to restructure the existing system description but also for the purposes of transformational development. Approaches such as those described in Section 9.3.6 [BBB$^+$85, Par90] have shown this. From a practical perspective, transformational development steps can also be applied when enough *automated tests* exist to check the behavioral identity of an evolving system.

9.3.7 Transformation Languages

In order to be able to use the types of transformations mentioned flexibly, we need a separate language to define transformations. This is the only way to define transformations explicitly.

In mathematics, replacement rules are defined in the form of equations. We can therefore use the equation language of mathematics to define replacements. In the formal methods, one use of replacement rules is to specify typing rules in the form of a calculus (as is the case in CIP [BBB$^+$85, BEH$^+$87]) to define algebraic transformations of textual programming and specification languages. Alternatively, (as is the case in Isabelle [NPW02, Pau94]) replacement rules can be used to specify theorems as applicable transformations using application conditions. In mathematics, the replacement rules are part of the "mathematical language" itself. In contrast the calculus languages of Isabelle/HOL or CIP are separate sublanguages that fit harmoniously with the basis language that is to be transformed.

In analogy to textual grammars, graph grammars [Nag79, NS91] allow us to define graph structures and are particularly suitable for saving the structure of UML models typically defined as graphs (in the mathematical sense)

in tools. On this basis, we can define transformations for graphs, for example, using graph replacement systems [Sch88, Sch91, LMB+01], algebraic approaches [EEPT06, Tae04], or a notation for controlling the transformations by means of Fujaba's "Story Diagrams" [FNTZ00]. The graph replacement system PROGRES [Sch91, Zün96] contains not only graph-matching techniques but also a complex control mechanism for executing and for backtracking replacement rules. The concept of the triple graph grammars [Sch94] also allows graphs to be transformed between different structures, thus also allowing a transformation between models of different languages (such as from automata to Java).

Most modern approaches for defining transformations on models use metamodels in the form of object structures. Here, class diagrams are used similarly to the description in Chapter C, Volume 1 to define the abstract syntax. We can then define transformations on the abstract syntax respectively on the object structures and thus circumventing the specific representation of the models. Transformation languages such as ATL [JK05], MOLA [KCS05], BOTL [MB03], or Epsilon Transformation Language [KRP11] use this approach which is frequently based on EMF [SBPM08]. [CH06] contains a good classification of these types of transformation languages. Under the name "Query View Transformation (MOF QVT)", OMG has defined a specification for such transformation languages [OMG08]. Graph replacement systems such as MOFLON [AKRS06, WKS10] now build on this specification. It is generally accepted in the software language engineering community [CFJ+16] that transformations are an essential technique to deal with languages.

While metamodel techniques and some graph transformation approaches [LKAS09] rely primarily on the generic, language-independent definition of transformations in an analogy to mathematics, other techniques try to develop a transformation language that is aligned as closely as possible with the underlying modeling language [BW06, Grø09, RW11, HRW15]. The advantage of a transformation language based on the specific syntax is that it is more comprehensible for the user. Hence, users prefer such a specific transformation language rather than using a transformation language that is completely unknown or requires to deal with the abstract syntax of the modeling languages.

Selecting a suitable transformation language and related tools for defining refactoring steps is necessary in order to define the transformations covered in Chapter 10.

10

Refactoring of Models

Human actions sustain in their impacts.
according to Gottfried Wilhelm von Leibniz

Based on the core principles for model transformations and the specialization of these transformations as refactoring rules, this chapter discusses possible forms of refactoring rules for UML/P, the transfer of rules from other languages to UML/P, and a *superimposition approach* for transforming larger data structures.

10.1 Sources for UML/P Refactoring Rules ... 286
10.2 A Superimposition Method for Changing Data Structures ... 306
10.3 Summary of Refactoring Techniques ... 320

© Springer International Publishing AG 2017

B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_10

Similarly to mathematical theories, there exists an unlimited number of refactoring rules. Therefore, in addition to an available set of basic rules and a selection of more complex rules for specific problems, it is worthwhile knowing mechanisms that allow us to reuse existing refactoring rules from other languages as well as combine rules or adapt them for special purposes.

This chapter first investigates refactoring rules existing in other languages and how they can be transferred to UML/P. It then uses examples to discuss more complex refactorings based on this initial investigation. In the same way as for test patterns, the discussion explains the advantage of independent adaptability compared to a detailed list of rules.

Section 10.1 discusses mechanisms for creating refactoring rules and for transferring existing refactoring rules to UML/P. Suitable sources of refactoring rules include Java as the underlying programming language for UML/P, the automata theory for Statecharts, and verification techniques for OCL.

Building on this discussion, Section 10.2 presents an approach for changing data structures modeled with class diagrams. This approach is based on OCL invariants and allows us to develop and execute larger refactoring processes. To do this, it takes ideas from verification by establishing predicates as explicitly formulated relationships between the data structures to be adapted. Furthermore, it uses automated tests to check that the relationships are correct.

10.1 Sources for UML/P Refactoring Rules

The previous chapter discussed model transformations and their theoretical aspects in general. We will now apply these considerations to specific examples. We will primarily discuss mechanisms and sources for transferring refactoring rules to UML/P from known approaches.

Refactoring steps executed on UML/P generally affect multiple sublanguages. For example, when shifting a method in a class diagram, parts of the Java/P code, the sequence diagrams, and even Statecharts may be affected. Therefore, we cannot discuss refactoring techniques for UML/P for individual notations in complete isolation.

We have to select the granularity of a refactoring such that the transformation steps remain manageable and the automated tests can be executed regularly. Steps that are too large or that are not sufficiently supported by tests lead to a big bang syndrome involving a lot of effort for identification of errors. This means that, as far as necessary, we should break large refactorings down into small steps and create a plan for their application. Nevertheless, there are a lot of necessarily *larger refactorings* that cover very specific problems.

In comparison to Java programs, UML/P is relatively compact. This is because UML/P allows a separation between technical code and application-specific code, generates auxiliary methods automatically, and, due to the sep-

aration into different views, gives developers a better overview. This compactness of UML/P allows us to control larger refactoring steps than possible with Java. For example, a class diagram supports the planning of the refactoring and the use of OCL invariants allows us to model and check assumptions made for a refactoring. This leads to the approach discussed in Section 10.2 for performing larger changes of the data structure using refactoring—in that approach, several UML/P notations are in use.

While collections of smaller and medium-sized refactorings are already available for object-oriented programming languages such as Java or Smalltalk (for example, with [Fow99]), this type of refactoring steps is only just being set up for modeling languages such as UML/P. [Dob10] gives an overview of techniques for refactoring UML models which concentrate primarily on class diagrams and executable variants of UML.

The most elaborate transformations are those applied to class diagrams [SPTJ01, Ast02, GSMD03]. [Ast02], for example, describes an approach that uses UML class diagrams to support refactoring of Java programs. In this approach, class diagrams are extracted from existing code in order to identify code deficits.

Graph grammars [Nag79] and tools based on graph grammars offer an excellent basis for transformation approaches. Early approaches of this type for the graphical modeling language UML are described in [EH00b] and [EHHS00]. In [EHHS00], these transformations are even used to describe dynamic system runs and their execution steps.

In current literature, UML diagrams are still rarely discussed with the primary goal of refactoring. However, the application of refactoring in particular to constructive description techniques, such as class diagrams, Statecharts, and OCL is interesting. In contrast, adapting exemplary descriptions, such as object and sequence diagrams, is comparatively easy. As these diagrams are used primarily for defining tests, we usually have to adapt them, especially if a test fails after a refactoring. In such cases, it can be useful to develop several new tests from a failed test. For example, if a single method call was replaced by a series of connected method calls then various forms of allowed call orderings should be tested.

The transformations of Statecharts that do not affect the observable behavior were discussed in detail in Section 5.6.2, Volume 1. A collection of goal-oriented rules were presented that allow us to simplify Statecharts, for example, by flattening hierarchical states. Many of these rules can also be applied in the reverse direction. Some rules, however, such as the reduction of nondeterminism in Statecharts, are a true refinement in the sense defined in Section 9.3.2. Based on the economic aspects described in Section 9.2, we should apply refactoring techniques for Statecharts in particular for systems or system parts with complex state spaces and a high level of criticality. These include, for example, avionics systems, safety protocols, or complex transaction logics in bank systems.

The goal of this section is to discuss how sets of rules for the modeling language UML/P from other refactoring approaches can be adopted and which refactorings already exist. It is not the goal of this section to develop a complete catalog. Instead, this section uses selected examples to demonstrate how to define refactoring rules for UML/P and what effects can be achieved. It thus provides a technique for developing refactoring rules independently, rather than a predefined catalog of refactorings. In particular, this technique includes the approach discussed in the next section for refactoring data structures—an approach that can be used to extract new refactoring rules from specific applications.

10.1.1 Defining and Representing Refactoring Rules

Similarly to a transformation for code generation, we can describe a refactoring in two ways. A technical, detailed, and precise description is particularly suitable for implementation in tools. As the basic prerequisite, this implementation requires the abstract syntax as well as context conditions that are defined precisely on this abstract syntax.

A second form of representation that is introduced in this section is denoted as methodical guideline. The second form is often using a relatively general example that does not always cover all cases. Context conditions are then discussed more informally (although precisely) and special cases are explained. As usual for patterns, specific examples are given. For larger refactorings in particular, it is useful to discuss the consequences, advantages, and disadvantages. Finally, references to related refactorings as well as reversing refactoring are given.

The format for representing a refactoring is therefore based on the format for code generation from Table 4.11 and is presented as a template in Table 10.1.

In [Fow99], refactoring rules are universally represented as a triple: *motivation*, *mechanics*, and *example*. The *mechanics* section describes an operative list of individual steps that are useful for performing the refactoring. Special cases are also discussed.

Template for representing refactoring rules	
Problem	What is the problem that we want to eliminate with this rule?
Goal	If not already clear from the motivation and problem description, the goals of the refactoring are described again here.
Motivation	When will the refactoring rule be used and why? What improvements are achieved for the software development?

(continued on the next page)

(continues Table 10.1.: Template for representing refactoring rules)

Refactoring	There is usually a primary transformation rule for the refactoring which is represented in the following form: `Origin` ⇓ `Result` • The representation of a refactoring transformation is analog to the representation of generation transformations described in Table 4.11. `Origin` and `Result` are explained there. • Explanatory texts describe context conditions, special cases, and how these are handled. • If the refactoring is split into multiple individual steps, these are described here. These individual steps are also referred to as the "mechanics".
Further refactorings	To be able to apply the primary transformation, there are usually additional required transformations that are represented in the same way.
Implementation	Technical details such as the modification of method bodies are presented and explained here.
Examples	Examples can be used to explain the principle and to discuss special cases.
Noteworthy	With additional considerations, notes, and the discussion of potential problems, this section completes the description. In particular, it refers to the consequences, advantages, and disadvantages of the refactoring described.

Table 10.1. Template for representing refactoring rules

Now that we have clarified the reasons for and the background of refactoring and the appearance of refactoring rules, Fig. 10.2 provides additional terminology based on Fig. 9.1.

10.1.2 Refactoring in Java/P

In terms of syntax, there are only minimal differences between the Java/P programming language embedded in UML and defined in Appendix B, Volume 1 and the official Java standard. The most significant difference is that in accordance with the code generation described in Chapter 4 and Chapter 5, the code bodies from Java/P are subject to transformations. For example, attribute access is converted into `get` and `set` methods. Attributes are therefore always encapsulated and some of the refactorings defined in [Fow99]

The basic definitions for refactoring:

Refactoring: A technique for transforming models based on rules that preserves the externally observable behavior. A refactoring can be split up into a series of individual steps.

Refactoring rule: A goal-based rule for performing refactoring. It contains a series of observation-invariant model transformations formulated with schema variables that are to be applied to the initial model, reasons for doing so, and a discussion of the effects.

Refactoring step: The application of a refactoring rule at a specific point.

Observation: A test consisting of multiple models that describe the properties required by the system.

External observation: An observation that must not be changed without requiring project-external consultations. External observations are defined by acceptance tests and tests of interfaces to neighboring systems, fixed frameworks, etc.

Fig. 10.2. Terminology definitions for refactoring

are unnecessary in Java/P. For example, the refactoring rule "Encapsulate Field" [Fow99, p. 206] is performed automatically by a code generator. This has the advantage that, although the encapsulation is ensured, the encapsulation methods remain hidden to the developer while modeling.

Other refactoring rules can be adopted from [Opd92] and [Fow99]. Thesis [Opd92] defines 26 low-level refactorings for C++—three of these are intended for creating and deleting program elements (classes, functions, attributes), fifteen for adapting existing program elements, and two for moving program elements. Three further refactorings are compositions of previous transformations. Three additional refactorings are used to generalize and specialize the class hierarchy and to handle aggregation and thus have an effect beyond individual classes. [Fow99] contains corresponding analogies for Java. The transferability of the individual refactoring rules from [Fow99] to Java/P or to UML/P diagrams is discussed below.

Refactorings in [Fow99]

[Fow99] contains 72 refactoring rules that are referred to as an initial and incomplete refactoring catalog suitable for extension. The catalog has indeed been extended occasionally, e.g., in discussion forums. The number of composite and complex refactorings based on Java has thus grown. It is not surprising, however, that the number of basic refactorings has barely changed in recent years. This is because a basic refactoring handles only a very small number of language concepts. Accordingly, the number of basic transformations applied to a language is limited and [Fow99] remains the main source for refactorings.

Therefore, in the following, we will look at 68 of the refactoring rules published in [Fow99] in a compact way. The analysis discussed here assumes that the reader is familiar with [Fow99]. The four rules not presented are referred to as *"big refactorings"* in [Fow99]. They deal with separating complex inheritance hierarchies or modifying procedural code in object structures, for example.

We classify the rules according to two criteria: (1) Which elements does the rule affect? (2) What impact does the rule have on those elements? The six columns correspond to the Java language elements *code bodies*, *methods including constructors*, *attributes*, *class signatures including interfaces*, *inheritance relationships*, and *associations*. The effects noted in the corresponding columns are *move (***m***)*, *introduce (∗)*, *delete (†)*, and *change (ch)*.

The table reflects the changes caused by the primary refactoring rule. Further changes to other language elements may occur in special cases.

Classification of the refactoring rules from [Fow99]

Name of the refactoring rule	Code	Methd.	Attrib.	Cl.sig.	Inher.	Assoc.
Add Parameter	*ch*	*ch*		*ch*		
Change Bidirectional Association to Unidirectional	*ch*		†			*ch*
(simplified in UML/P by the generator)						
Change Reference to Value	*ch*					*ch*
Change Unidirectional Association to Bidirectional	*ch*		∗			*ch*
(simplified in UML/P by the generator)						
Change Value to Reference	*ch*					*ch*
Collapse Hierarchy		**m**	**m**	†	†	**m**
Consolidate Conditional Expression	*ch*	∗				
Consolidate Duplicate Conditional Fragments	*ch*					
Decompose Conditional	*ch*	∗				
Duplicate Observed Data	*ch*	∗	∗	∗	∗	∗
Encapsulate Collection	*ch*	∗				
Encapsulate Downcast	*ch*	*ch*				
Encapsulate Field	*ch*	∗	*ch*			
(unnecessary if the generator does that)						
Extract Class		**m**	**m**	∗		∗
Extract Interface				∗	∗	
Extract Method	**m**	∗				
Extract Subclass		**m**	**m**	∗	∗	
Extract Superclass		**m**	**m**	∗	∗	
Form Template Method	*ch*	∗		*ch*		
Hide Delegate	*ch*	∗		*ch*		†
Hide Method		*ch*		*ch*		

Continued on the next page

(Continuation of table 10.3.: Classification of the refactoring rules from [Fow99])

Name of the refactoring rule	Code	Methd.	Attrib.	Cl.sig.	Inher.	Assoc.
Inline Class		**m**	**m**	†		†
Inline Method	**m**	†				
Inline Temp	*ch*					
Introduce Assertion	*ch*					
(in UML/P there are further options)						
Introduce Explaining Variable	*ch*					
Introduce Foreign Method	*ch*	*				
Introduce Local Extension		*		*	*	*
(uses an adapter or a subclass)						
Introduce Null Object	*ch*	*		*	*	
Introduce Parameter Object	*ch*	*ch*		*		
Move Field	*ch*		**m**			
Move Method	*ch*	**m**		*ch*		
Parameterize Method	*ch*	*ch*		*ch*		
Preserve Whole Object	*ch*	*ch*		*ch*		
Pull Up Constructor Body	**m**	*				
Pull Up Field			**m**			
Pull Up Method		**m**				
Push Down Field			**m**			
Push Down Method		**m**				
Remove Assignments to Parameters	*ch*					
Remove Control Flag	*ch*					
Remove Middle Man	*ch*	†		*ch*		*
Remove Parameter	*ch*	*ch*		*ch*		
Remove Setting Method		†	*ch*			
Rename Method	*ch*	*ch*				
Replace Array with Object	*ch*			*		
Replace Conditional with Polymorphism	*ch*			*	*	
Replace Constructor with Factory Method	*ch*	*		*ch*		
(unnecessary if the generator does that)						
Replace Data Value with Object	*ch*		*ch*	*		*
Replace Delegation with Inheritance	*ch*	†		*ch*	*	†
Replace Error Code with Exception	*ch*	*ch*				
Replace Exception with Test	*ch*					
Replace Inheritance with Delegation	*ch*	*		*ch*	†	*
Replace Magic Number with Symbolic Constant	*ch*		*			
Replace Method with Method Object	*ch*	*	*	*		*
Replace Nested Conditional with Guard Clauses	*ch*					
Replace Parameter with Explicit Methods	*ch*	*		*ch*		
Replace Parameter with Method	*ch*	*ch*		*ch*		

Continued on the next page

(Continuation of table 10.3.: Classification of the refactoring rules from [Fow99])

Name of the refactoring rule	Code	Methd.	Attrib.	Cl.sig.	Inher.	Assoc.
Replace Record with Data Class *(not relevant for UML/P)*				*		
Replace Subclass with Fields	*ch*	*ch*		†	†	
Replace Temp with Query	**m**	*				
Replace Type Code with Class	*ch*		†	*		
Replace Type Code with State/Strategy *(a Statechart would also be useful)*	*ch*		†	*	*	*
Replace Type Code with Subclasses	*ch*		†	*	*	
Self Encapsulate Field *(unnecessary if the generator does that)*	*ch*	*				
Separate Query from Modifier	*ch*	*		*ch*		
Split Temporary Variable	*ch*					
Substitute Algorithm	*ch*					

Table 10.3.: Classification of the refactoring rules from [Fow99]

In almost all refactoring rules, the class that contains the elements modified, introduced, or deleted is affected. Therefore, the corresponding column is only labeled if a part of the methods known externally (`public`) is subject to a modification.

In contrast to all other rules, the refactoring rule *"Encapsulate Collection"* describes how to handle container classes and is therefore dependent on the Java class library. This example shows that refactoring rules that operate on the class libraries as well as on the language are necessary.

As we can see from the table, many of the refactorings in [Fow99] comprise several steps. Thus, if we expand a method with *"Inline Method"*, [Fow99] proposes deleting this method. This transformation is only applicable when the expanded method is not used anywhere else. Alternatively, the transformation could have been split into two independent refactorings.

The level of detail of many of the rules—such as *"Form Template Method"* for extracting a method into a superclass that is realized in similar form in multiple subclasses—contains a goal oriented approach for improving the structure how methods call each other. These types of rules often use other rules such as the moving of methods in the class hierarchy. Thus the refactoring techniques described in [Fow99] are often not atomic, but a combination of several transformations into a goal-oriented larger transformation is necessary. Furthermore, formal algebraic transformations often are required to simplify the code, but are also not discussed very explicitly and in detail.

Transferring the Refactorings to UML/P

We can identify several approaches for creating transformation rules. For example, we can create transformation rules based on the theory presented in Fig. 9.5. This theory investigates the existing language and identifies equivalent or more refined representations of program expressions. In contrast, we can create transformation rules for a new language by adapting rules of a known language. To do this, we can apply the approach presented in Fig. 10.4.

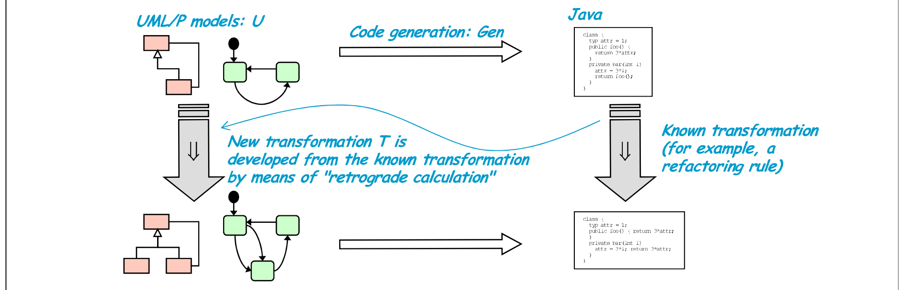
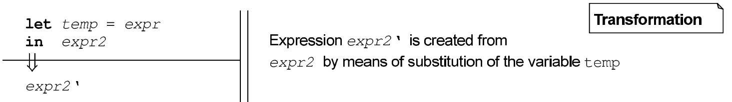


Fig. 10.4. Refactoring rules can be transferred

With this approach the refactorings for Java contained in [Fow99] can be transferred to UML/P. The basis for this is the code generation discussed in Chapter 4 as the connection between UML/P and Java. We can use this code generation to convert the refactoring rules that exist for Java to UML/P. However, some of the rules can then become obsolete, as indicated in Table 10.3. Other rules can be transferred in multiple forms. The greater the conceptual difference between Java and UML/P, the more complex the backward propagation of the transformation rules to UML/P becomes. For class diagrams and Java, the difference is very small and the backward propagation is therefore largely canonical. However, the conceptual difference between Statecharts and Java is so large that completely separate transformation rules for Statecharts are necessary.

OCL and Java have many common language concepts. Therefore, a number of the refactoring rules described in [Fow99] can be transferred to OCL. In OCL, for example, temporary variables are defined with the **let** construct. We can also expand or transform these variables. In OCL, we extract new methods by defining queries in the underlying class diagram. For example, we can describe the rule "*Inline Temp*", which expands a variable used temporarily with the following rule:



With this rule, every occurrence of the variable *temp* in the expression *expr2* is replaced by the subexpression *expr*. In OCL, many of the context conditions that are necessary in Java are automatically satisfied due to the lack of side effects and the determinism.

Other rules from [Fow99] modify the structure or the signature of classes or add new classes. These rules can therefore also be applied adequately to class diagrams (and the dependent Java/P code bodies). Again, the set of rules is in no way complete. For example, "*Hide Method*" is a rule for converting a public method into a private method but the inverse transformation is so simple that it is not covered by a rule. Other rules cover converting a class into an interface or modifying attached stereotypes, for example.

Rules which change the signature of a method or an attribute also have an impact on places in tests, Statecharts, or sequence diagrams where these elements are used. For example, we have to adapt a test used for a sequence diagram if the internal call structure changes and stereotypes such as ≪match:complete≫ require this in a sequence diagram (see Section 6.3, Volume 1).

10.1.3 Refactoring Class Diagrams

In Section 10.1.2, in the context of Java/P we identified a number of refactoring rules that modify classes and thus also have an effect on class diagrams. However, there are further refactoring rules for class diagrams. The rules for transforming class diagrams can be divided into the following categories:

Small refactorings are used to handle single elements of a class diagram—these include deleting or adding an attribute, for example.

Goal-oriented, medium-sized refactorings are transformation steps that include a motivation and a goal. We discussed this form of refactoring step for Java in Section 10.1.2. In most cases, it affects several model elements including some outside an individual model. However, the impact of these rules is locally limited.

Improving the representation allows developers to access information better without changing the content. It includes, for example, reorganizing class diagrams.

Abstract class diagrams can be used to describe interfaces and components. This type of class diagram contains only that part of the data structure and methods that was *published* explicitly and is thus available in a stabilized form. We can obtain this type of class diagram from the internal structure using abstraction steps.

We will now discuss these categories of transformation rules individually.

Small Refactorings

Many transformation rules only consist of the transformation of one, single syntactical element. The syntax of class diagrams is described in Appendix C, Volume 1 and consists of 19 nonterminals. For each nonterminal, we can introduce a new element, delete an existing element, or modify parts of an existing element. This includes, for example, renaming an attribute, an association, or a role, refining a cardinality, modifying a visibility, changing a navigation direction, introducing or eliminating a qualifier, and replacing the type of a variable. Because these modifications—which focus on one syntactical element—are relatively small and can be applied canonically, these refactorings are not listed here. However, some of these modifications have context conditions or require further transformations before they are used.

Medium-sized and large refactorings are goal-oriented and are defined based on practical experience. In most cases, they modify several elements and in some circumstances, they even modify a significant part of an application.

Goal-Oriented Refactorings

As Table 10.3 shows, the rules described in [Fow99] affect several elements in most cases and combine goal-oriented strategies for improving the design. In principle, for example, we could break down the migration of an attribute into the following individual steps: (1) Introduce new attribute, (2) Modify the points that use the attribute, and (3) Delete the old attribute. However, we only create a strategy with a goal by combining these individual steps.

Nevertheless, the rule for migrating attributes in [Fow99] has a deficit when applied which we can eliminate by using invariants, for example. T We have to ensure that there is a unique navigation path between the old and the new class of the attribute which is not subject to any temporal changes.

The migration of an attribute between two classes demonstrated below is a simple example for the interaction of multiple UML/P notations which can be coordinated by using a class diagram.

Simple special cases of this migration are that the new class acts as the superclass or the attribute is static and therefore exists only once. In most cases, however, there are two classes whose objects are connected to one another via a potentially complex navigation path. Starting from the situation represented in Fig. 10.5(a), the specified attribute is to be moved from `A` to `B`.

On the one hand, we must ensure that there is a suitable relationship between the objects of both classes. In most cases, we can represent this with an expression `a.`*exp* for each object `a:A` which leads uniquely to the objects of class `B`. This expression can contain complex navigation paths and method calls but has no side effects. As a result of the derived association `connection` shown in Fig. 10.5(b) and the OCL constraint `Connect`, this

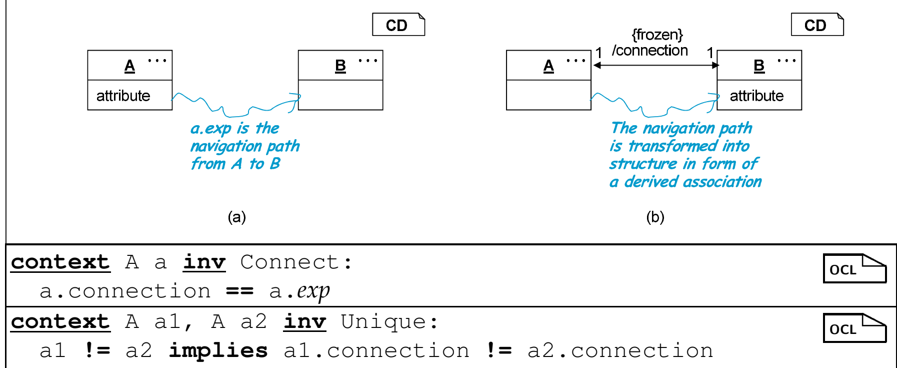


Fig. 10.5. Moving an attribute between classes

expression is included in the structure of the class diagram and is thus accessible for easier handling. In particular, it is now sufficient to apply the tag {`frozen`} to the derived association to ensure that the objects of class `B` are not replaced, as this would otherwise mean that the outsourced attribute would implicitly change its content.

On the other hand, we must ensure that each `A` object still has its own outsourced attribute value. Accordingly, the OCL constraint `Unique` requires that each `A` object is linked to a separate `B` object.

The conditions compiled act as context conditions for transferring an attribute to another class. The result is summarized in the refactoring rule given in Table 10.6.

As demonstrated in the refactoring rule, invariants can be omitted at the end of the refactoring or in a subsequent evolution step of the system. This is necessary, for example, if the goal of the refactoring is to realize the replacement of the attribute content `attribute` in the future by reassigning `B` objects.

Refactoring: Migration of an attribute	
Problem	An attribute belongs to the wrong class. The attribute value is individual for each object of the class and the classes are not in an inheritance relationship.
Goal	The attribute will be moved to the new class.
Motivation	The attribute is much more used by the other class, it is useful to move the attribute there.

(continued on the next page)

(continues Table 10.6.: Refactoring: Migration of an attribute)

Refactoring	A ... attribute — B ...
	 a.exp is the navigation path from A to B
	 CD ... // Code `a.attribute` Java/P
	 ⇓
	 {frozen} /connection 1 1
	 A ... — B ... attribute
	 CD ... // Code `a.exp.attribute` Java/P
	 OCL
	 `context A a inv: a.connection == a.exp;`
	 `context A a1, A a2 inv: a1 != a2 implies a1.connection != a2.connection`
	 ⇓
	 A ... — B ... attribute
	 CD ... // Code `a.exp.attribute` Java/P
	
	 • First, the navigation path `a.`*exp* is identified.
	 • This path has to satisfy two conditions described by the derived association and the invariants and can thus, e.g., be tested.
	 • If the attribute is declared as `private`, access methods must be introduced where applicable.
	 • When the attribute is moved, all access paths are adjusted at the same time. Algebraic simplifications of expressions become possible here.

Special cases	If the attribute is static or is moved upwards in the inheritance hierarchy, the additional association is not necessary.
Noteworthy	Generalizations are possible if, for example, multiple `A` objects with the same attribute content share a common `B` object.

Table 10.6. *Refactoring:* Migration of an attribute

The temporary definition of a navigation path from class `A` to class `B` is often omitted in literature; in Java itself, representing this requires a great deal of effort. Due to its language richness, UML/P is much more suitable here.

Improving the Form of Representation through Refactoring

As already discussed in Section 2.4, Volume 1, there are many facets to the relationship between a model and an implementation. For example, syntactically different models can be semantically identical and when converted into code, lead to the same system. Therefore, the differences in these models relate only to their representation, and not to their implementation. [MRR11e] describes suitable algorithms for recognizing semantically equivalent models and for representing their semantic differences.

A standard example for equivalent models is the merging of class diagrams discussed in Section 2.4, Volume 1: from two or more subdiagrams, we develop an overall model that contains the same information.

The *migration* of information from one class diagram into another or the *splitting* of class diagrams are similar steps that are sometimes applied during development. Splitting is suitable, for example, if a class diagram becomes overloaded due to the repeated addition of functionality and structure in the form of new classes, methods, and attributes. Splitting is also interesting if the system snapshot represented in the diagram can be divided into two relatively independent subsystems which, in the further course of the project, are to be processed by separate developer teams.

Migrating classes between diagrams also helps to improve the representation of the model. Detailed information about individual classes, such as attributes or methods, can be migrated between diagrams if the diagrams have overlapping parts.

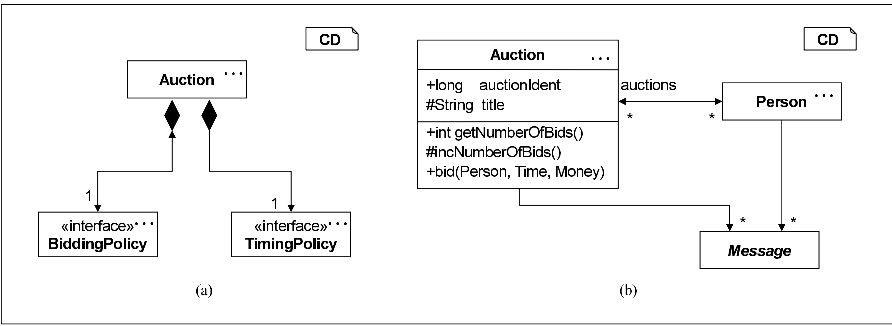


Fig. 10.7. Migration of detailed information in class diagrams

In this situation, it is important to differentiate between migrating an attribute or a method from one class to another and migrating information between class diagrams. In our example, the attributes and methods remain in the same class and are merely represented in a different place.

Another form of editing of class diagrams is the *expansion* of the detailed information of classes. For example, information available from other diagrams can also (redundantly) be represented in a diagram without this information being removed at another point.

The examples discussed demonstrate that we can use refactoring not only to improve the system structure but also to represent the structures of a system in a different way. We can also observe this phenomenon in some of the refactorings discussed in [Fow99] if, for example, the proposal is to change the name of a method so that the name is a better reflection of the content of the task of the method. However, in [Fow99], refactorings often affect presentation and structure simultaneously. Splitting a class thus improves the

presentation of the class to the developer but also modifies the structure of the system.

The syntactical richness of UML/P is one of the reasons for the increased demand for an improvement in the representation of models. In the programming language Java, the variability of the source code is limited to the order of the methods and attributes represented, insertions, algebraically equivalent transformations of expressions, and so on; in UML/P, however, there are more variants for representing the same information. One reason for this is that the definition points for attributes and methods are not defined uniquely—instead, they can be located in various different class diagrams. There are generally also a number of semantically equivalent forms of representation for OCL constraints. For example, the hierarchy, transitions, and states in Statecharts can be manipulated by the rules introduced in Section 5.6.2, Volume 1.

Using Abstract Class Diagrams to Define Interfaces

On the one hand, the syntactical richness of UML/P offers the advantage that we can select the appropriate, compact form of representation for each situation; on the other hand, it leads to the problem that it is more difficult, for example, to find the point of definition for an attribute and therefore sufficient tool support is necessary. This problem must be regulated by a good modeling standard. It has shown to be helpful, for example, to use a detailed class diagram for each subsystem that is not divided further and that lists all attributes and methods with their signatures. Further class diagrams are used to represent connections between subsystems. These contain only a subset of the existing classes and associations and mostly ignore detailed information. As discussed in [HRR98], we can also use a class diagram as an interface for the parts of a component that are accessible from outside. This type of class diagram then also represents an abstraction of the actual model of the component and is sufficient to allow the developer to use the component.

The different forms of class diagrams can be derived from one another via systematic transformations, which can also be called refactoring steps. Thus, we can use techniques for merging diagrams, migrating or expanding detailed information, and in the opposite direction, removing redundantly available information. What is important, however, is that for class diagrams in particular, the form of use must be explicated through suitable stereotypes. The representation indicators "©" and "..." are suitable, for example, for indicating whether the detailed information represented is complete or incomplete.

For pragmatic reasons, however, we should try to keep the redundancy between different representations of the same information as low as possible. Redundancy often leads to inconsistencies if we modify a part of the

system, for example, with refactoring steps and cannot use a tool to ensure this consistency automatically. In that case, redundancy increases the amount of work involved in implementing changes. On the other hand, redundancy used cleverly is an important means for executing consistency tests and checks. This includes the redundancy between a test model and the implementation, but also, for example, a ("published") class diagram disclosed as an interface of a component that represents an abstraction of the implementation and has to be consistent with this implementation.

10.1.4 Refactoring in OCL

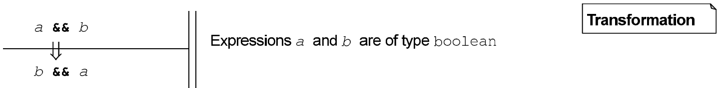
Because OCL has no side effects and is determined, there is a wide range of transformations available that we can apply to OCL expressions. These include examples from Section 9.1 or rules for handling containers, such as:

Transformation

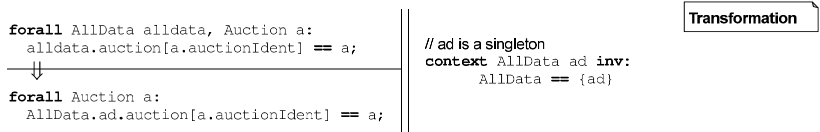
```
a.add(x).size
    ⇓
a.size+1
```

1. Expression *a* is of type `List(X)`
2. Expression *x* is of type `X`
3. Expression *x* has a defined value and has no side effects

In addition to the algebraic transformations, it is primarily the rules of logic that are important for modifying OCL expressions. Typical rules are those of Boolean logic such as commutativity:



Because OCL is embedded in a UML context, many expressions can only be formulated using elements from the underlying models. Let us assume that there is a specification stating that there is exactly one object of the class `AllData` and that this class is accessible with `AllData.ad`. In this context the following transformation is possible:



As already discussed in Section 9.3.6, mathematics has a long tradition in transforming expressions correctly. These transformation techniques have been made more precise and further refined by means of logic calculi and algebraic systems. Today, there are a number of tools that allow a precise manipulation of formulas. These include, for example, the theorem prover based on HOL [NPW02] or the KIV system [Rei99]. Embedding OCL in HOL, as discussed in [BW02a] and [BW02b], allows us to transform OCL expressions into HOL and to apply the verification apparatus available there to OCL.

The refactoring rules on OCL indeed form a logic calculus for OCL. The precision of the context conditions, which is common for a logic, is very helpful if the application of the rules is to be supported with automated tools. Context conditions on the syntax that can be tested automatically can be adopted accordingly by a tool. For the context conditions that cannot be tested automatically, we can use various strategies:

- The context conditions are *checked informally for plausibility*. This does not ensure that the transformation is correct. However, the conditions are generally invariants which can be used in tests. This means there is a possibility that an invariant which has been transformed incorrectly will be recognized by a defective test. The probability of this occurring, however, is relatively low, as there is generally no "coverage" of the different alternatives of an invariant by tests. Nevertheless, this pragmatic approach is sufficient for various types of projects.
- The situation is significantly improved when we use additional tests to check that a context condition is correct. This means that the context condition of a transformation formulated in OCL is itself seen as an invariant and temporarily inserted in the code during some of the transformation steps. It is easier to check the plausibility of context conditions if a good test suite is available or further tests are defined. As the tests themselves run automatically and can therefore be used efficiently, the additional effort for the approach is rather low and should be used at least for context conditions which are critical or not entirely clear.
- Verification of the context conditions offers security in terms of the correctness of the transformation but usually cannot be executed or requires pretty much effort.

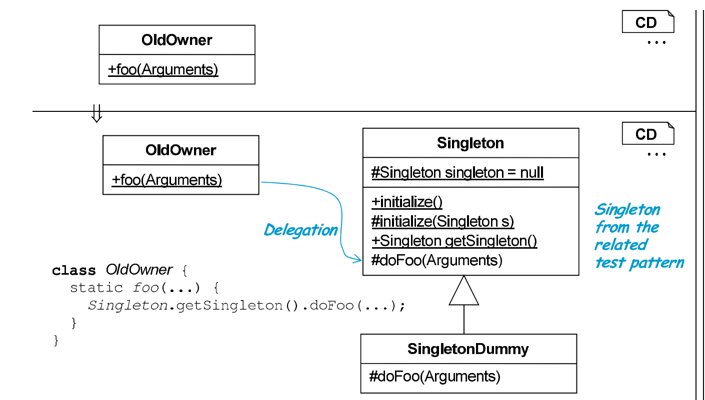
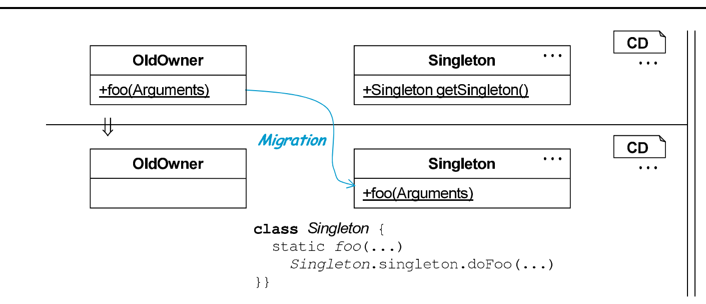
Section 10.2 discusses and expands on the proposed procedure of checking context conditions by means of tests and demonstrates this by changing the data structure.

OCL is designed as a specification language in the context of other UML diagrams. This is another reason why OCL offers little support for verification techniques. In practice, therefore, the first two approaches are preferred.

10.1.5 Introducing Test Patterns as Refactoring

Chapter 8 describes several refactoring patterns that increase possibilities for defining tests. These patterns were presented mainly by describing the resulting structure. However, the system under test often exists in a different suboptimal form and we have to adapt it accordingly in order to be able to define tests effectively. Therefore, we have to transform the system with refactoring techniques to inject the structure proposed by the test pattern.

For example, the following refactoring rule introduces the system structure discussed in Table 8.9. It allows the test environment to adapt a static method by encapsulating it in a singleton.

Refactoring: Making static methods adaptable for tests	
Problem	Static methods are inaccessible for tests as they cannot be overwritten with dummies.
Goal	The goal is to delegate the functionality of a static method to an instantiated object which, for tests, can be replaced by a dummy, while avoiding a publicly accessible static variable for this object.
Motivation	See Table 8.9 for a description of the test pattern.
Refactoring part 1	 • The class `Singleton` is introduced as described in the diagram. • The initialization of the class `Singleton` is ensured. • *doFoo* contains the same functionality as `foo` from `OldOwner`. Where applicable, note the attributes used (see also refactoring for migrating a method). • `OldOwner.foo` now delegates to `doFoo`.
Refactoring part 2	

(continued on the next page)

(continues Table 10.8.: Refactoring: Making static methods adaptable for tests)

	• To encapsulate the singleton, static method `foo` is migrated into the singleton. • Before deleting the method in `OldOwner`, we have to adapt all calls to new method `foo`. • We can now also delete the method `getSingleton` and the singleton object is no longer publicly accessible.
Implementations	The implementations of the individual methods can be found in Table 8.9.
Examples	This transformation was used in the auction system to encapsulate the database connection and the logging, for example (see also Fig. 8.8).
Noteworthy	The rule was divided into two parts as the first part can be used independently. This is recommended if the singleton is to remain publicly accessible. If the class `OldOwner` has no further tasks, we can use it directly instead of the new `Singleton` class introduced. The encapsulation of static variables and the definition of methods for accessing and manipulating these variables are related to this refactoring rule.

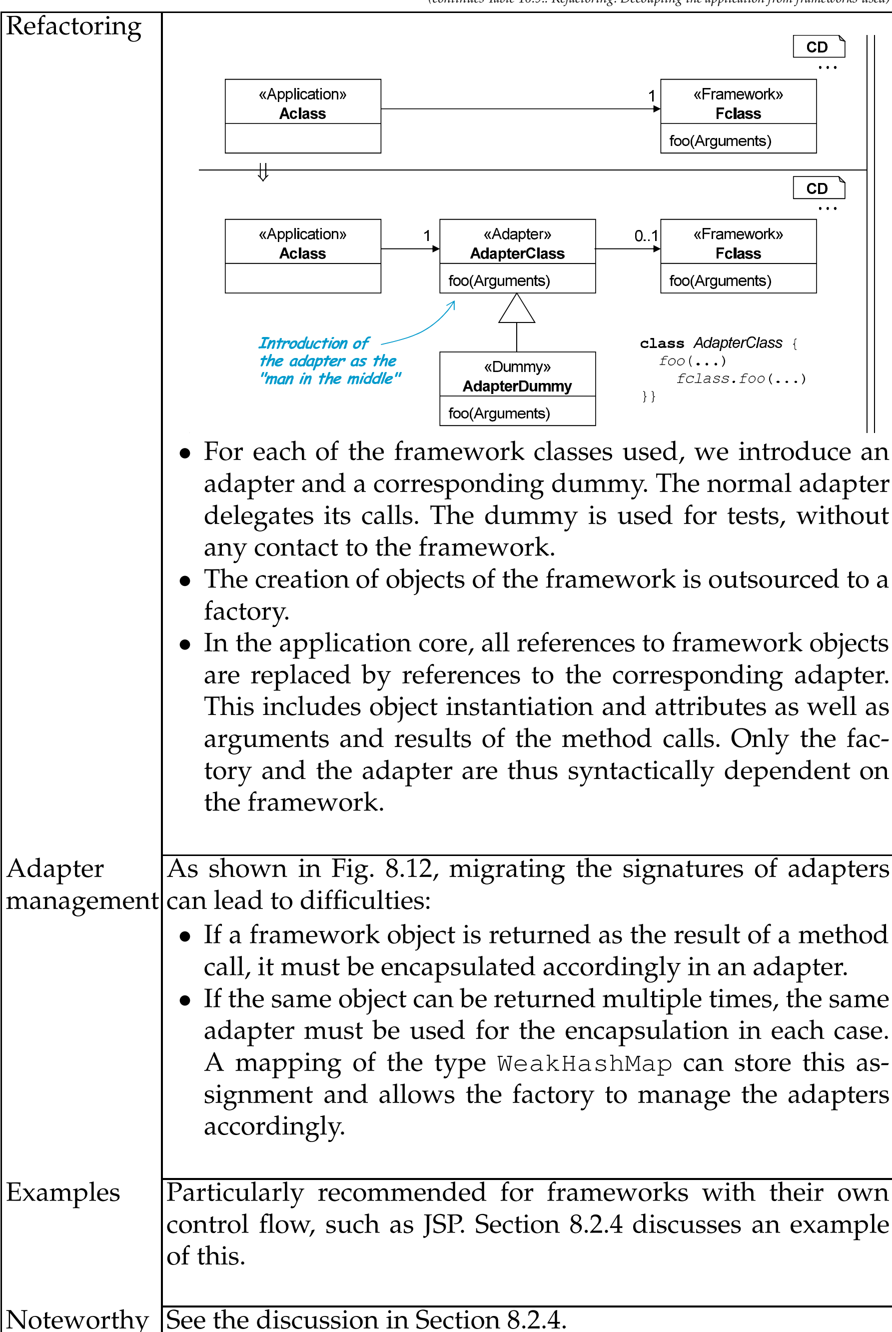
Table 10.8. *Refactoring:* Making static methods adaptable for tests

The other test patterns defined in Chapter 8 can be represented similarly to the refactoring rule above. This is demonstrated, for example, for the rule in Table 10.9 that decouples the application of frameworks discussed in Section 8.2.4.

Refactoring: Decoupling the application from frameworks used	
Problem, goal, and motivation	As described in Section 8.2.4, a framework usually cannot be adapted, not even for tests. To improve testability of the application testable, we separate the application and the framework that it uses by an adapter layer. Further advantages of this technique are described in [SD00].

(continued on the next page)

(continues Table 10.9.: Refactoring: Decoupling the application from frameworks used)



Refactoring	• For each of the framework classes used, we introduce an adapter and a corresponding dummy. The normal adapter delegates its calls. The dummy is used for tests, without any contact to the framework. • The creation of objects of the framework is outsourced to a factory. • In the application core, all references to framework objects are replaced by references to the corresponding adapter. This includes object instantiation and attributes as well as arguments and results of the method calls. Only the factory and the adapter are thus syntactically dependent on the framework.
Adapter management	As shown in Fig. 8.12, migrating the signatures of adapters can lead to difficulties: • If a framework object is returned as the result of a method call, it must be encapsulated accordingly in an adapter. • If the same object can be returned multiple times, the same adapter must be used for the encapsulation in each case. A mapping of the type `WeakHashMap` can store this assignment and allows the factory to manage the adapters accordingly.
Examples	Particularly recommended for frameworks with their own control flow, such as JSP. Section 8.2.4 discusses an example of this.
Noteworthy	See the discussion in Section 8.2.4.

Table 10.9. *Refactoring:* Decoupling the application from frameworks used

10.2 A Superimposition Method for Changing Data Structures

Refactoring steps are relatively small and systematic. This is to ensure that the application of the rule is manageable and any errors that occur can be recognized and eliminated efficiently. In this section, we will discuss a technique that simplifies the management of complex refactoring steps without breaking them down into many individual steps. This technique is particularly suitable if we want to change data structures modeled with class diagrams. It is based on the idea—developed in the auction project and applied successfully in several cases—of using the old and new data structure in parallel during the transformation and placing them in a relationship with one another using suitable invariants. Because the first step in this approach is the addition of the new data structure without removing the old one, the approach is referred to as *superimposition*. The sections below first present the approach and then demonstrate it and discuss it in detail using two examples.

10.2.1 Approach for Changing the Data Structure

Refactoring steps are suitable for changing a data structure while ensuring the correctness of the modification with tests as far as possible. In more formal approaches, such as [BBB+85], ensuring the context conditions for this transformation with verification techniques is well-understood. Based on the concepts developed in [BBB+85], Table 10.10 proposes a pragmatic approach for changing a data structure. Here, the use of invariants to define the relationships between the old and the new data structure is a significant element for a correct transformation.

Changing the data structure using superimposition	
Problem, goal, and motivation	It is not always easy to break a change of data structure down into a number of small refactorings, although the larger number of steps bears the increased risk of introducing an error. The goal is to use invariants that relate the old and the new data structure to thus control larger refactoring steps and to obtain confidence in their correct application.

(continued on the next page)

(continues Table 10.10.: Changing the data structure using superimposition)

Approach	**Changing a data structure with refactoring techniques** consists of the following steps: 1. Identification of the old data structure to be replaced. 2. Development of the new data structure, the related methods, and required tests which are added to the existing, old data structure. The existing system is not changed; its functionality is retained. This is checked by means of tests. 3. Definition of invariants that place both data structures in a relationship. 4. At all points at which the old data structure is *changed* or *values are assigned* to it, the new data structure is now also modified or gets values assigned. After each of these modification points, the corresponding invariants are inserted in order to check them. This integrates the new data structure into the system executions without involving it in the behavior of the system. This is checked by the tests. 5. All points that *use* the old data structure are now converted to the new data structure. This is again checked by the tests. 6. In most cases, the modification means that some program parts can be transformed and thus simplified algebraically. The result is checked by the tests. 7. At the end, the old data structure that is no longer used is removed. The system runs as usual. This is checked by the tests.
Examples	Sections 10.2.2 and 10.2.3 demonstrate the approach using two examples from the auction system.
Noteworthy	• Steps 5 and 6 are usually performed together. • If individual steps are complex, it is advisable to perform additional intermediate tests. • Tests are often affected by the changeovers and can also be transformed in step 5. Tests can also become obsolete, or it may be necessary to define additional tests.

Table 10.10. Changing the data structure using superimposition

In contrast to a verification approach, the trick in this approach is to use the invariants in tests. Assuming that there is a sufficiently good test suite for the system, we can ensure that the transformation is correct with a high degree of probability. As the tests required for checking the change of the data structure in the approach proposed in this book already exist, the change of data structure can be performed efficiently. Actually, this principle was used

at several occasions in the auction project with extraordinary success; the effort required for more complex changes to the data structure was much lower than estimated because the approach was systematic, there were few errors, and these errors were identified, localized, and eliminated very quickly.

If there is still insufficient confidence in the correctness of the transformation after the testing activity as discussed in Section 10.1.4, an additional verification is possible, for example, based on the Hoare logic.

The approach outlined in Table 10.10 for changing a data structure has to be suitably adapted based on the actual complexity and form of the data structure. If, for example, elements of the old data structure are used as method parameters, we have to add the new data structure in parallel by extending the method parameters (step 2). This allows us to use the preconditions of such methods to check that the two data structures match (step 3). In a further refactoring step at the end of the process, the method parameters of the old data structure that are no longer required are removed (step 7).

In addition to the removal of the old data structure in step 7, the advantages of the new data structure usually become evident in the simplification of the programs in step 6. Of course, steps 5 and 6 can also be performed together. In order to be able to describe the relationships between both data structures effectively, it can be useful to temporarily use additional methods for transformation between the data structures and also remove these at the end.

Although we can use this approach for almost all refactoring rules, such as moving an attribute or splitting a class, the technique is primarily suitable for larger refactorings. The section below uses two examples from the auction project to show how we can apply this approach.

10.2.2 Example: Representing a Bag of Money

In a first version of the auction system, money was represented by a number of the data type `long`. For the purposes of internationalization, the system had to be converted to explicit `Money` objects because different currencies should be processed within an auction. The following steps change data structures in a simplified way.[1]

Step 1: As shown in Fig. 10.11, the initial data structure is identified.

Step 2: The new data structure is shown in Fig. 10.12. Furthermore, suitable tests are developed for the new class `Money`. These tests test the functions offered by the class to a sufficient level.

Step 3: Invariants between the two data structures are easy to identify. Using the query `valueInCent`, the following can be defined as an invariant:

```
context Auction a inv BestBidEqualsCurrentBid:                OCL
  currentBidInCent == bestBid.valueInCent()
```

[1] The auction system actually considers different currencies, quantity-based prices such as “`Euro/kg`”, decimal places, and slots of varying sizes.

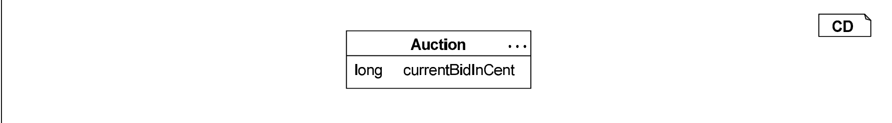


Fig. 10.11. Original data structure for representing sums of money

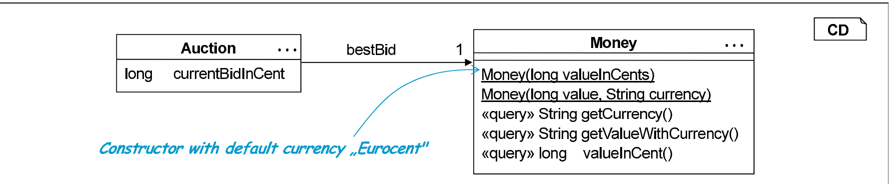


Fig. 10.12. Extended data structure for representing bid values

According to the two-valued logic for OCL used in UML/P and the handling of undefined values, the invariant is satisfied precisely when association `bestBid` is a link to a `Money` object that has the appropriate content.

Step 4: The new data structure is introduced but is not used yet. In this step, therefore, it is modified or values are assigned to it at all points where the same is done to the old data structure. Fig. 10.13 shows an extract from the method that accepts a bid and calculates the new best bid.

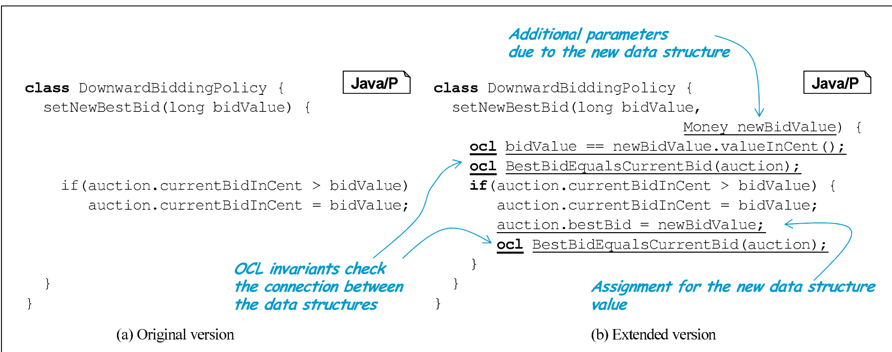


Fig. 10.13. Assignment of values to the new data structure

Because the method shown receives the current bid as an argument, a second argument is introduced for the new representation of the bid. The first OCL constraint is derived from the invariant `BestBidEqualsCurrentBid` and ensures that the arguments are correct. The other two OCL constraints test the invariant at the beginning of the method and after the money object has been changed.

Step 5: Values are now assigned to the new data structure but the old data structure is still in use. Therefore, all program elements that use the old data structure are now replaced. In many cases, the invariants can be used to do this. In this case, the invariant `BestBidEqualsCurrentBid` formulated in step 3 can be understood directly as a replacement instruction. The left-hand side of the equation

```
currentBidInCent == bestBid.valueInCent()                OCL
```

can be replaced by the right-hand side of the equation at all places that use the equation. Fig. 10.14 shows this for the result of step 4 (Fig. 10.13).

```
class DownwardBiddingPolicy {                                         Java/P
  setNewBestBid(long bidValue, Money newBidValue) {
    ocl bidValue == newBidValue.valueInCent();
    ocl BestBidEqualsCurrentBid(auction);
    if(auction.bestBid.valueInCent() > newBidValue.valueInCent()) {
       auction.currentBidInCent = bidValue;
       auction.bestBid = newBidValue;
       ocl BestBidEqualsCurrentBid(auction);
    }                                      Use of the new data structure
}}
```

Fig. 10.14. Using the new data structure in Java code

This transformation is also useful for the OCL constraints. For example, the specification of the method `setNewBestBid` in accordance with Fig. 10.15(a) can in a first step be expanded to the form shown in (b) and then transformed into the version in (c). In these transformations, however, we must ensure that the invariants used to represent the relationship between the old and the new data structure are not coincidentally replaced as well.

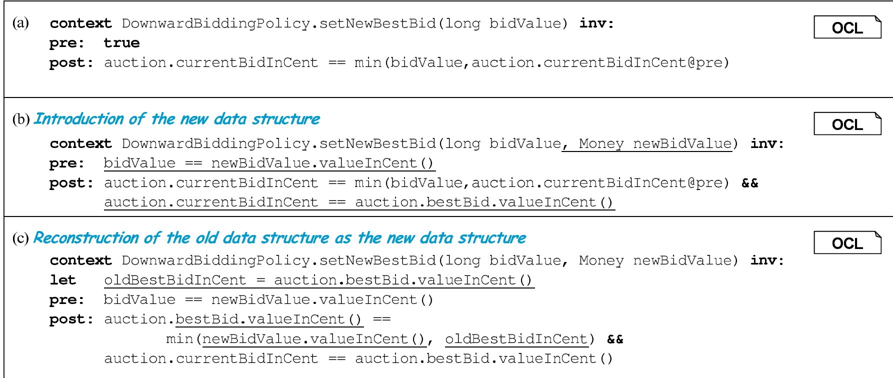


Fig. 10.15. Using the new data structure in OCL constraints

This example also shows that the transformation cannot always be executed fully automatic. In our example, an attribute is replaced by a method call and the operator `@pre` can no longer be applied, for example. Therefore, the corresponding value is buffered in a **let** variable.

Step 6: The simplification of the resulting pieces of code and in particular, of the expressions, is a significant step towards keeping the code created readable and elegant. Steps 5 and 6 are often performed together. For large changes of data structure, however, step 6 can be split up into several small steps. The following replacement applies, for example:

Transformation

```
" "+bestBid.valueInCent()+" Eurocent"
⇓
bestBid.getValueWithCurrency()
```

Calculations can also be simplified, as shown in the following transformation, provided that the old `bid1` object is not known at another point:

Transformation

```
bid1 = new Money(bid1.valueInCent()
                 +bid2.valueInCent());
⇓
bid1.add(bid2);
```

What is important here, just like for all the other steps, is that the automated tests are executed after each step. If there are insufficient tests available, we have to develop additional tests.

Step 7: In the last step, we can now remove the old data structure along with all invariants and conditions that place the old and the new data structures in a relationship. The result is shown in Fig. 10.16.

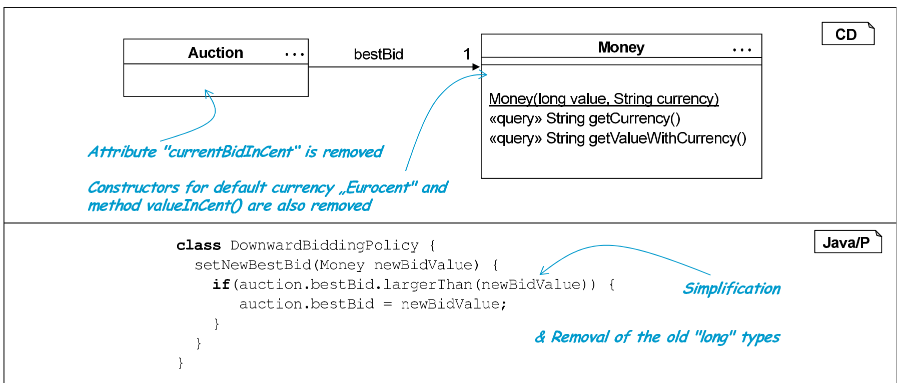


Fig. 10.16. Result of changing the data structure

For the simple example shown, the method used is relatively complex and time-consuming. The very detailed method proposed here is only rec-

ommended if the change is more complex and thus more prone to errors. However, the data structures involved do not necessarily have to be complex. It is also helpful to apply this technique if a lot of attributes of the type `long` are to be replaced by `Money` objects and the complexity thus arises from the number of elements to be replaced.

10.2.3 Example: Introducing the Chair in the Auction System

The superimposition method was used in the auction system at this level of detail for the first time when the following requirements appeared: (1) A bidder can take part in multiple auctions simultaneously, and (2) a colleague who is observing an auction does not have to be employed by the same company as the bidder.

In fact, requirement (1) was known from the very beginning but was not implemented immediately. This was because auctions are by nature of short duration and a situation with multiple parallel auctions for the same bidder was initially improbable. This changed as auctions of similar goods synchronized in terms of time were desired in order to boost competition.

Step 1: Identification of the Old Data Structure

From the very beginning, the auction system was designed to offer customers further roles for observation (described in Appendix D, Volume 1) in addition to the active bidders and the auctioneer. Multiple variants of external observers were permitted. Bidder colleagues receive the same information as the actual bidder but cannot submit bids. The recognition of colleagues was realized via a common `Company` object. Requirement (2) originates from the finding that large companies have different locations and subsidiaries, engage external consultants as bidders, etc., and therefore flexibilization was required.

The example described was implemented very efficiently and without any errors using the approach outlined in this section. This is even more surprising as, due to the central importance of the changed system structure, not only the application core but also the database, the system for setting up auctions, and the graphical interface right up to the password-protected login procedure based on employment with the company had to be adapted. A secondary factor was also that a number of unit and acceptance tests had to be changed.

Fig. 10.17 shows a simplified excerpt of the initial situation for changing the application core.

The last OCL constraint `SameInfos` shows that a bidder's current own bid, the symbol used for representation etc. are identical for persons of the same company.

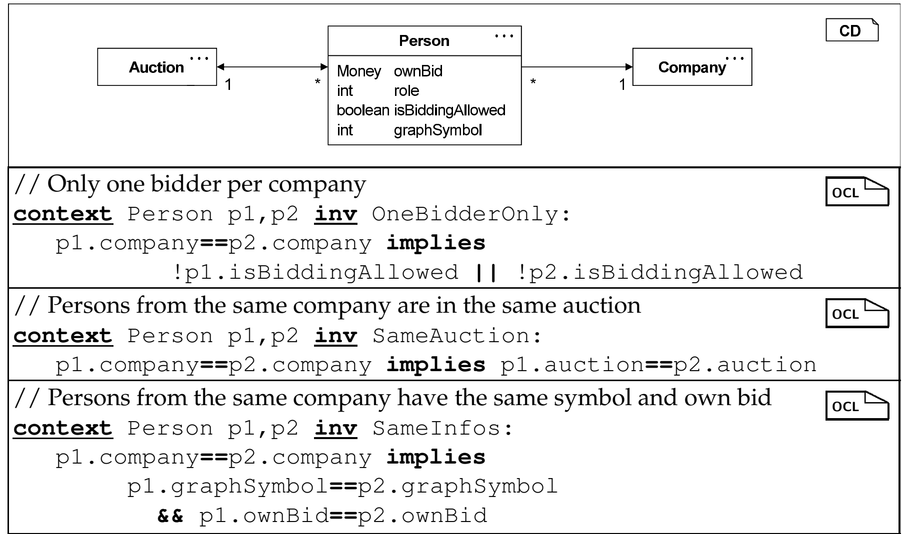


Fig. 10.17. Initial situation with invariants

Step 2: Development of the New Data Structure

The situation in Fig. 10.18 was identified as the desired data structure. In this situation, the roles are defined by subclasses rather than a flag. Furthermore, the abstraction `Chair` was introduced as a metaphor for the chair of a person in a classic auction.

The original conditions `OneBidderOnly` and `SameInfos` no longer apply. Invariant `SameAuction` becomes `ChairSameAuction`. The new invariants `ChairAssoc1` and `ChairAssoc2` were introduced. These invariants demonstrate the role of the class `Chair` with respects to the association between `Person` and `Auction`. Together with the introduction of the `Chair` classes, a number of new tests were developed for the new data structure but these are not represented here.

Step 3: Definition of the Invariants

The two class diagrams in Fig. 10.17 and 10.18 show only parts of the implementation, but overlap in some classes and an association. The class diagrams now together describe the implementation. For the code generation the two diagrams are merged, as described in Section 2.4, Volume 1. Building on this, we can now identify the required invariants between the old and the new data structure. Initially, we continue to assume that persons take part in only one auction, as the tests are designed for the old data structure:

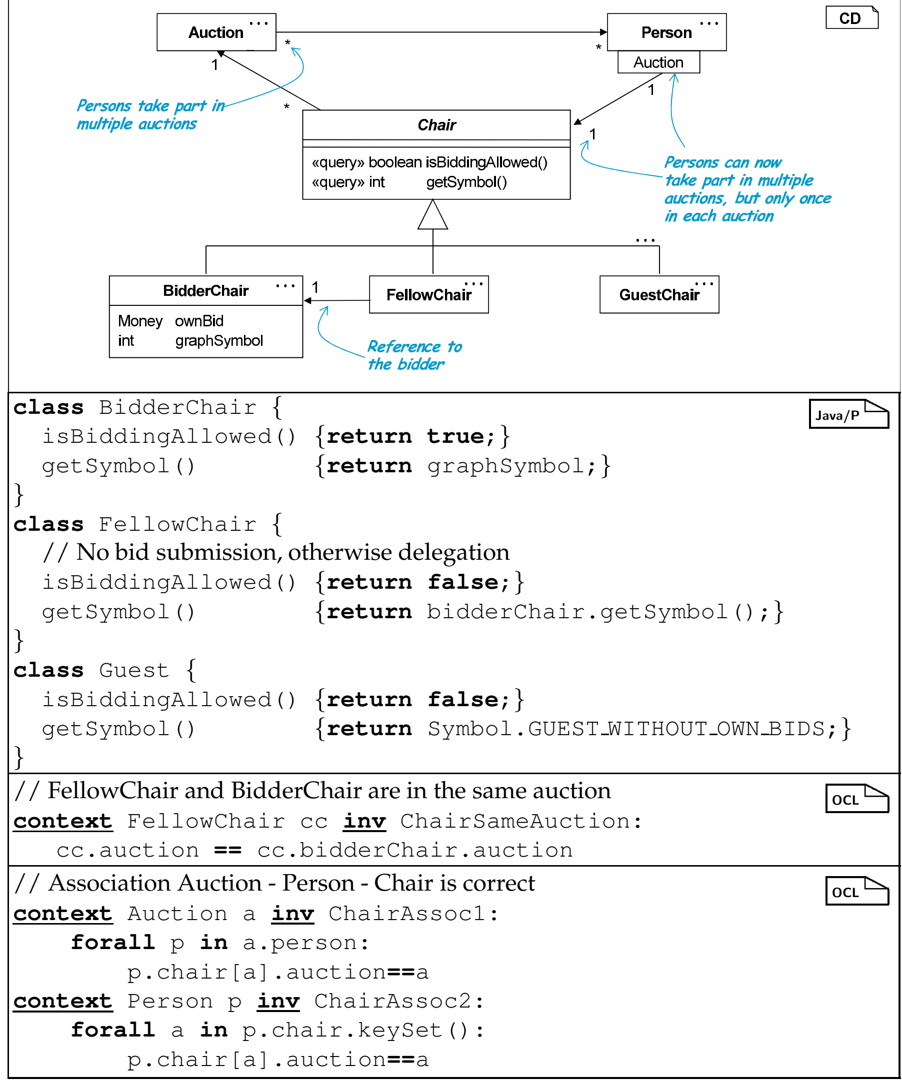


Fig. 10.18. Target structure with invariants

```
context Person p inv:                                    OCL
  // Initially only one chair for each person
  p.chair.size==1;
```

Accordingly, any `p.chair` is the unique `Chair` object reached from `Person p`. This allows us to identify some invariants that affect the transfer of information from the `Person` object to the `Chair` object.

```
context Person p inv PersonChairInvs:                          OCL
  let Chair c = any p.chair in
    // Bidder has BidderChair
    ( p.role==IS_SUPPLIER && p.isBiddingAllowed <=>
                                  c instanceof BidderChair ) &&

    // Bidder colleague has FellowChair
    ( p.role==IS_SUPPLIER && !p.isBiddingAllowed <=>
                                  c instanceof FellowChair ) &&

    // isBiddingAllowed is the same
    p.isBiddingAllowed == c.isBiddingAllowed() &&

    // Symbol is the same
    p.graphSymbol == c.getSymbol()
```

The following properties also apply but are represented separately so that they can get individual names:

```
context Person p inv BidderChairInv:                           OCL
  let Chair c = any p.chair in
    // Bid is the same as for bidder
    typeif c instanceof BidderChair
             then p.ownBid == c.ownBid
             else true
```

```
context Person p inv FellowChairInv:                           OCL
  let Chair c = any p.chair in
    // Bid is the same as for bidder colleague
    typeif c instanceof FellowChair
             then p.ownBid == c.bidderChair.ownBid
             else true
```

The connection between the bidder colleague and the related bidder is organized via a link. If person `p1` is allowed to bid and person `p2` represents a bidder colleague from the same `Company`, the link must be set accordingly:

```
context Person p1, Person p2 inv:                              OCL
  let BidderChair   c1 = (BidderChair) any p1.chair;
      FellowChair   c2 = (FellowChair) any p2.chair in
  defined(c1) && defined(c2) && p1.company==p2.company
    implies c2.bidderChair==c1
```

The new data structure is complex enough to almost certainly introduce errors during the development of the new data structure and their invariants. At least, we can check the models permanently with our syntax checks and automated tests. Due to the redundancy that we introduce we have the following possibilities to identify errors:

1. In the automated tests
2. In the old data structure (which is assumed to be initially correct)
3. In the new data structure
4. Via invariants that connect the old and new data structures

Step 4: Assignment of Values to the New Data Structure

In the next step, the code for assigning values to the new data structure is integrated at all necessary points. In doing so, these invariants are used to check that the new code is correct. As the invariants already exist, we can use them as guiding specifications which describe how the implementation has to be adapted. For example, the implementation for `getSymbol()` can be extracted from `p.graphSymbol==c.getSymbol()`. This means that the considerations for the definition of the invariants are reused and this increases the efficiency of the development. However, if the invariants are used to derive the implementation, defective invariants are not recognized and in contrast, result in a defective implementation. Therefore, we have to decide for each case individually whether to define the implementation independently of the invariants because there are tests for the old data structure that will subsequently be converted to the new data structure and then check that the new data structure is correct.

The assignment of values to the new data structure is demonstrated in Fig. 10.19 by an example based on the method for storing a bid for a person. The form of the bids used here is described in Appendix D, Volume 1.

Steps 5 and 6: Integration of the New Data Structure and Optimization

In combination with step 6, step 5 allows us to modify and optimize the system in stages. A conservative approach is to first continue to provide all previous methods to thus allow the existing interfaces to remain available to the environment of the modified data structure. However, it should also be checked where optimizations—for example, through the expansion of methods—are applicable.

The conservative approach can be demonstrated with the simple example of the `get` and `set` methods. Sections 4.2.2 and 5.1 describe how to generate these `get/set` methods from an attribute of the class diagram. If the attribute is moved, the related methods are no longer generated. However, if these methods have been used in some places, a manual definition of the methods can be made available. For example, the following is a suitable replacement that is based on the new data structure:

Java/P

```
class Person {
  Money getOwnBid(Auction a) {
    return this.chair.get(a).getOwnBid();
}}
```

```
class Auction {                                                      Java/P
  handleBid(Bid bid) {
    // ...
    BidMessage bm = new BidMessage(bid.auction, ...)
    // Save operation for all persons
    for(Iterator(Person) ip = person.iterator(); ip.hasNext(); ) {
      Person p = ip.next();
      p.receiveBid(bm);
    }
    ocl forall p in this.person: FellowChairInv(p);
  }
}
class Person {
  receiveBid(BidMessage bm) {
    // Own bid or bid by bidder colleague?
    if(bm.person.company == this.company)
      ownBid = bm.money;
    // Own bid?
    if(bm.person == this) {
      ((BidderChair)this.chair.get(bm.auction)).storeOwnBid(bm);
      ocl BidderChairInv(p);
    }
    // ... send message to client Applet
  }
}
class BidderChair {
  storeOwnBid(BidMessage bm) {
    ownBid = bm.money;
  }
}
```


Fig. 10.19. Assignment of values to the new structure added

We can also use the explicit definition of this method to overwrite the method that is otherwise generated as standard. Therefore, we can use such explicit definitions of `get/set` methods easily and elegantly to divert the access to attributes in the old data structure to the new data structure. This conservative implementation is suitable for checking that the transformation is correct with the existing tests.

```
class Person {                                                       Java/P
  receiveBid(BidMessage bm) {
    if(bm.person==this) {
      ((BidderChair)chair.get(bm.auction)).storeOwnBid(bm);
    }
    // ... send message to client Applet
  }
}
```

Fig. 10.20. Simplification and removal of the old data structure

Step 7: Removal of the Old Data Structure

The old data structure—along with all unnecessary invariants—is now removed. Fig. 10.20 shows the result for the method `receiveBid`.

Adapting the Tests to the New Data Structure

It is quite common that after a refactoring step some tests are no longer correct. The transformation of a test sometimes fails because the called methods or observed attributes are no longer present. In step 5, we can use the duplicate representation of the data structures that exists after step 4 to migrate the existing tests. However, tests that are programmed against abstract interfaces may require only simple transformations or no transformation at all. In contrast to the product code/model, it is usually not necessary to optimize tests in step 6. It is sufficient to keep the tests executable and meaningful. Superfluous tests can be removed.

A test consists of different types of UML diagrams. Object diagrams are used, for example, to represent the set of test data and the expected result. As a consequence of the superimposition approach, we also have to extend object diagrams. The object diagram in Fig. 10.21, for example, shows an excerpt of a set of test data to which the new data structures have already been added.

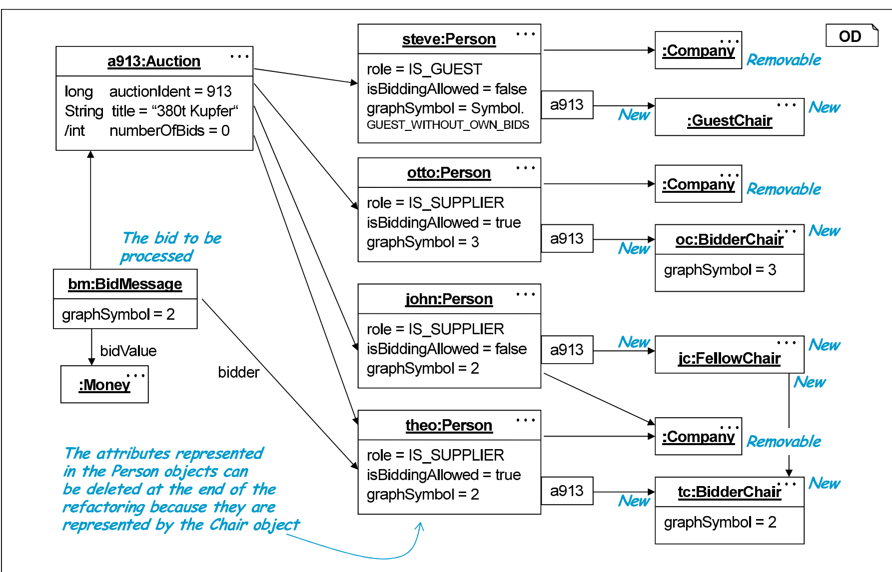


Fig. 10.21. Modified object structure as a set of test data

Object diagrams used constructively must contain a complete representation of the test data and are therefore almost always affected by a change of the data structure. In contrast, in an additive approach, object diagrams used as predicates are relatively stable because they are often not affected by the new part added.

We can modify sequence diagrams systematically in an additive approach in similar fashion. We add the new interactions to a sequence diagram provided they are to be observed by the test described. If a relatively free interpretation of the observation was selected in the sequence diagram, for example, by using the stereotype ≪match:free≫, we do not have to add these new interactions to the diagram and we can continue to use the diagram unchanged. Fig. 10.22 shows the observation of the interaction of the auction with the persons involved when distributing the message for the new bid. This observation takes the interaction with the new `Chair` objects into account.

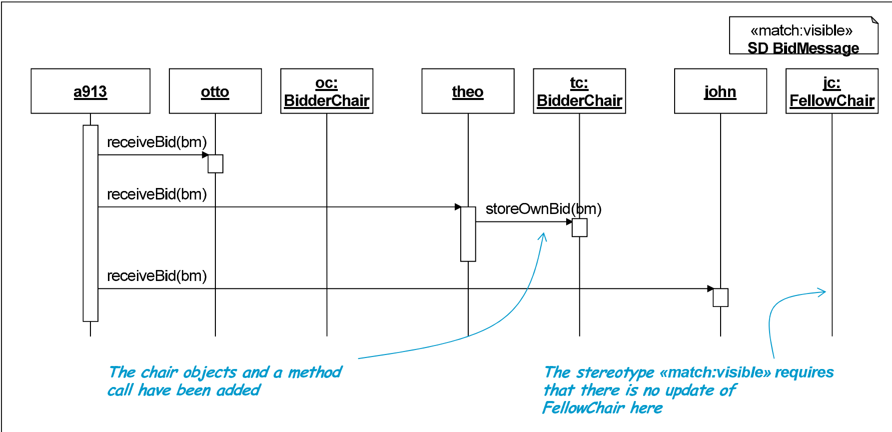


Fig. 10.22. Sequence diagram checks the modified system run

Summary of the Refactoring with Superimposition

In summary, we can state the following for the examples demonstrated here as an excerpt and the underlying superimposition method for performing refactoring:

- The confidence that the refactoring is correct is partly established through the existing tests based on the old structure and transferred to the new structure. However, as a result of the invariants used, the old and new data structures are placed in a relationship and this further increases the confidence that the transformation is correct.
- The additional effort involved in developing these invariants and integrating them temporarily is offset by the advantage that, as a result of this approach, larger transformations can be executed as one unit. Therefore, we do not have to break the change of data structure performed in

the last example down into a number of smaller refactoring steps. Alternatively, we would have to execute individual steps to first introduce the class `Chair`, migrate the individual attributes, replace the different `Chair` variants represented by flags with subclasses, and, finally, establish the relationship between `FellowChair` and `BidderChair` before being able to remove the `Company` class at the end.

- As illustrated by the examples in Fig. 10.21 and 10.22, this approach also supports the migration of tests by allowing a separation between the addition of the new and the removal of the old data elements and interactions in two steps.

Overall, when changes of data structure are needed, the superimposition method qualifies as an effective alternative or supplement to the refactoring rules described in [Fow99].

This allows us to create additional, more general refactoring rules for UML/P, which we can reuse in other, similar situations. The principle is similar to that for creating frameworks and design patterns. Reusable parts are extracted from a special application and generalized to create a general rule. Special cases and alternative situations can be recognized when the rule is used for other applications and then integrated in the rule.

10.3 Summary of Refactoring Techniques

Chapter 9 and this chapter looked at the foundational principles for transformational software development based on UML/P. The chapters also combined these principles with the methodology for performing refactoring steps. Instead of listing individual rules, the chapters discussed how we can use existing sets of rules—such as those for Java—to transfer refactoring rules for UML/P. A pragmatic extension, based primarily on invariants, was given with the description of a *superimposition approach* for changing data structures. The practical usability of this approach was demonstrated with examples. We have seen that, together with the approaches for defining tests discussed in Chapters 6, 7, and 8, UML/P is an excellent language for the evolutionary development of the product system and the test cases.

Refactoring is a technique that has become increasingly popular since [Fow99]. Its roots in evolutionary development were documented in [Opd92]. In fact, the idea of a transformational software development which is performed similarly to the mathematical derivation of expressions goes back much further [BBB+85, Dij76, PR03]. It was only through [Fow99] that this approach for transforming existing systems, which was initially presented in a very formal way, was made accessible for wider usage through practical descriptions based on the design patterns from [GHJV94]. One of the main success factors was the replacement of the verification approaches for ensuring the correctness of a transformation by means of *automated tests*. Although

these tests do not ensure that the transformation is correct, practical experience shows that they offer good protection against introducing errors during the transformation. The use of automated tests to define the required *notion of observation* in a refactoring is a significant accomplishment that results from this.

In the last years new refactoring rules have been developed and it is not clear how many rules a portfolio should provide for practical purposes. We do know general principles—such as "divide et impera" or the "generalisation" of methods often through extending parameters—from which rules can be derived. [TDDN00], for example, discusses the language independence of refactoring rules by extracting common features between Java and Smalltalk rules. There are also language-specific refactoring rules, such as the handling of exceptions in Java or Statecharts in UML/P. A third class of refactoring techniques results from the handling of framework-specific or component-specific situations. In Java, for example, a `Vector` can be replaced by a `Set` structure if the order and number of the elements contained therein are irrelevant.

Refactoring techniques will enjoy permanent success above all when tool support continues to mature. Development environments already allow efficient identification and handling of all points where a method or an attribute occurs. However, concrete support for refactoring steps up to the proposal of algebraic simplifications will still require a lot of effort from tool manufacturers in the future.

The set of rules specifically for Java is currently being steadily extended and questions are arising regarding the transfer of rules between languages and a better foundation for the correctness of refactoring rules. The rules must therefore be made more precise so that they can be implemented automatically as tactics and are understood in all implications (context conditions). While the refactoring rules in [Fow99] are very helpful for manual application, their context conditions, special cases, etc. are currently too informal to be accessible for a formal examination of correctness.

11

Summary, Further Reading and Outlook

The essence of knowledge is,
having it, to apply it.
Confucius

Finally, we give a summary and an in detail outlook on further readings and publications in the context of the UML/P.

11.1 Summary 324
11.2 Outlook 325
11.3 Agile Model Based Software Engineering 328
11.4 Generative Software Engineering 331
11.5 Unified Modeling Language (UML) 332
11.6 Domain Specific Languages (DSLs) 332
11.7 Software Language Engineering (SLE) 335
11.8 Modeling Software Architecture and the MontiArc Tool 339
11.9 Variability and Software Product Lines (SPL) 342
11.10 Semantics of Modeling Languages 344
11.11 Compositionality and Modularity of Models and Languages 348
11.12 Evolution and Transformation of Models 349
11.13 State Based Modeling (Automata) 351
11.14 Modelling Cyber-Physical Systems (CPS) 354
11.15 Applications in Cloud Computing and Data-Intensive Systems 355
11.16 Modelling for Energy Management 356
11.17 Modelling Robotics 358
11.18 Automotive Software 359
11.19 Autonomic Driving and Driver Intelligence 360

© Springer International Publishing AG 2017

B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_11

While the UML has become more stable in its evolution it became clear that there are many more forms of models relevant for software development. Models used in the various domains of software and systems development share similarities, but also need different domain-specific adaptations. Therefore, this final chapter we do not only summarize the current state of UML/P and give an outlook on the modeling future, but in particular give hints to further readings and publications that report on modeling techniques, modelling languages, semantics and related topics that were mostly built on the UML/P and the results gained from its development. Because the author developed the UML/P already at the TUM—Munich University of Technology—and then started a group specifically for modelling issues in 2003 at the TU Braunschweig and moved in 2009 to the RWTH Aachen University these further reading suggestions starting with Section 11.3 are mainly publications from this group.

11.1 Summary

Volume 1 [Rum16] and this book, Volume 2 [Rum17], introduced a number of model-based concepts and techniques for the rapidly evolving *Software Engineering portfolio*. Based on practical experience and well-founded analytical concepts, these books form a double bridge. On the one hand, they use UML to develop theoretical approaches for practice and thus increase industrial applicability of these approaches. On the other hand, the books apply concepts from agile methods to the UML.

Until now, UML has been used primarily in plan-based methods for defining milestones and development phases. Using the concepts discussed in this book, we can combine the values system, principles and development practices of agile methods with UML.

The main product of Volume 1 was the definition of a precise language profile of UML/P that we can use for many types of applications. This language profile (1) ignores less important concepts, (2) offers a precise explanation of the meaning of all used constructs, and (3), thanks to additional concepts, is tailored to be used as a programming language, modeling language, and a language for defining test cases.

The main technical concepts of agile methods that are directly affected by the language used are code generation, the ability to define automated tests, the testability of the code generated, and the evolution of the models that are necessary because of changing requirements or a software architecture that has potential for improvement.

Therefore these techniques were applied to UML/P in both books. We also discussed how to use UML/P models to define automated tests and how to apply refactoring techniques to UML/P models. The books provided a number of test patterns specifically designed to improve the testability of

object-oriented software and for testing the functions of distributed and concurrent systems. These tests allowed us to define a precise notion of the concept "observation" that serves as a basis for our refactoring steps.

The UML/P language profile presented in this book and the techniques based on this language profile form the basis for efficient development. They enable us to optimize the flexibility, efficiency, and costs of the process, the quality and maintainability of the product, time-to-market, and, ultimately, customer satisfaction. They do so by enabling us to select the process with its associated development-appropriate practices. A further optimizing contribution is the tailoring of our notations to agile development approaches.

The techniques presented for generating code and test cases and for the transformational refactoring of UML models are an excellent basis for the Model-Driven Architecture" (MDA). MDA implies a heavily model-driven approach in which different layers of models are developed in succession and are ideally generated automatically from each other. The concepts available in UML/P—such as the explicit labeling of incompleteness "...", or stereotypes such as ≪match:initial≫ for labeling the precision of an observation given by sequence diagrams—allow us to design very elegant models at different levels of abstraction and to convert them into one another by means of transformations.

UML/P and the transformations described in this volume therefore support MDA and even extend it significantly. MDA primarily supports "top-down" transformations of abstract, platform-independent models into detailed, platform-specific models and code. In contrast, the refactoring of models tends to be "horizontal": refactoring techniques improve the architecture of a system without necessarily leaving the abstraction level.

In this book, the interaction of vertical and horizontal transformations is described by the combination of code generation and refactoring. For an efficient use this requires code generation that is as automated as possible. Although this is now state of the art, it is often not possible to realize the automation adequately with the current tools available and this is therefore a differentiating feature in tool selection.

11.2 Outlook

With the process model and the underlying notation outlined in Volume 1, any reader can now gather further practical experience. Even without a mature tool, we can see that the use of UML/P and the concepts for tests and refactoring based on UML/P have positive effects on the system architecture and system quality. Transforming test models manually at least improves the understanding and the effectiveness of the development and increases the quality of any resulting tests.

However, tool developers must continue to work intensively on offering functionality for generating code and tests beyond class diagrams. We

also need general generation procedures as well as the possibility to address frameworks and component architectures when generating for specialized domains. Due to the effort involved in creating comfortable graphical software development tools, it is often better to create the tool as an extension (plug-in), for example, for an existing open source IDE. This also applies to coverage metrics for tests, generating test cases, or connecting verification tools for refactoring rules.

At the time of the original publication of both books, UML was published in version 2.3. Some concepts of the UML/P language profile are better supported in the current UML version but in general, UML 2, with its many new features, still needs more consolidation. This is precisely why the elegance, simplicity, and clarity of UML/P—which are also reflected in the comparatively short description of the abstract syntax in the appendix of [Rum16]—are a significant advantage compared to the language standard itself which suffers under the weight of too many political influences.

The consolidation provided by UML/P has created a basis for a number of extensions and investigations of usability. These include the empirical analyses of the quality and effectiveness of the known test procedures for which previously data based primarily on procedural languages existed. However, the development language used has a significant influence on the test characteristics. The number and form of practical refactoring rules and the measurement of the quality of a software structure require empirical studies that still need to be collected for a UML/P-based approach.

We can use UML/P to develop very good domain-specific language adaptations. For example, the property-oriented modeling language OCL is well suited for defining business rules for dynamic configuration and load-balancing for cloud systems which, when satisfied or violated, each trigger certain required actions. Today, complex systems such as SAP R/3 offer a number of parameters that reflect the business logic of the system. Therefore, instead of transforming OCL and other UML/P models directly into code, the system can interpret the artifacts. This enables a dynamic modifiability during system runtime, as is the case in an energy monitoring system, for example [FKP+10].

For the modelling of components and their interfaces it is advisable to use a class diagram or object diagrams. Statecharts can be used for an abstract state model of the component that the context should know about; these are often called *protocol state machines*. OCL method specifications describe the usage conditions for method calls to a component and sequence diagrams define the permitted interaction patterns. For tests, a component dummy can be created partially automatically. Conversely, we can test a component for conformity with the assured properties from outside via its interface. The use of components then becomes more interesting because automated tests can significantly increase confidence in the correctness of a component purchased from a third-party.

The Future of Modeling

It is difficult to predict how modeling techniques will be used in the software development process in the future. On the one hand, a larger part of the community beliefs that modeling still has a lot of potential for improvement in terms of both the languages and the related tools. However, it is difficult to optimize the tool landscape because tools typically cannot be used in isolation and instead, have to be an integrated part of a complex tool landscape.

Furthermore, we are seeing an increasingly more specific form of development approaches for software in the different domains. Depending on the factors that influence complexity and the risks that prevail in each domain, very different development processes are used and often, these are subject to very detailed standards. Moreover, after the wave of integration of modeling languages which reaches its peak in the UML language standard, there is a noticeable trend towards domain-specific languages (DSLs) which, although they each have to be defined separately, are comfortable to use when applied thanks to their compactness and simplicity.

In future, it should be possible to classify projects roughly into the following groups: (1) UML-based, (2) DSL-based, and (3) agile, modeling-free projects. However, the three approaches can be mixed, for example, if UML models are used for agile generation and a DSL is used for runtime configuration.

To use UML in an agile way, however, we need a much improved and more efficient tool infrastructure. Firstly, there are no good tools for versioning, for creating variants, for effective refactoring, for tracing requirements, no comfortable editors, etc.

Also important is the possibility for a lightweight use of models, as described in these books, for example to generate code. This includes, for example, that generated code does not have to be modified manually and that it does not even have to be read or understood. Unfortunately it is still often necessary today to fill code frames from class diagrams or at least to understand what functions we can program against. Models require *modularity* and explicit *interfaces* so that we can hide internal details that are encapsulated and thus elevate the success of the modularization from programming languages to models.

In addition to code generation, mature analysis and synthesis techniques are required that ensure syntactic consistency and that allow us to detect interesting properties of the system at an early stage. In particular, the techniques include automated analysis procedures that allow us to detect the violation of invariants and create examples of this. They also include synthesis procedures which calculate complete (internal) models from incompletely defined views and use these models for animation or code generation. Today, for example, it is comparatively easy to enter an OCL contract and a specific set of input data in a SAT solver which then calculates a solution that can be understood as the output of the specified method call. However, it is still not

yet possible to derive a constructive algorithm automatically from an OCL contract specification.

For heterogeneous languages such as UML in particular, the modularity of the models already mentioned also raises the question of how languages can be defined in a modular form. This is discussed in [Völ11], for example, but has not yet been applied to UML. A suitable modularization of the language UML for independent analyses and synthesis procedures also appears to be intrinsically difficult because UML was defined largely as a monolith respectively without any encapsulation mechanisms for sublanguages.

Thus, the use of a modeling language for various purposes is mentioned at various points in these books but as yet, UML has no mechanisms that would support such diversity of use. For example, depending on the form of use, Statecharts can be represented very differently in code (this is understood) but therefore also have different consistency conditions in the context of class, sequence, or activity diagrams (and that is not reflected adequately in UML).

Even though UML has certain deficits, it is currently still the best generally used modeling language in the field of software development. Many successful projects show that on the one hand, it is rather easy to use; on the other hand, current research—including research on the above-mentioned topics—shows where further improvements will be necessary.

11.3 Agile Model Based Software Engineering

Agility and modeling in the same project? Today, too many developers and project managers think that the use of models in software development leads to a heavy-weight, tedious development process. They believe that sooner or later, the models are usually outdated, are not being co-evolved, are buggy and no longer helpful. On the contrary, agility means to concentrate on the program as a core artifact without much extra documentation. Agility enables efficient evolution, correction and extension. As such, it seems to conflict with modeling.

One of our research hypotheses was initiated in [Rum04] and can be phrased like this: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process."

Fig. 11.1 illustrates that one or more domain specific modeling languages (DSML) are used as a central notation in the development process. DSMLs or the UML serve as a central notation for the software development. A DSML can be used for programming, testing and modeling.

We found that modeling will be used in development projects much more intensively, when the benefits become evident early. This means constructive generation or synthesis of code from the models needs to be among the

first steps of a model-based development process. All other interesting techniques, such as test synthesis or high level analysis techniques seem to come second. As a consequence, executability of modeling languages is an interesting future research direction.

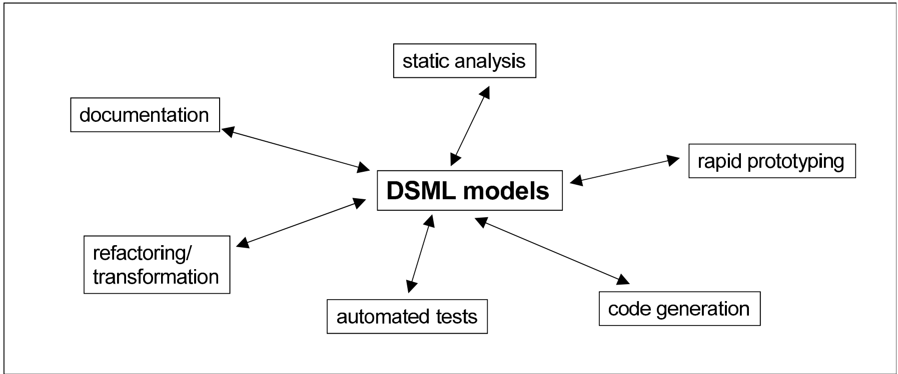


Fig. 11.1. What domain specific models can do for us

Execution of UML and DSLs

The question, whether UML should be executable, is discussed in [Rum02]. We found this a promising approach for larger subsets of the UML language, but also identified a number of challenges. We therefore started our research agenda to solve these challenges in order to make MBSE truly successful in the agile software development. We explored in detail, how UML fits for that purpose. Not only the deficiencies of existing UML tools but also the UML language itself need to be adapted to fit the needs of an agile software development process.

In [Rum03] we discussed how modeling of tests helps to increase reuse and efficiency. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code.

In [Rum16] and this book [Rum17] (German versions available under [Rum11, Rum12]), the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The UML/P embodies class, object, sequence diagrams, Statecharts and OCL in combination with Java to model code as well as tests as sketched in Fig. 11.2.

Forms of language integration, e.g., using object diagrams in the OCL to describe desired or unwanted object structures, are presented there as well.

In the last decade, we implemented a language workbench called MontiCore[1] which is initially described in [GKR$^+$06]. On top of that, we realized

[1] see www.monticore.de

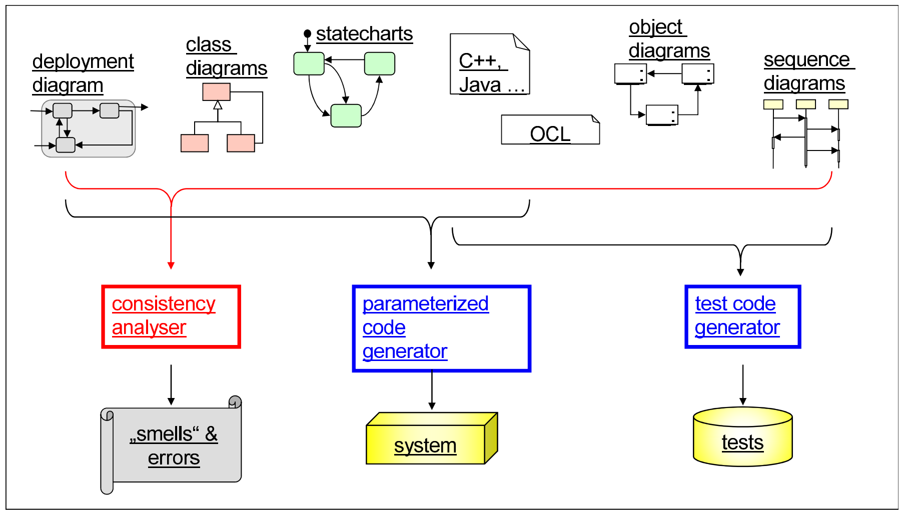


Fig. 11.2. Generation of code and tests and model analysis

most of the language components of the UML/P in [Sch12]. This includes a precise definition of the textual languages, type checks, checks for other context conditions within and between UML sub-languages and a framework for the implementation of code generators.

Specific Concepts assisting Agile Development

Agile development processes require quite a lot of specific activities, techniques and concepts that differ from documentation-based development. Research on this, e.g., includes a general discussion of how to manage and evolve models [LRSS10] or a precise definition for model composition as well as model languages [HKR+09]. Compositionality is particularly important and must be designed carefully as it allows the tools to analyze and generate incrementally, thus being much more agile than today's modeling tools. We also discussed in detail what refactoring means and how refactoring looks like in the various modeling and programming languages [PR03]. The UML/P is implemented in such a way that models can be specified free of redundancies even in different levels of abstraction, which enhances refactoring and evolution techniques on models. To better understand the effect of an evolved design, we discuss the need for semantic differencing in [MRR10].

When models are the central notation, model quality becomes an important issue. Therefore, we have described a set of general requirements for model quality in [FHR08]. We distinguished between internal and external quality. External quality refers to the correctness and completeness of a model with respect to the original that it describes, while internal qual-

ity refers to the model presentation and thus plays the same role as coding guidelines for programming languages.

We also know that, even when using the UML/P, there is additional effort necessary to adapt the tooling infrastructure to the project specific needs. This becomes more pressing, when a domain specific language is specifically designed for a project. [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project.

We assume that the use of models at runtime will become a pretty agile and efficient development technique. It allows developers to delay design decisions to runtime adaptation and configuration of systems. However, reliability then becomes an issue. In [CEG+14] we have therefore discussed how to improve reliability while retaining adaptivity.

11.4 Generative Software Engineering

In Section 11.3 we clarified that generating software is an important capability for a tooling infrastructure that successfully assists modeling in the development process. We believe that modeling will only become an integral part of the process in many industrial projects, if automatic derivation of executable code and smooth integration with handwritten code is a standard feature of its tooling.

We therefore examined various aspects of generation. E.g. in [Rum16] and this book, we define the language family UML/P (a simplified and semantically sound derivative of the UML) which is designed specifically for product and test code generation from class diagrams, object diagrams, Statecharts and sequence diagrams as shown in Fig. 11.2.

[Sch12] developed a flexible, modular and reusable generator for the UML/P based on the MontiCore language workbench ([KRV10, GKR+06]). With MontiCore we are able to easily define extensions of languages as well as new combinations and thus are able to reuse the defined UML/P sublanguages and generation techniques in various applied projects.

Our architectural analysis and design language (AADL) MontiArc is also based on this generation technology. As described in [HRR12] it can be used for the cloud as well as Cyber-Physical Systems, such as cars or robotics.

Tooling and especially generators will only be successful in practical projects, if they have an appropriate impact on the development process, i.e. development processes need to be adapted or completely reshaped according to the availability of a generator. In [KRV06], we discussed additional roles necessary in a model-based software development project (while other roles either vanish or their workload can greatly be reduced).

The generation gap problem is addressed in [GKRS06]. There, we discuss mechanisms to keep generated and handwritten code separated, while integrating them in the product and enabling the repetitive generation (which is much more valuable than one-shot generation).

For various purposes, including preparation of a model for generation, it is helpful to define model transformations. We are able to create transformation languages in concrete syntax, that reuse the underlying language concepts. In [Wei12] we show how this looks like. Even more important we describe how to systematically derive a transformation language in concrete syntax. Since then we have applied this technique successfully for several UML sublanguages and DSMLs.

Sometimes executability can be a disadvantageous characteristics for a modeling language, especially when people start modeling concrete algorithms instead of abstract properties. We therefore discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] as well as the advantages and perils of using modeling languages for programming in [Rum02].

11.5 Unified Modeling Language (UML)

Many of our contributions build on UML/P are described in the two books [Rum16, Rum17] and implemented in [Sch12].

UML's semantic variation points are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "system model" [BCGR09a, BCGR09b, BCR07b, BCR07a]. They have e.g. been applied to define class diagram semantics [CGR08] and activity diagram semantics [GRR10].

Precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences for ADs [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10].

We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04, Rum02]

The question, how to adapt and extend the UML led to [PFR02] on product line annotations for UML and to more general discussions and insights on how to use meta-modeling for defining and adapting the UML [EFLR99, FELR98c, SRVK10].

A very early discussion on the challenges for the UML discussed by the pUML group can be found at [KER99].

11.6 Domain Specific Languages (DSLs)

People are modeling everywhere. Both science and philosophy use models to understand and describe the concepts and phenomena in their fields. En-

gineering disciplines use models to describe the systems they want to design. We all use models, but only computer science defines and studies the set valid of models, namely the modeling language explicitly. This is made necessary because computer scientists use models not only to communicate among each other, but also with computers.

Computer science, therefore, investigates very much into languages. We use universally applicable modeling languages to describe problems and problem contexts. We employ general-purpose programming languages to implement solutions. We specify properties, architect and design solutions. And we define tests, as well as an increasing number of application specific languages and domain specific languages (DSLs) tailored for a concrete target area.

A DSL is always constructed with a particular domain in mind. Examples include HTML for websites, Matlab for numerical computation, or SQL for relational database management. In each instance the DSL trades some of the expressiveness of GPLs in order to allow for more concise models in the target domain.

As software systems have become essential components of nearly all innovative products, increasingly many non-ICT experts now find themselves working with these systems.

Furthermore, complexity of software-based systems is increasing. While modeling languages such as the UML provide a high level of abstraction to deal with complexity, these languages are usually still too technical (hence UML profiles are useful, as discussed in [GHK+07, PFR02]). DSLs address both of these problems. Non-ICT experts benefit from DSLs by being able to transfer already familiar language concepts to the new application. Experienced users benefit by having a smaller mental gap between the software system and the associated real world models.

The main drawback of domain specific languages currently is still their challenging creation process. Not only does the creation of a computer language necessitate the fundamentals, such as a carefully defined grammar and corresponding translation programs. Productive usage of a language also requires extensive tool support. Generative Software Engineering techniques (see Section 11.4) are at the center of attention for attempts to meet these challenges. In [SRVK10] we discuss the state of the art and current efforts to develop languages through meta modeling.

Fig. 11.3 depicts the role of DSLs in a model-based Software Engineering process. DSLs and the models expressed with them are becoming first-class elements of the Software Engineering process. In order to support this development, research needs to be focused on new, effective, and efficient ways of creating DSLs and corresponding tool support.

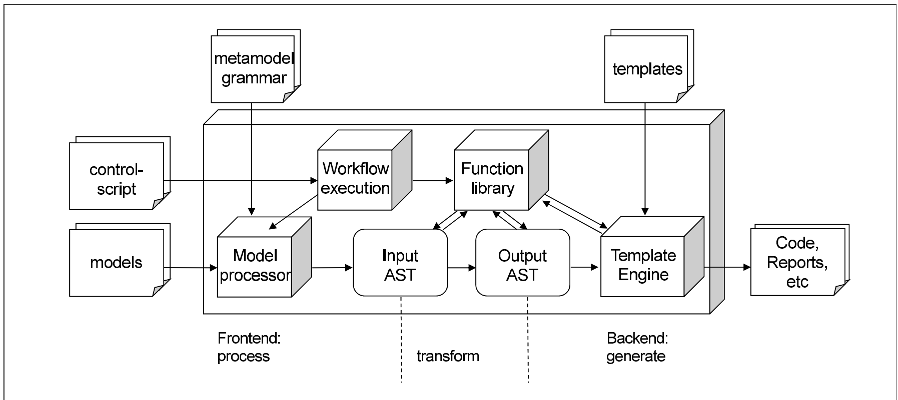


Fig. 11.3. Tool support for defining and using a DSL

DSL Language Definition

DSLs have to be designed carefully to meet their respective requirements. A core design of a DSL consists of a desired concrete and abstract syntax. We examine the relations between concrete and abstract syntax and propose a language definition format in [KRV07b, KRV10], which allows the combined definition of concrete and abstract syntax.

In [FHR08] we discuss metrics and potential guidelines, that help to achieve high quality models and extend this into a collection of design guidelines for DSLs in [KKP+09]. Our experience shows that these guidelines tremendously improve the quality of DSLs. They target and enable suitability, reuse, conciseness, and usability.

How to define the semantics of DSLs is discussed in Section 11.10. The aspect of variability in syntax and semantics for DSLs in general and UML in particular has been discussed in [GR11].

Composition of DSLs

Modularity is a key concept in software development and the enabler for efficient reuse. We investigated the application of modularity to the development of DSLs in [GKR+07, KRV08, Völ11]. Modularity has been successfully applied in various areas of the DSL development process, such as concrete and abstract syntax, context conditions, and symbol table structures and has been implemented in our language workbench MontiCore.

We can compose independently developed languages into integrated families of DSLs, which allows us to describe a system from various viewpoints using these different DSLs. The language family UML/P, defined in [Sch12], serves as an example of this technique.

As described in [KRV08] we can inherit from existing languages and adapt certain language concepts. An often used example is to extend an action language by new forms of actions.

We can reuse existing languages by embedding them as sub-languages. E.g. Java's expression language can be used for various purposes within a modeling DSL. This way we have integrated Java statements and expressions into UML/P. We are further investigating the decomposition of generators and modular composition of generated code.

These concrete techniques are summarized in the broader discussion on the so called "global" integration of domain specific modelling languages and techniques in a conceptual model [CBCR15], which is published in [CCF+15].

DSL Tooling

As previously mentioned, the usability of a language depends on the availability of powerful tooling. We have implemented the MontiCore DSL workbench as a realization of all the aforementioned concepts regarding DSLs. It is available as a stand alone tool as well as a collection of Eclipse plugins. It also creates stand alone tools as well as tailored Eclipse-based plugins for the defined DSLs [KRV07a]. We generate editors with syntax highlighting, syntactic and semantic content assist and auto completion, graphical outlines, error reporting, hyperlinks etc., just from the DSL definition.

In [LRSS10] we discuss the need for evolution and management of models. We especially identify the need for comfortable transformation languages. Therefore, [Wei12] presents a tool that creates an infrastructure for transformations that are specifically dedicated to an underlying DSL. The generated transformation language is quite understandable for domain experts and comes with an engine dedicated to transform models of this DSL.

MontiCore

More details about the MontiCore DSL workbench can be found in [GKR+06, KRV08, KRV10] as well as the MontiCore website[2].

11.7 Software Language Engineering (SLE)

Identifying or engineering appropriate languages for the various activities in software and systems development is one of the most important issues in Software Engineering. Even programming languages are still subject of improvement. For many other activities, such as architectural design, behavioral modeling, and data structure specifications, we use the general purpose

[2] see www.monticore.de

UML. But UML and its tooling still are much less elaborate and hence subject to extensive syntactic, semantic, and methodic improvement.

In various domains, however, it is more appropriate to employ Domain Specific Languages (DSLs) (see Section 11.6) to enable non-software developers specifying properties, configuring their systems, etc. in terms of established domain concepts and corresponding language elements. With the upcoming age of digitalization, we thus expect a growth in domain specific languages (DSLs) and increasing efforts in their efficient engineering, integration, and composition.

Design of a domain specific language is a complex task, because, on the one hand, it needs to be precise enough for being processed by a computer and, on the other hand, comprehensible by humans. Monolithic design of a language can already benefit from methods for language engineering in the small including design guidelines and tooling. The MontiCore language workbench is such a tool to assist development of languages. It, for example, provides techniques for an integrated definition of concrete and abstract syntax of a language [KRV07b, Kra10], but is much more a framework for compositional language design [KRV10].

Language Engineering in The Large

To efficiently engineer languages in the large, we need all the help that we can get. As software languages are software too, it is not surprising that the following techniques help:

1. Elaborate tooling to assist language development.
2. Reuse of tools, e.g. for parsing and for parameterizable pretty printing.
3. Reuse of language components.
4. Decomposition of the language to be designed in smaller components.
5. Refinement and adaptation of existing languages.
6. Automatic derivation of new languages from existing ones.

To improve understanding of language engineering, we have defined the terms language and language components in [CBCR15] and how to capitalize on this in [CCF^{+}15]. In [SRVK10], we discuss the possibilities and the challenges using metamodels for language definition, identifying, for instance, the need for metamodel merging and inferencing, as well as assistance for their evolution.

Language and Tool Composition

"Divide et Impera" is the core concept of managing large and complex tasks. Language design therefore needs to be decomposed along several dimensions: First, we want to decompose the language in language components. Some of these components, for example the basic language for full qualified

names, constants, (Boolean) expressions, or imperative statements, should be designed in a reusable form.

In a second dimension, we want to decompose the tooling along the activities (frontend: model processing, context conditions, internal transformations, backend: printing) and decompose each of these activities along the individual language components. MontiCore 3, e.g., is able to decompose the frontend language processing along the decomposition of the language itself [KRV10, Völ11, KRV08, HMSNRW16]. MontiCore also assists modular decomposition of the backend code generation based on different targets and different sublanguages [RRRW15].

Language Derivation

Language derivation is, to our believe, a promising technique to develop new languages for a specific purpose that are relying on existing basic languages. Formally, a language derivation is a mapping D, that maps a base language B into another language $D(B)$. This mapping produces new languages, not models. To automatically derive such new languages $D(B)$ or, at least, assist such derivation with tools, the base language B itself has to be modeled explicitly, for instance as a metamodel or as a grammar together with its well-formedness rules in a reasonably explicit form. Thus, language derivation can be partially understood as model transformation on a metalanguage. We so far successfully conceived three language derivation techniques.

Transformation languages in concrete syntax

Instead of using a fully generic transformation language that is applicable to a base language B, we automatically derive a transformation language $T(B)$ that merges elements of the concrete syntax of B with generic—and thus reusable—elements for defining transformations on B models. The result is a comprehensible and easy applicable transformation language that modelers find familiar, because it systematically reuses the syntax of the base language B. Automatic derivation of such transformation languages using concrete syntax of the base language is described in [HRW15, Wei12].

Because the language derivation operator T is applicable to any language, we have successfully applied it to class diagrams, object diagrams, MontiArc, automata, and more. The operator T not only derives the new languages $T(B)$, but the tool infrastructure behind T also generates the transformation engine necessary to execute transformations defined in $T(B)$ (which finally transform models of the base language B).

Tagging languages

A tagging model is used in the context of a base model M and adds additional information in form of tags to the elements defined in M. This, for example, can be used to add technology-specific information or advice on how

code generation, model merging and other algorithmic transformations have to handle the tagged elements. Tags can, for example, instruct a persistence generator, whose data model classes are mapped into single transportable DAOs or add security restrictions to data objects. For activity diagrams, tags can describe, where to find the appropriate activity implementation, etc.

Tagging models share the idea of UML's stereotypes, but are not part of the base model. Instead the separate tagging model references the base model. This has the advantages (1) that the base model can be reused without technology specific pollution, (2) several different tag models are possible for the same base model in different technological spaces (e.g., iPhone, Android or Windows clients), and (3) a tag model can also be reused for different base models.

A tagging language is the language of the tagging models and thus is highly dependent on the base language that it tags (i.e., it must be aware of the modeling elements of the base language). [GLRR15] describes how to systematically derive tagging languages from a base language and how code for processing tagging models can be generated automatically.

This also rests on the concept of a tag definition language, which allows defining the possible form and values that tags may have, as well as which kind of model elements they can be applied to and therefore acts as type definitions for tags.

Delta languages

Another way of deriving new languages from existing languages is described in [HHK+15a, HHK+13], where a base language B is used to derive a delta language *Delta(B)*. The delta language *Delta(B)* enables to explicitly describe differences between a base model of B and the model variant (also of B). This helps to define system variability in a bottom-up fashion. A delta model describes which model elements are added, modified, or deleted on the base model. Thus the delta approach is popular for the management of variability and Software Product Lines (SPL) as discussed in 11.9.

Delta language techniques are specifically suited for architectural languages, such as MontiArc (see Section 11.8) to add and modify components as well as channels, but also have been applied to Simulink in an industrial context.

Again the delta operator transforms a base language B into a language *Delta(B)* allowing to describe delta models. Each delta model can be applied individually and therefore n deltas amount to 2^n variants (modulo application dependencies and orders).

11.8 Modeling Software Architecture and the MontiArc Tool

Distributed interactive systems have become more and more important in the last decades. It is becoming the standard case that a system is distributed. Typically such systems consist of subsystems and components like

- sensors, control units, and actuators in cyber-physical machines,
- high performance computing nodes,
- big data storage nodes,
- messages transmitted between web services in cloud computing applications, or
- interaction between mobile humans.

The main paradigm for communication in distributed systems is asynchronous message passing between actors. The logical or physical architecture of a hierarchically decomposed system can be modeled like the excerpt of a car locking device in Fig. 11.4.

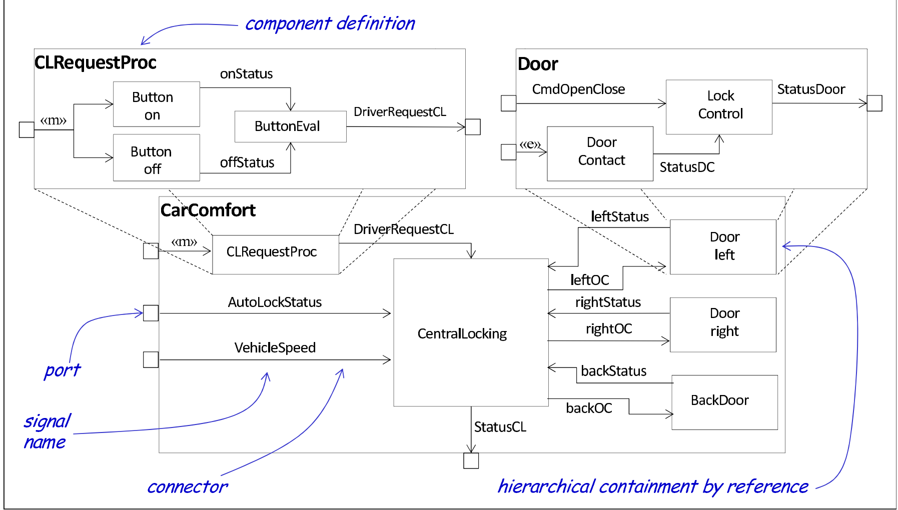


Fig. 11.4. Hierarchic architecture model e.g. used in the automotive domain

Messages can be

- event signals, e.g., messages on a bus,
- values measured by sensors and discrete event signals,
- streams of telephone or video data,
- method invocation, or
- complex data structures passed between software services.

Semantically our approach is formally sound and well-defined using streams, state machines, components, as well as expressive forms of composition and refinement (see Section 11.10). A challenge in the design and implementation of these systems is the development of an appropriate architectural decomposition of the system and fitting component interfaces suitable for property analysis, effective realization, and reuse of components under variability considerations. We have made a number of contributions to this field from more theoretical considerations up to a concrete tooling infrastructure called MontiArc.

Theoretical foundation of Software Architecture Modeling

A theoretical foundation of a model-based development in terms of an integrated, homogeneous, but modular construction kit for architectural models is described in [BR07]. Mathematical foundations are given for modeling of interfaces, building architectures through composition and decomposition, layering architectures as well as hierarchical decomposition, and implementation of components using state machines. Especially the refinement (see also [PR99]) of hierarchy, interfaces, and behavior is discussed as well as abstraction mechanisms for the integration of abstract viewpoints. The presented theory consists of a set of theorems and provides a basis for architectural modeling without sticking to a concrete syntax of a modeling language.

MontiArc - Architecture Modeling and Architectural Programming

The architectural design language MontiArc has been developed for modeling distributed interactive systems. It captures active components (agents, actors) of a logical or physical distribution, their interfaces (ports), the communication infrastructure between the components, and a hierarchic decomposition. MontiArc is a full ADL, although we have omitted some uninteresting concepts from the AADL standard and could then optimize others.

MontiArc is described in [HRR12] in detail. MontiArc is a textual language and comes with an eclipse-integrated editor. It provides a simulation framework that can execute behavior implemented in Java and attached to MontiArc models in a declarative way so that analysis on MontiArc models becomes possible.

Because the language MontiArc is designed for extensibility, several sublanguages for behavior may be embedded directly within component definitions. MontiArc is e.g. extended with automata to MontiArcAutomaton [RRW13, RRW14]. In [HRR10], an extension of MontiArc with Java is presented, which becomes a full programming language that exhibits architecture, data structure and behavior. [RRRW15] describes how the language is composed of individual sublanguages. With this approach, a smooth integration of architectural design and programming is achieved. We call this architectural programming.

Architectural Variability

Section 11.9 discusses our contributions in more detail, but it is worth to mention that much variability research was applied to and experimentally verified using MontiArc.

Variability of a system has to be considered and modeled by appropriate means during all phases of the development but especially in the architectural design. MontiArc has thus been extended in two different ways, hierarchical variability modeling and delta modeling, in order to explore ways to enable architectural modeling of variants defined in a product line.

In [HRR+11], we explored a variability mechanism based on MontiArc that allows specifying component variants fully integrated at any level of the component hierarchy. Here variation points may have hierarchical dependencies. Associated variants define how this variability can be realized in component configurations. As a general drawback of this approach, systems are restricted to the set of predefined variations and cannot be extended. This approach is not additive.

We thus explored delta modeling as an additive approach to variability design. This will allow a company to immediately start to develop and think in terms of product lines, even years before the full variability model is extracted (reengineered) from former and ongoing projects. The main idea is to represent any system by a core system and a set of deltas that specifies modifications. In [HRRS11] we describe Delta-MontiArc (also Δ-MontiArc), which applies this concept successfully to MontiArc. The core is a MontiArc model. A delta language is defined describing how to add, remove, or modify architectural elements. The concrete realization of Δ-MontiArc using MontiCore [3] is described in [HKR+11a]. The developed language allows the modular modeling of variable software architectures and supports proactive, reactive as well as extractive product line development. As a next step, we explored in [HRRS12] how to evolve a complete delta-based product line, e.g. by merging or splitting deltas.

Requirements, Evolution, Dynamics of Architecture

A methodological approach to close the gap between the requirements architecture and the logical architecture of a distributed system realized in a function net is described in [GHK+07, GHK+08]. It supports the tracing of requirements to the logical software architecture by modeling the logical realization of a feature that is given in a requirement in a dedicated feature view. This allows us to break down complexity into manageable tasks and to reuse features and their modular realization in the next product generation. [GKPR08] extends this modeling approach to model variants of an architecture. These concepts are now successfully integrated into automotive development processes.

[3] see www.monticore.de

We also defined a precise verification technique that allows developers to decompose logical architectures into smaller pieces of functionality, e.g. individual features in [MRR13, Rin14], and to verify their consistency against a complete architecture in [MRR14]. Our hypothesis is that with this technique, developers will be able to decompose requirements into features and compose their implementation late in the development process. That will definitely increase reusability of features.

An overview and a detailed discussion on the challenges of co-evolution of architectural system descriptions and the system implementation is given in [MMR10]. Architectural descriptions of a system deal with multiple views of a system including both its functional and nonfunctional aspects. Especially, critical aspects of a system should be reflected in its architecture. The description must also be accurately and traceably linked to the software's implementation so that any change of the architecture is reflected directly in the implementation, and vice versa. Otherwise, the architecture description will rapidly become obsolete as the software evolves to accommodate changes.

While many architecture styles assume static structures, we explored a modeling technique to describe dynamic architectures in [HRR98]. It allows developers to express dynamically extensible interfaces of components with so-called Component Interface Diagrams (CID).

11.9 Variability and Software Product Lines (SPL)

Most products, like cars, printers, mobile phones, etc., exist in various variants. Software for product variants is quite similar, but typically differs in new or additional features that sometimes deeply affect the software's architecture. Software variants are managed as a Software Product Line (SPL) that captures the commonalities as well as the differences. Software Product Lines have many benefits, they:

- decrease development time of new product variants,
- decrease time to market,
- lead to better software quality,
- improve reuse, and
- reduce bug fix time.

Variability is to a larger extent related to evolution. We discuss our approaches to evolution understanding in Section 11.12.

Feature diagrams and Views

Feature diagrams describe variability in a top down fashion in the problem space. We studied the application of this top down approach e.g. in the automotive domain in [GKPR08].

Feature diagrams suffer from the need to first decompose the problem space and understand possible features in order to build the feature diagram before being able to apply it. In [GHK+08, GKPR08] we also speak of a 150% model. This normally enforces a product line definition phase in which the requirements and features need to be collected which creates additional costs. Among others we discuss decreasing these costs in [GRJA12].

Delta Modeling

We discuss delta modeling as a bottom up SPL modeling technique in [HRR+11]. Deltas can both be used as substitute and as extension to traditional feature based development. Deltas allow us to build a product line incrementally starting with a base variant when the need for a new feature arises. Starting with a core version, each delta describes the changes necessary to derive a new variant. Deltas allow to add, replace, modify and delete components of a model resp. implementation and is thus rather general.

Each set of valid deltas configures a product variant. We have successfully applied delta modeling to the architectural analysis and design language (ADL) MontiArc by creating Delta-MontiArc [HRR+11, HRRS12] as well as applied it to Simulink creating Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows using them for Software Product Line evolution [HRRS12]. In [HHK+15a] we have generalized this approach to the general question, how to synthesize a delta modeling language based on a given modeling language. Thus deltas can generally be applied to other languages too.

Variability in Language Definitions

On a related line of research, we also have studied variability of modeling languages, which allows us to define and reason about syntactic and semantic variation points, which is e.g. in the UML a big topic as it seems the UML standard will otherwise not be able to accommodate all stakeholder requirements.

For this purpose we defined a systematic way to define variants of modeling languages [CGR09]. We applied this research e.g. in the form of semantic language refinement on state charts in [GR11]. In [PFR02] we discussed how to apply annotation to the UML to describe product variation points.

SPL and Delta Modeling in Industry

We have introduced SPL and delta modeling in several companies and are proud of successfully helping companies to manage their variants. We also learnt, that industrial success means that each company needs a tailored process that fits the company culture, used tool chains, size of products and the desired agility of variant construction. SPL does not come free of initial cost.

A typical SPL introduction process consists of three stages: (1) Understanding the current situation in the company. Current process? Size of projects? Number of existing and planned variants? How similar are those? Current costs of evolution for individual products? Available and desired tool chain? (2) Derivation of a long list of potential technical, process and organizational measures for an SPL based future with efficient development of high quality systems. Categorization and prioritization. (3) Implementing the most promising steps and understand the effects.

11.10 Semantics of Modeling Languages

Over the years we have explored analysis, synthesis, evolution, definition of views, and abstraction based on models in deep detail. For all these purposes, we need a sound semantical foundation of the meaning of the models.

We also need a proper semantics when applying a given language to new domains, such as monitoring energy consumption or modeling flight safety rules for the European air traffic [ZPK+11]. We do this regularly with our tool workbench MontiCore [KRV10].

The Meaning of Semantics and its Principles

Over the years we have developed a clear understanding of what the semantics of a model and a modeling language is. For example in [HR04] we discussed different forms of semantics and what they can be used for. We in particular distinguish between "meaning" that can be attached to any kind of modeling language and an often used narrow interpretation, that uses "semantics" synonymously to behavior of a programm.

Each modeling language, let it be UML or a DSL deserves a semantics, even if the language itself is for modeling structure, such as Class Diagrams or Architecture Description Languages. Furthermore, modeling languages are not necessarily executable and as their main purpose is abstraction from implementation details, they are usually not fully determined, but exhibit forms of underspecification. We discuss a very general framework for semantics definition in [HR04]. At the core, we use a denotational semantics, which is basically a mapping M from source language L (syntax) into a target language respectively a target domain S (semantic domain). Here we see a combination of functions, where the first simplifies the syntax language by mapping redundant concepts to their simplest form (less concepts used, but usually more complex models) as shown in Fig. 11.5.

While many attempts of defining semantics only give examples on how the mapping M looks like, we advocate an explicit and precise definition of M to be able to analyze or compare the semantics of models. E.g. refinement and evolution of models rely on such explicit denotational semantics.

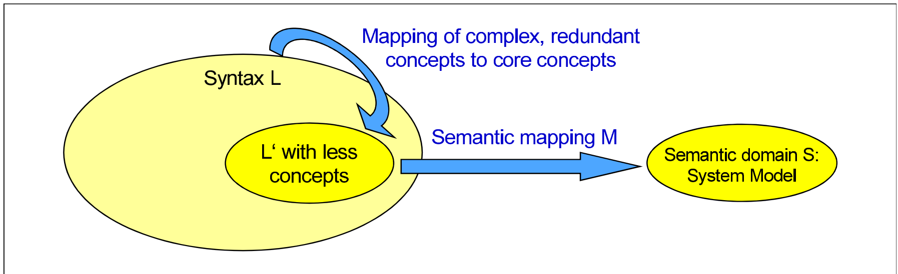


Fig. 11.5. Denotational semantics for a language defined in several steps

System Model as Semantic Domain

To define a semantic domain we use a mathematical theory, that allows us to explicitly specify the desired properties of the target system, we are aiming at. We call the developed theory system model. Its first version is explicitly defined in [RKB95, BHP$^+$98] (including [GKR96, KRB96]).

The system model for the full UML however became a rather large mathematical theory, that captures object-oriented communication (method calls, dynamic lookup, inheritance, object identity) as well as distributed systems at various levels as states and statemachines. We therefore developed the full system model for the UML in [BCGR09b] and discuss the rationale for it in [BCGR09a]. See also [BCR07a, BCR07b] for more detailed versions and [CGR08] for an application on class diagrams. In Fig. 11.6 we see the hierarchy of the mathematical model.

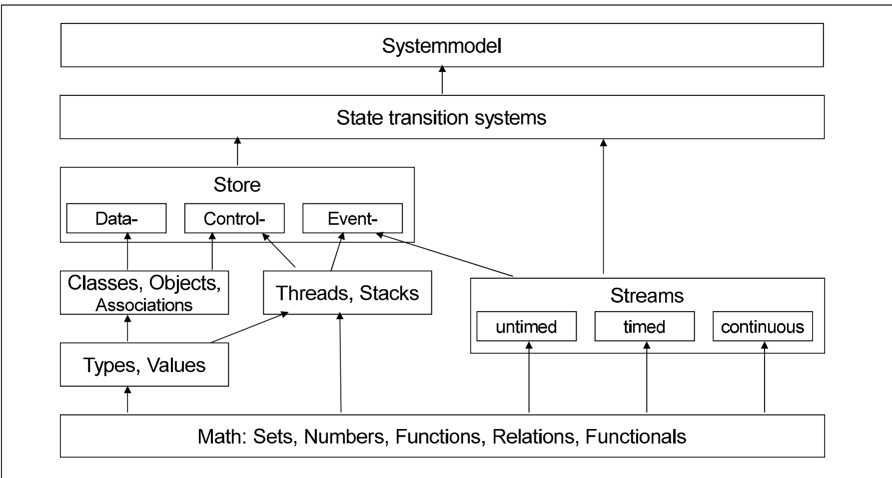


Fig. 11.6. System model as foundation for semantics

The system model and its variants are used for a variety of tool embeddings of the semantic domain. [MRR10] explains the case for semantic model differencing as opposed to syntactic comparison. For example in [MRR11a] (based on [MRR11b]) we encoded a part of the semantics, big enough to handle differences of activity diagrams based on their semantics, and in [MRR11e] we compare class and object diagrams based on their semantics.

In [BR07] we have defined a much simpler mathematical model for distributed systems based on black-box behaviors of components, hierarchical decomposition, but also the sound mathematical theory of streams for refinement and composition. While this semantic model is useful for distributed real-time systems, such as cloud, Internet or Cyber-Physical Systems, it does not exhibit concepts of objects and classes.

We also discussed a meta-modeling approach [EFLR99]. As nothing is as mighty and comfortable as mathematical theories, one needs to carefully design the semantics in particular if a concept of the language does not have a direct representation in the semantics domain. Using a meta-model to describe the semantics is appealing, because the syntactic domain L is meta-modeled anyway, but also demanding, because both the semantic domain S and the mapping M need to be encoded using meta-modeling instead of mathematical concepts. We learnt, that meta-modeling is limited, e.g. in its expressibility as well as due to finiteness.

Semantics of UML and Object-Orientation

In the early days, when modeling technology was still in its infancy is was of interest to precisely understand objects, classes, inheritance, their interactions and also how modeling technologies, like the upcoming UML, describe those. [BGH$^+$97] discusses potential modeling languages for the description of an exemplaric object interaction, today called sequence diagram, and a complete description of object interactions, which obviously needs additional mechanisms e.g. a sequential, parallel or iterative composition of sequence diagrams.

[BGH$^+$98] discusses the relationships between system, a view and a complete model in the context of the UML.

Abstraction, Underspecification and Executability

A modeling language is only a good language, if it allows to abstract away from certain implementation details. Abstraction however often means that its models are not fully determining the original, but exhibit underspecification. Underspecification is regarded as freedom of the developer or even of the implementation to choose the best solution with respect to the given constraining specification. It is an intrinsic property of a good modeling language to allow underspecification.

As a consequence a semantic mapping of an (underspecified) model into a single running programm cannot be correct or useful (to capture the semantics adequate). To tackle underspecification we use a set-based mapping. This means a single model is mapped to a set of possible implementations all of which fulfill the constraints given by the model. This approach has important advantages:

1. Each element in the semantics can be an executable implementation, we just do not know, which of them will be the final implementation.
2. Given two models, the semantics of composition is defined as intersection: these are exactly the systems that implement both models. This approach is based on “loose semantics”, where an implementation is allowed to do everything that has not explicitly been forbidden by the specification.
3. A model is consistent exactly when it has a nonempty semantics.
4. Refinement of a model on the syntactic level maps to set inclusion on the semantics.

Using sets of executable systems in the semantic mapping combines the denotational approach with an operational approach that is perfectly suited for semantics for modeling languages.

Semantic Variation Points

In the standardization of the UML the contributors had some challenges to agree on the meaning of quite a few modeling concepts. To some extent this is due to political reasons (tool vendors try to push their already implemented solution), but to a large extent this is also due to the attempt of the UML to describe phenomena in various real world and application domains as well as software/technical domains. As it is a bad idea to capture different phenomena with the same syntactical concept, the UML Standard introduces the semantic variation point without describing precisely what it means and how to describe it.

In [GR11, CGR09] we have discussed the general requirements for a framework to describe semantic and syntactic variations of a modeling language. We also introduced a mechanism to describe variations (1) of the syntax, (2) of the semantic domain, and (3) of the semantic mapping using feature trees for class diagrams and for object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. Feature trees are a perfect concept to capture variation points and denotational semantics based on a system model allowing to explicitly describe the effect of the variant.

In this book we have embodied the semantics in a variety of code and test case generation, refactoring and evolution techniques to make UML semantics amenable to developers without exposing the formalism behind. In [LRSS10] we have discussed evolution and related issues in greater detail.

Streams and Automata as Semantic Foundation

Just as a short notice, we have used the mathematical concept of streams (e.g. Broy/Stolen [BS01b]) and various extensions including automata [Rum96] as semantic basis for the kind of systems, we have in focus: distributed, asynchronously communicating agents, which can be regarded as active objects.

11.11 Compositionality and Modularity of Models and Languages

"Divide and conquer" as well as "abstraction" are the most fundamental strategies to deal with complexity. Complex (software) systems become manageable when divided into modules (horizontally, vertically and/or hierarchically). Modules encapsulate internal details and give us an abstract interface for their usage. Composing these modules as "black boxes" allows us to construct complex systems.

Model-Based Software Engineering (MBSE) uses models to reduce complexity of the system under development. MBSE however has reached a point, where models themselves are becoming rather complex. This clearly rises the need for suitable mechanisms for modularity within and between models. In [BR07] we have described such a set of compositional modeling concepts, perfectly suited for modular development of interacting systems.

A modular approach for MBSE cannot only help us mastering complexity, but is also a key enabler for model based engineering of heterogeneous software systems as discussed in [HKR+09].

A compositional approach has to take into account several levels of the entire MBSE process, starting with the respective modeling language in use, the models themselves and, eventually, any generated software components. We have examined various aspects of model composition in [HKR+07], describing a mathematical view on what model composition should be. It defines the mechanisms of encapsulation, and referencing through externally visible interfaces.

[KRV10, KRV08] examine modularity and composition for the definition of Domain Specific Languages (DSLs) or Domain Specific Modeling Languages (DSMLs). Since DSLs are becoming more and more popular, reuse of DSL fragments (i.e. language components) is vital to achieve an efficient development process. But aside from the language definition, the accompanying infrastructure needs to be modular as well (as described in [KRV07a]). Infrastructure such as validation or editor functionality should be reusable if parts of the underlying DSL are reused, e.g. as part of another language. [Völ11] provides the underlying technology for compositional language development, which we e.g. applied to robotics control [RRRW15].

Based on the experiences in language design, we have defined a set of guidelines to derive a good quality of a DSL in [KKP+09].

We have summarized our approach to composition and the challenges that need to be solved in [CBCR15], in form of a conceptual model of the compositional, so called "globalized" use of domain specific languages, which we published together with related topics in [CCF+15].

As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional, e.g. technical information for model elements in extra documents and thus facilitates reuse of the original model in different contexts with individual tag sets, but also of tags on different models. It furthermore allows to type the tags.

11.12 Evolution and Transformation of Models

Models are central artifacts in model-driven software development (MDD). However, software changes over time and so do models. Many of the new requirements imposed by stakeholders, technology adaptations, or bug and performance improvements do not only affect the implementation, but also require an evolution, refinement or refactoring of the models describing the system. When models play a central role in the development process, it is therefore necessary to provide a well-founded, methodologically sound and tool-based assistance for evolving models according to changing needs.

Evolution

Agile methods, such as XP or Scrum, to a large extent rely on the ability to evolve the system due to changing requirements, architectural improvements and incremental functional extensions. While agile methods use code as their central artifacts, a model-driven method concentrates on modeling artifacts. In [Rum04] and Chapter 2 of this book we describe an agile model-based method that relies on iterated and fully automatic generation of larger parts of the code as well as tests from models, which in turn enables us to apply evolutionary techniques directly on the various kinds of models, e.g. the UML. We argue that combining automatic and repeatable code generation with tool-assistance for model transformation allows to combine agile and model-based development concepts for a new and effective kind of development process.

An overview on current technologies for evolving models within a language and across languages is given in [LRSS10]. We refined this with a focus on evolving architecture descriptions for critical software-intensive systems [MMR10].

Refinement

Refinement is a specialized form of transformation of models that adds information, while all conclusions a developer could derive from the abstract

model still hold. Stepwise refinement is therefore an important development technique as it prevents unwanted surprises when abstract models are implemented.

In [PR94] we developed a precise understanding of automaton refinement that is especially useful for software development, as it uses a loose semantics approach, where no implicit assumptions are made that need to be invalidated in the refinement steps. In [KPR97] we applied this refinement concept to feature specifications.

Finally, we developed a powerful set of refinement rules for pipe-and-filter architectures in [PR99]. Its rules allow us to refactor the internal structure of an architecture, while retaining respectively refining the externally promised behavior. We speak of "glass box" refinement as opposed to "black box" refinement, where only the external visible behavior is taken to consideration, and "hierarchical decomposition", where a black box behavior is decomposed into an (forthwith immutable) architecture.

Refactoring of models

Refactoring aims to improve the internal structure while preserving its observable behavior and became prominent with agile development.

In [PR01] we traced back refactoring of programs to related techniques e.g. known from math or theorem provers. In [PR03] we have discussed, the existing refactoring techniques for specifications and models. We, e.g., found a number of well defined refactoring techniques for state machines, logic formula, or data models that come from formal methods, but have not yet found their application in software development. In Chapter 9 we therefore discuss refactoring techniques for various UML diagrams in detail.

If a model refactoring is actually a refinement, then dependent artifacts are not affected at all. However, it may be that a refactoring does have effect on related artifacts. In [MRR11a] we discuss a technique to identify semantic differences for UML's activity diagrams. It can be used to understand the effects of a refactoring resp. evolutionary change.

In [MRR11c] we provide the mapping of UML's class diagrams to Alloy allowing to understand semantic differences between refactoring steps on data structures by exhibiting concrete data sets (object structures) as witness of semantic differences.

Understanding model differences

While syntactic differences of models are relatively easy to understand, it is an interesting question that given two models, e.g. where one evolved from the other, and a clear semantics (see Section 11.10), what are the semantic differences between those models? In [MRR10] we discussed the necessity for this and since then have defined a number of semantic based approaches for this.

We also applied compatibility checking of evolved models on Simulink, e.g. in [RSW+15].

Delta transformations to describe software variability

Software product line engineering is most effective, if planned already on the modeling level. For this purpose, we developed the delta approach for modeling. Each delta describes a coherent set of changes on a model. A set of deltas applicable to a base model thus describes a model variant (see also Section 11.9).

We applied delta modeling on software architectures in [HRRS11] and extended this into a hierarchical approach in [HRR+11]. Second, we discussed in [HRRS12], how to evolve a complete product line architecture, by merging deltas, or extracting sub-deltas etc., which allows us to keep a product line up to date and free of undesired waste. Third, based on the experience we gained from applying the delta approach to one particular language, we developed an approach to systematically derive delta languages from any modeling language in [HHK+13, HRW15].

Model transformation language development

As we do deal with transformations on models in various forms, we are very much interested in defining these transformations in an effective and easily understandable form. Today's approaches are concentrated on the abstract syntax of a modeling language, which a typical developer should not be aware of at all. We heavily demand better transformation languages.

In [Wei12] we present a technique that derives a transformation language from a given base language. Such a transformation language reuses larger parts of the concrete syntax of the base language and enriches it by patterns and control structures for transformations. We have successfully applied this engine on several UML sublanguages and DSMLs.

11.13 State Based Modeling (Automata)

Today we see that many computer science theories are based on state machines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using state machines for modeling systems.

As an aside, we believe that a sound and precise integration of the digital theory (automata) of computer science with control theory (calculus) used by almost all other engineering and science disciplines is one of the most interesting challenges that we experience at the moment. Cyber-Physical Systems urgently require such an integrated theory.

Our contributions to state based modeling can currently be split into three parts:

- understanding how to model object-oriented and distributed software using state machines resp. Statecharts,
- understanding refinement and composition on state machines, and
- applying state machines for modeling of systems.

State Machines as Semantics for Object-Oriented Distributed Software

A practically useable language for state based modeling is quite different from a pure theory, because a concrete modeling notation, for example, allows us to denote finitely many (typically very few) states while the implementation normally has an infinite state space.

In early papers like [GKR96], we have discussed how a system model can describe object-oriented systems. Built on this experience, a complete semantic model has been created for object-oriented systems in [BCR07b]. Objects, inheritance, states, method calls, stack, distribution, time as well as synchronous and asynchronous communication are completely defined and encoded into state machines. The theory is, therefore, suitable as semantic model for any kind of discrete systems. [BCGR09b] describes a condensed version of this system model and [BCGR09a] discusses design decisions, how to use the system model for denotational semantics, and taming the complexity of the system model.

Refinement and Refactoring of Statemachines

Starting with [PR94], we want to know, how to use state machines to describe abstract behavior of superclasses and refine it in subclasses. While [PR94] was rather informal, we have formalized the refinement relation in [RK96] by mapping a state machine to a set of possible component behaviors based on Focus's streams. In the Ph.D. thesis [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97], where a feature is a sub-automaton that adapts the original behavior in a refining form, precisely clarifying where feature interaction is allowed or harmful.

It became apparent that a state machine either serves as an implementation, where the described behavior is partial and can only be extended but not adapted, or that a state machine describes a specification, where the behavior is constrained to a possible, underspecified set of reactions, promised to the external users of a state machine. Here, refinement always means reduction of underspecification, telling more behavioral details to the external user. This is constructively achieved, e.g., by removing transitions that have alternatives or adding new behavior (transitions), if previously no transition was given at all.

Specification languages are particularly strong, if only explicitly given statements and no implicit additional assumptions hold (such as: implicit ignoring of messages, if they cannot be processed by a transition). See [Rum96,

Rum16] for details. The concept of chaos completion should be used to define semantics of incomplete state machines. This is much better for behavioral refinements than the concept of ignoring messages or error handling in cases where no explicit transition is given. The main disadvantage of "implicit ignoring" is that you never know whether the specifier intended this as desired behavior or just did not care (which is a big difference when we want to refine the specifier´s model!).

Our State Machine Formalism: I/O^ω Automata

[Rum96] describes an I/O^ω-automaton as $(S, M_{in}, M_{out}, \delta, I)$ consisting of:

- states S
- input messages M_{in}
- output messages M_{out}
- transition relation $\delta \subseteq S \times M_{in} \times S \times M_{out}^\omega$
- initial states I

where $M_{out}^\omega = M_{out}^* \cup M_{out}^\infty$ is the set of all finite and infinite words over M_{out}.

Transition relation δ is nondeterministic and incomplete. Each transition has one single input message from M_{in} but an arbitrary long sequence of output messages from M_{out}. Nondeterminism is handled as underspecification allowing the implementation (or the developer) to choose. Incompleteness is also understood as underspecification allowing arbitrary (chaotic) behavior, assuming that a later implementation or code generator will choose a meaningful implementation, but a specifier does not have to decide upfront. Fairness of choice for transitions is not assumed (but possible), as it is counterproductive to refinement by deciding on one alternative during the implementation process.

Most interestingly, describing transitions in δ with input and corresponding output leads to a much more abstract form of state machines, which can actually be used in the modeling process. First there are no (explicit) intermediate states necessary that would distribute a sequence of output messages in individual transitions (which is the case in classic Lynch/Tuttle I/O-automata, where a transition has exactly one input or output message). Second our I/O^ω automata preserve the causal relation between input and output on the transitions (whereas I/O automata distribute this over many transitions). We believe I/O^ω automata are therefore suited as a human modeling language and are thus used in a syntactically enriched, comfortable form as Statecharts in [Rum16].

Composition of State Machines

One state machine describes one component. In a distributed system, many state machines are necessary to describe collaborating components. The overall behavior of the component collaboration must then be derivable from the

knowledge about the form of composition (architecture describing communication channels) and the specified behavior (state machines) of the components. [GR95] describes how timed state machines are composed.

This technique is embedded into the composition and behavioral specifications concepts of Focus using streams and state machines in a nice overview article [BR07]. Most important, refinement of a component behavior by definition leads to a refinement of the composed system. This is a very important property, which is unfortunately not present in many other approaches, where system integration is a nightmare when components evolve.

Unfortunately, the untimed, event driven version of state machines that is very well suited for refinement and abstract specification has no composition in general. Further investigation is necessary.

Usage of Automata-based Specification

All our knowledge about state machines is being embedded in the model-based development method for the UML in [Rum16]. Furthermore it is applied for robots in MontiArcAutomaton [RRW14], a modeling language combining state machines and an architectural description language in [THR+13] as well as in building management systems in [FLP+11].

11.14 Modelling Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) are software controlled, collaborating physical machines.

In [KRS12] we discuss that this new term arises mainly due to the increased ability of computers to sense their environment and to interact with their contexts in various ways. As consequence, CPS are usually designed as distributed networks of interacting nodes and physical devices (machines) that carry out certain tasks. Often some of these devices are mobile (robots or autonomous cars, but also smartphones, airplanes and drones) and interaction with humans is essential. CPS are therefore complex in several dimensions: they embody characteristics of physical, networked, computational-intensive, and of human-interactive systems. Furthermore, they typically cannot be developed as monolithic systems, but need to be developed as open, composable, evolving, and scalable architectures.

Nowadays, CPS are found in many domains, including aerospace, automotive, energy, healthcare, manufacturing, and robotics. Many distributed CPS use a virtual communication network mapped to the Internet or telecommunication infrastructure.

At the heart of CPS engineering has to deal with the challenge that control theory, built on integration and differentiation calculus used by almost any engineering discipline, and the digital theory of state machines are not very

well integrated and thus do not allow us to describe CPS in an integrated way. Many attempts have been made, but a good standard yet has to emerge.

The complexity and heterogeneity of CPS introduces a wide conceptual gap between problem and solution domains. Model-driven engineering of such systems can decrease this gap by using models as abstractions and thus facilitate a more efficient development of robust CPS.

For the aviation domain, a modeling language [ZPK+11] allows to specify flight conditions including trajectories, status of the airplanes and their devices, weather conditions, and pilot capabilities. This modeling language allows EuroControl to operationalize correct flight behavior as well as specify and detect "interesting events". As long term interest, we intensively do research on how to improve the engineering for distributed automotive systems as well. For example [HRR12] outlines our proposal for an architecture centric development approach, which we apply to robotics in [RRW13, RRW14].

CPS, automotive, robots and energy management is discussed separately in Sections 11.16, 11.17, 11.18 and 11.19.

11.15 Applications in Cloud Computing and Data-Intensive Systems

As web-based application and service architectures are continuing to grow in complexity, criticality and into new application domains, their development, integration, evolution, operation and migration poses ever more and ever larger challenges to Software Engineering. In [KRR14] we discuss in detail the paradigm of cloud computing that is arising out of a convergence of existing and new technologies. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development.

Cloud-based systems pose a multitude of different challenges. The demand for seamless scalability with system load leads to highly distributed infrastructures and software architectures that can grow and shrink at runtime. The lack of standards, complemented by the variety of proprietary infrastructure and service providers, leads to a high degree of technological heterogeneity. High availability and connectivity with a multitude of clients leads to complex evolution and maintenance processes. These challenges come coupled with distinct requirements posed by the individual application domain. Application classes like Internet of Things as described in [HHK+14], Cyber-Physical Systems described in [KRS12], big data, app and service ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools. For example in [HHK+14, HHK+15b] we discuss how to handle privacy in the cloud.

In our research we tackle these challenges by perusing a model-based, generative approach. The core of this approach are several modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. UML/P models, and class diagrams in particular, describe several other aspects of the system, such as its domain and data models, its interfaces and interactions, and its monitoring and scaling capabilities. Among other tools, code generators most prominently take these models as input and generate application-specific frameworks that implement big parts of the system's technical aspects and provide technology-agnostic, ease-to-use interfaces for the cloud-based application's actual business logic.

We have applied these technologies to various cloud systems, cars, buildings, smart phones and smart pads and various other kinds of sensors. We built a rather successful and technologically sound framework for web based software portals [HKR12] that we offer under sselab.de for general use. Another set of cloud systems helps to deal with energy management and is described in [FPPR12, KPR12]. It continuously monitors building operation systems to derive operational data and compare these to the building specification. We use cloud technologies to maintain data, dynamically execute calculations and host management services enabling reduction of building energy costs.

11.16 Modelling for Energy Management

In the past years it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Today housing, offices, shops and other buildings are responsible for 40% of the overall energy consumption and 36% of the EU CO2 emissions.

The EU 2020 Climate and Energy package sets three key objectives: (1) 20% reduction in EU greenhouse gas emissions, (2) Raising the share of EU energy consumption produced from renewable resources to 20%, and (3) 20% improvement in the EU's energy efficiency compared to 1990.

Thus the management of energy in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Improvements in this field can be found at multiple scales: smart grids, demand-response systems, energy efficient neighbourhoods, energy efficient buildings, user awareness, micro- and mini renewable energy sources, to name a few. While there has been a lot of research on increasing the efficiency of single devices and also of single buildings there is a huge need for ICT based approaches within this field to integrate and combine the heterogeneous approaches. By such an integrated solution the efficiency can be raised even more.

Within several research projects we developed methodologies and solutions for integrating heterogeneous systems at different scales. Starting with single buildings we developed in collaboration with the Synavision GmbH and the Braunschweig University of Technology the ICT tool Energy Navigator.

The Energy Navigator´s Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already during the design phase. Resulting from a lack of process integration the AFS can close the loop between modelling the structure and behavior of the building and its facilities, measuring operational data from sensors, matching model and operational data during analysis and reporting of the results. The results can be reused to adapt the model or to find faults in the implementation.

Within the Energy Navigator a DSL is used to enable the domain expert to express his specific domain knowledge via first class language concepts. These concepts include rules, functions, characteristics, metrics, time routines and states. Proposed by the DIN EN ISO 16484 a state based approach should be used to describe the functional behavior of facilities. We adapted the well known concept of state machines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12] in a screenshot in Fig. 11.7, which is taken from a commercial tool developed by the Synavision GmbH.

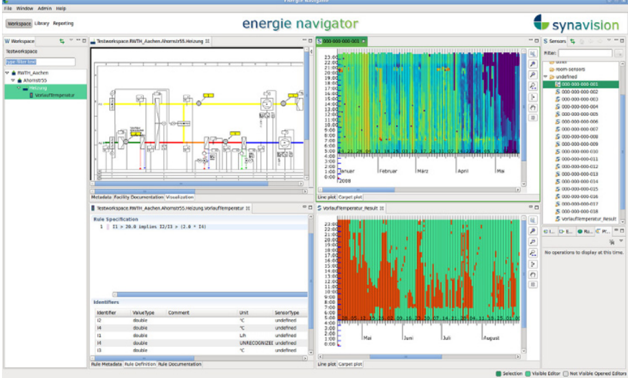


Fig. 11.7. Application of models to a building energy management system

Moving up the scale we investigated several existing approaches for energy efficient neighbourhoods that aim at moving from a local, building specific optimum to a more global optimum. By efficiently using results of simulation and optimization calculated optimal set points for local consumption and generation can be utilized. Therefore, information from several hetero-

geneous data sources, such as single sensor data, structural data, data on installed devices, geospatial data or weather data is needed. Based on existing approaches we developed a Neighbourhood Information Model that follows a meta-model based approached and utilized code generation techniques to automatically generate adapters between heterogeneous data models. Following this approach we are able to fully integrate the data sources on an abstract level and are still extensible at runtime.

11.17 Modelling Robotics

We consider modern robotics as a special field of Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and real world challenges. The engineering of robotics applications requires composition and interaction of many software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications.

Our research in model-driven Software Engineering for robotics on the one hand focuses on software architectures to structure reusable units of behavior. On the other hand, we concentrate on modeling languages (DSLs) for robotic product assembly tasks in industrial contexts as well as planned and unplanned logistic tasks. We apply this to indoor robots interacting with humans as well as to industrial robots as well as to autonomously driving cars.

Modeling Robotic Application Architectures and Behavior

Describing a robot's software architecture and its behavior in integrated models, yields many advantages to cope with this complexity: the models are platform independent, can be decomposed to be developed independently by experts of the respective fields, are highly reusable and may be subjected to formal analysis.

In [RRW12] we have introduced the architecture and behavior modeling language and framework MontiArcAutomaton which provides an integrated, platform independent structure and behavior modeling language family with an extensible code generation framework. MontiArcAutomaton's central concept is encapsulation and decomposition known from component and connector architecture description languages (ADLs). This concept applied to the modeling language, the code generation process and the target runtime to bridge the gap of platform specific and independent implementations along well designed interfaces. This facilitates the reuse of robot applications and makes their development more efficient.

MontiArcAutomaton extends the ADL MontiArc and integrates various component behavior modeling languages implemented using MontiCore.

The integration of automata and tables to model component behavior are described in [RRW13]. The integration capabilities of MontiArc have been extended and generalized in [RRRW15]. If interested, the MontiArcAutomaton website[4] provides further information on the MontiArcAutomaton framework.

In several projects, we modeled logistics services with Festo Robotino Robots, ROS, and Python.

LightRocks: Modeling Robotic Assembly Tasks

The importance of flexible automatized manufacturing grows continuously as products become increasingly individualized. Flexible assembly processes with compliant robot arms are still hard to be developed due to many uncertainties caused—among others—by object tolerances, position uncertainties and tolerances from external and internal sensors. Thus, only domain experts are able to program such compliant robot arms. The reusability of these programs depends on each individual expert and tools allowing to reuse and the compose models at different levels of detail are missing.

In cooperation with the DLR Institute on Robotics and Mechatronics we have introduced the LightRocks (Light Weight Robot Coding for Skills) framework in [THR+13] which allows robotics experts and laymen to model robotic assembly tasks on different levels of abstraction, namely: assembly tasks, skills, and elemental actions. Robotics experts provide a domain model of the assembly environment and elemental actions which reference this model. Factory floor workers combine these to skills and task to implement assembly processes provided by experts. This allows a separation of concerns, increases reuse and enables flexible production.

The framework is implemented based on MontiCore language profiles for UML/P Statecharts and UML/P class diagrams, which allows to reuse much of the UML/P framework for modeling, model validation, code generation, and editor generation.

11.18 Automotive Software

Development of software for automotive systems has become increasingly complex in the past years. Sophisticated driver assistance, infotainment and car2X-communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. While we are carrying out in numerous projects in the automotive domain, here we concentrate on three aspects: autonomic driving, modeling of functional and logical architectures and on variability. To understand all these features we in

[4] see www.monticore.de

[GRJA12] describe a requirements management that connects with features in all phases of the development process, helps to handle complex development tasks and thus stabilizes the development of automotive systems.

Modeling logical architecture: function nets

The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. Here, feature views modeled as a function net are used to implement the mapping between feature-related requirements and the complete logical architecture of a car.

Variability of car software

Automotive functions that may be derived from a feature view are often developed in Matlab/Simulink. As variability needs also to be handled in development artifacts, we extended Matlab/Simulink with delta modeling techniques (see also Section 11.9). A core Simulink model represents the base variant that is transformed to another variant by applying deltas to it. A delta contains modifications that add, remove or modify existing model elements. This way, features of an automotive system may be developed modularly without mixing up variability and functionality in development artifacts [HKM+13]. New delta models that derive new variants may be added bottom-up without the need for a fully elaborated feature model.

In practice, product lines often origin from a single variant that is copied and altered to derive a new variant. In [HRRW12], we provide means to extract a well defined Software Product Line from a set of copy and paste variants. This way, further variant development is alleviated, as new variants directly reuse common elements of the product line.

[RSW+15] describes an approach to use logical and model checking techniques to identify commonalities and differences of two Simulink models describing the same control device in different variants and thus allows to understand incompatibilities.

11.19 Autonomic Driving and Driver Intelligence

From the viewpoint of Software Engineering, intelligent driver assistance and, in particular, autonomic driving is an interesting and demanding challenge because it includes the development of complex software embedded within a distributed, life-critical system (car) and the connection of heterogeneous, autonomic mobile devices (other cars, infrastructure, etc.) in one big distributed system.

We are involved in a number of projects with major European car manufacturers in which we transfer modern software development techniques to

the car domain. This transfer is necessary as, with its increasing complexity, software becomes a demanding driver of the overall systems development process and not just an add-on.

In the Carolo project, we built Caroline—a completely autonomous car - and participated in the Darpa Urban Challenge, where our car was driving autonomously in an urban area for hours. We successfully achieved the best place as newcomers (and best non-Americans). This resulted from a number of facts, including the rigorous application of agile development methods, such as XP and Scrum and a simulation for driving scenarios. In [BR12b] we describe the process driven by story cards as a form of use cases, a continuously integrated and running software up to a rigorous test, and simulation infrastructure, called Hesperia.

Fig. 11.8. Autonomous car “Caroline” finishing the Urban Grand Challenge in 2007

In particular, we have developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation (not only visualization!) of the car within its surrounding: the city, pedestrians and especially other cars [BBR07]. Our simulator is capable of running automatic back-to-back tests on the complete software system with no real hardware involved by producing sensor input from the simulation and acting according to the steering output of the autonomic driving software. Every night and, when necessary for every version change, the tests are automatically executed.

This technique allows us a dramatic speedup in development and evolution of autonomous car functionality, and thus, enables us to develop software in an agile way [BR12a]. We have successfully shown that agile development of high-quality software is possible and very effective in the automotive domain. However, it remains a challenge to combine this innovative modern way of agile, iterative systems development with the current devel-

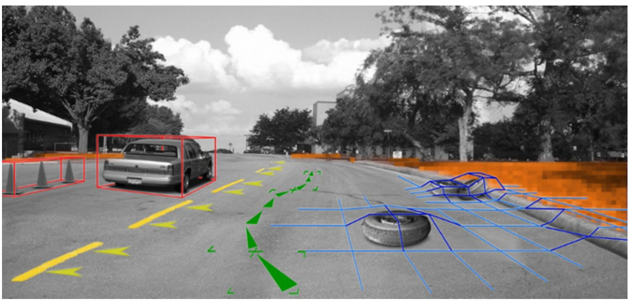


Fig. 11.9. How Caroline sees and interprets the world

opment standards, such as ISO 26262, in order to allow the OEMs to benefit both from efficiency and quality on the one hand and legal issues on the other hand.

In [MMR10] we gave an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions.

As tooling infrastructure, we mainly used an IDE such as Eclipse and, in particular the SSElab storage, versioning and management services [HKR12]. Without those agile development would not have been possible.

References

[AG83] A. Albrecht and J. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction. *IEEE Transactions on Software Engineering*, 9:639–648, June 1983.

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375. Springer, 2006.

[AM01] K. Auer and R. Miller. *Extreme Programming Applied. Playing to win*. Addison-Wesley, 2001.

[AMB+04] A. Abran, J. Moore, P. Bourque, R. Dupuis, and L. Tripp. SWEBOK. Guide to the Software Engineering Body of Knowledge, 2004 Version, 2004.

[Ast02] D. Astels. Refactoring With UML. In *Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002, May 26-30, Alghero, Italy*, pages 67–70, 2002.

[Bal98] H. Balzert. *Lehrbuch der Software-Technik. Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag. Heidelberg, 1998.

[Bal00] H. Balzert. *Lehrbuch der Software-Technik. Software-Entwicklung, 2. Aufl.* Spektrum Akademischer Verlag. Heidelberg, 2000.

[BB00] F. Basanieri and A. Bertolino. A Practical Approach to UML-Based Derivation of Integration Tests. In *Proc. 4th Intl. Software Quality Week Europe (QWE'2000), Brussels*. QWE, 2000.

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP, Vol 1: The Wide Spectrum Language CIP-L.* LNCS 183. Springer-Verlag, 1985.

[BBK91] G. Bernot, M. Bidoit, and T. Knapik. Observational Approaches in Algebraic Specifications: a Comparative Study. Technical report Liens-91-6, Laboratoire d'Informatique de l'Ecole Normale Superieure, 1991.

© Springer International Publishing AG 2017

B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9

[BBR07] C. Basarke, C. Berger, and B. Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BBWL01] M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf. Extreme Modeling. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, pages 175–189. Addison-Wesley, 2001.

[BCGR09a] M. Broy, M. Cengarle, H. Grönniger, and B. Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] M. Broy, M. Cengarle, H. Grönniger, and B. Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] M. Broy, M. Cengarle, and B. Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] M. Broy, M. Cengarle, and B. Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BDA+99] P. Bourque, R. Dupuis, A. Abran, J. Moore, and L. Tripp. The Guide to the Software Engineering Body of Knowledge. *IEEE Software*, 16:35–44, 1999.

[Bec01] K. Beck. Aim, Fire (Column on the Test-First Approach). *IEEE Software*, 18(5):87–89, 2001.

[Bec04] K. Beck. *Extreme Programming Explained. Embrace Change*. Addison-Wesley, 2004.

[BEH+87] F. L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer, 1987.

[Bei95] B. Beizer. *Black Box Testing*. John Wiley & Sons, New York, 1995.

[Bei04] B. Beizer. *Software Testing Techniques*. Dreamtech Press 2nd Edition, 2004.

[BF00] K. Beck and M. Fowler. *Planning Extreme Programming*. Addison-Wesley, 2000.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0, Part 1. Technical Report TUM-I9312, Technische Universität München, 1993.

[BG98] K. Beck and E. Gamma. Test-Infected: Programmers Love Writing Tests. *JavaReport*, July 1998.

[BG99] K. Beck and E. Gamma. JUnit: A Cook's Tour. *JavaReport*, August 1999.

[BGH+97] R. Breu, R. Grosu, C. Hofmann, F. Huber, I. Krüger, B. Rumpe, M. Schmidt, and W. Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHH+97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 – Object Oriented Programming. 11th European Conference, Proceedings*. Springer-Verlag, LNCS 1241, 1997.

[BHP+98] M. Broy, F. Huber, B. Paech, B. Rumpe, and K. Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BHW95] M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and Abstractor Specifications. *Science of Computer Programming*, 25(2):149–186, 1995.

[Bin94] R. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, 37(9):87–101, 1994.

[Bin99] R. Binder. *Testing Object-Oriented Systems. Models, Patterns, and Tools*. Addison-Wesley, 1999.

[BKPS04] G. Böckle, P. Knauber, K. Pohl, and K. Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt.verlag, 2004.

[BL01] L. Briand and Y. Labiche. A UML-based Approach to System Testing. In M. Gogolla and C. Kobryn, editors, *≪UML≫2001 – The Unified Modeling Language, 4th Intl. Conference*, pages 194–208, LNCS 2185. Springer, 2001.

[BL02] L. Briand and Y. Labiche. A UML-based Approach to System Testing. Technical report SCE-01-01, Carleton University, 2002.

[BLP01] D. Björklund, J. Lilius, and I. Porres. Towards Efficient Code Synthesis from Statecharts. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods. Workshop of the pUML-Group. October 1st, Toronto, Canada*, pages 29–41, LNI P–7. GI-Edition, Bonn, 2001.

[BMJ01] L. Bousquet, H. Martin, and J.-M. Jezequel. Conformance Testing from UML Specifications. Experience Report. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods. Workshop of the pUML-Group. October 1st, Toronto, Canada*, pages 43–55, LNI P–7. GI-Edition, Bonn, 2001.

[Boe81] B. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, 1981.

[Boe02] B. Boehm. Get Ready for Agile Methods with Care. *Computer*, 35(1):64–69, 2002.

[Bog99] M. Boger. *Java in verteilten Systemen. Nebenläufigkeit, Verteilung, Persistenz*. dpunkt.verlag Heidelberg, 1999.

[BPR04] J. Botaschanjan, M. Pister, and B. Rumpe. Testing Agile Requirements Models. *Journal of Zhejiang University*, 5(5), 2004.

[BR07] M. Broy and B. Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR11] M. Backschat and B. Rücker. *Enterprise JavaBeans und JPA: Grundlagen - Konzepte - Praxis zu EJB 3.1 und JPA 2.0, 3. Auflage*. Spektrum Akademischer Verlag, 2011.

[BR12a] C. Berger and B. Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] C. Berger and B. Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[Bra84] W. Brauer. *Automatentheorie: eine Einführung in die Technik endlicher Automaten*. Teubner, 1984.

[Bro98] M. Broy. *Informatik. Eine grundlegende Einführung. Band 2. Systemstrukturen und Theoretische Informatik. 2. Auflage*. Springer Verlag, 1998.

[BS01a] M. Boger and T. Sturm. Tool-support for Model-Driven Software Engineering. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods. Workshop of the pUML-Group. October 1st, Toronto, Canada*, pages 308–318, LNI P–7. GI-Edition, Bonn, 2001.

[BS01b] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[BvW98] R. Back and J. von Wright. *Refinement Calculus - A Systematic Introduction*. Springer, 1998.

[BW82] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer-Verlag, Berlin, 1982.

[BW02a] A. Brucker and B. Wolff. A Proposal for a Formal OCL Semantics in Isabelle/HOL. In *TPHOLs 2002*, LNCS. Springer-Verlag, Berlin, 2002.

[BW02b] A. Brucker and B. Wolff. HOL-OCL Experiences, Consequences and Design Choices. In J-M. Jézéquel and H. Hußmann, editors, *≪UML≫2002 – The Unified Modeling Language: Model Engineering, Concepts and Tools, 5th Intl. Conference*. Springer, LNCS, 2002.

[BW06] T. Baar and J. Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *Proc. of Sixth International Andrei Ershov Memorial Conference, Perspectives of System Informatics (PSI)*, Lecture Notes in Computer Science, pages 84–97, 2006.

[CBCR15] T. Clark, M. van den Brand, B. Combemale, and B. Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCD02] R. Cavarra, C. Crichton, and J. Davies. Using UML for Automatic Test Generation. In *International Symposium on Software Testing and Analysis ISSTA*. Springer-Verlag, 2002.

[CCF+15] B. H. C. Cheng, B. Combemale, R. B. France, J.-M. Jézéquel, and B. Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley Boston, 2000.

[CEG+14] B. Cheng, K. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. Müller, P. Pelliccione, A. Perini, N. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CFJ+16] B. Combemale, R. France, J.-M. JÃƒÂ©zÃƒÂ©quel, B. Rumpe, J. Steel, and D. Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CG98] L. Cardelli and A. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structures, FoSSaCS'98, LNCS 1378*, pages 140–155. Springer Verlag, 1998.

[CGR08] M. Cengarle, H. Grönniger, and B. Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] M. Cengarle, H. Grönniger, and B. Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[CH06] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CHS10] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[Coc06] A. Cockburn. *Agile Software Development, 2. Edition - The Cooperative Game*. Addison-Wesley, 2006.

[Con95] L. Constantine. *Constantine on Peopleware*. Yourdon Press, 1995.

[CW01] A. Cockburn and L. Williams. The Costs and Benefits of Pair Programming. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, pages 223–243. Addison-Wesley, 2001.

[DD08] T. Dyba and T. Dingsoyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833–859, 2008.

[Den91] E. Denert. *Software-Engineering*. Springer-Verlag, 1991.

[DH99] B. Demuth and H. Hußmann. Using UML/OCL Constraints for Relational Database Design. In R. France and B. Rumpe, editors, *≪UML'99≫ – The Unified Modeling Language: Beyond the Standard*, pages 598–613, LNCS 1723. Springer, 1999.

[DHL01] B. Demuth, H. Hußmann, and S. Loecher. OCL as a Specification Language for Business Rules in Data Base Applications. In M. Gogolla and C. Kobryn, editors, *≪UML≫2001 – The Unified Modeling Language, 4th Intl. Conference*, pages 104–117, LNCS 2185. Springer, 2001.

[Dib01] V. Dibartolo. FreeMarker: An open alternative to JSP. *JavaWorld, San Francisco, Calif. : Web Publ. Inc.*, 2001.

[Dij76] E. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[DN84] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(7):438–444, July 1984.

[Dob10] L. Dobrzanski. *UML Model Refactoring: Support for Maintenance of Executable UML Models*. Lambert Academic Publishing, 2010.

[Dou98] B. P. Douglass. *Real-Time UML. Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

[Dou99] B. P. Douglass. *Doing Hard Time. Developing Real-Time Systems with UML, Objekcts, Frameworks, and Patterns*. Addison-Wesley, 1999.

[DW98] D. D'Souza and A. Wills. *Objects, Components and Frameworks with UML – the Catalysis Approach*. Addison-Wesley, 1998.

[Eck09] J. Eckstein. *Agile Softwareentwicklung mit verteilten Teams*. dpunkt.verlag, 2009.

[Eck11] J. Eckstein. *Agile Softwareentwicklung in großen Projekten: Teams, Prozesse und Technologien - Strategien für den Wandel im Unternehmen* . dpunkt.verlag, 2011.

[EEKR99] H. Ehrig, G. Engels, H. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EFLR99] A. Evans, R. France, K. Lano, and B. Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[EH00a] A. Elting and W. Huber. Vorgehensmodelle contra Extreme Programming. 2 teilig. *sw development. Magazin für Software-Entwicklung*, 1&2, 2000.

[EH00b] G. Engels and R. Heckel. Graph Transformation and Visual Modeling Techniques. *Bulletin of the European Association for Theoretical Computer Science*, 71, June 2000.

[EH01] A. Elting and W. Huber. Immer im Plan? Programmieren zwischen Chaos und Planwirtschaft. *c't.*, 2, 2001.

[EHHS00] G. Engels, J.-H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *≪UML≫2000 – The Unified Modeling Language, 3th Intl. Conference*, pages 323–337, LNCS 1939. Springer, 2000.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications I*. Springer Verlag, Berlin, 1985.

[FELR98a] R. France, A. Evans, K. Lano, and B. Rumpe. Developing the UML as a Formal Modelling Notation. In J. Bezivin and P.-A. Muller, editors, *The Unified Modeling Language. ≪UML≫'98 Beyond the Notation. Mulhouse. Proceedings.*, pages 336–348, LNCS 1618. Springer, 1998.

[FELR98b] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. *Computer Standards & Interfaces*, 19:325–334, 1998.

[FELR98c] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FG99] M. Fewster and D. Graham. *Software Test Automation. Effective Use of Test Execution Tools*. ACM Press, New York & Addison-Wesley, 1999.

[FGJM85] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, pages 52–66, 1985.

[FHFB09] J. Fields, S. Harvie, M. Fowler, and K. Beck. *Refactoring: Ruby Edition*. Addison-Wesley, 2009.

[FHNS02] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected State Machine Coverage for Software Testing. In *Proceedings of the International Symposium of Software Testing and Analysis. ISSTA'02*, New York, 2002. ACM Press.

[FHR08] F. Fieber, M. Huhn, and B. Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FK00] D. Fields and M. Kolb. *Web Development with Java Server Pages*. Manning Greenwich, 2000.

[FKP+10] N. M. Fisch, T. Kurpick, C. Pinkernell, S. Plesser, and B. Rumpe. The Energy Navigator - A Web based Platform for functional Quality Mangement in Buildings. In *Proceedings of the 10th International Conference for Enhanced Building Operations (ICEBO' 10)*, Kuwait City, Kuwait, October 2010.

[FLP+11] N. M. Fisch, M. Look, C. Pinkernell, S. Plesser, and B. Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FNTZ00] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Selected Papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 296–309, London, UK, 2000. Springer-Verlag.

[Fow99] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FPPR12] M. Norbert Fisch, C. Pinkernell, S. Plesser, and B. Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[FPR01] M. Fontoura, W. Pree, and B. Rumpe. *The UML Profile for Framework Architecture*. Addison-Wesley, 2001.

[FSJ99] M. Fayad, D. Schmidt, and R. Johnson. *Building Application Frameworks. Object Oriented Foundations of Framework Design*. John Wiley & Sons, 1999.

[GH99] A. Gargantini and C. Heitmeyer. Using Model-Checking to Generate Tests from Requirements Specifications. In *Proceedings of the 7th European Software Engineering Conference (7th ACM SIGSOFT Symposium on the Foundations of Software Engineering), FSE'99, Toulouse, France*. Springer Verlag, LNCS 1687, 1999.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[GHK+07] H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, and B. Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, L. Rothhardt, and B. Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification. Third Edition*. Addison-Wesley, 2005.

[GKPR08] H. Grönniger, H. Krahn, C. Pinkernell, and B. Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] R. Grosu, C. Klein, and B. Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] H. Grönniger, H. Krahn, B. Rumpe, and M. Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, pages 67–81, 2006.

[Gla01] H. Glazer. Dispelling the Process Myth: Having a Process Does Not Mean Sacrificing Agility or Creativity. *Cross Talk. Journal of Defense Software Engineering*, Nov. 2001:27–30, 2001.

[GLRR15] T. Greifenberg, M. Look, S. Roidl, and B. Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] R. Grosu and B. Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR99] J. Goguen and G. Rosu. Hiding more of Hidden Algebra. In *FM'99, LNCS 1708*, pages 1704–1719, 1999.

[GR10] H. Grönniger and B. Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems. (16th Monterey Workshop)*. Microsoft Research, Redmond, 2010.

[GR11] H. Grönniger and B. Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] T. Gülke, B. Rumpe, M. Jansen, and J. Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[Grø09] R. Grønmo. *Using Concrete Syntax in Graph-based Model Transformations*. PhD thesis, Dept. of Informatics, University of Oslo, 2009.

[Grö10] H. Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten*. Aachener Informatik-Berichte, Software Engineering, Band 4. Shaker Verlag, 2010.

[Gro15] Standish Group. CHAOS Report 2015, 2015.

[GRR10] H. Grönniger, D. Reiß, and B. Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[GS02] J. Grabowski and M. Schmitt. TTCN-3 – Eine Sprache für die Spezifikation und Implementierung von Testfällen. *at – Automatisierungstechnik*, 50(3):A5–A8, März 2002.

[GSMD03] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards Automating Source-Consistent UML Refactorings. In *Proceedings of the International Conference on UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, pages 144–158. Springer-Verlag, 2003.

[GWM+92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report CSL-92-03, Computer Science Laboratory, SRI, March 1992.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Han10] E. Hanser. *Agile Prozesse: Von XP über Scrum bis MAP*. eXamen.press. Springer-Verlag, 2010.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Programming*, 8:231–274, 1987.

[Har08] E. R. Harold. *Refactoring HTML: Improving the Design of Existing Web Applications*. Addison-Wesley, 2008.

[HBG01] M. Holcombe, K. Bogdanov, and M. Gheorghe. Functional Test Generation for Extreme Programming. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, May 2001.

[HDF00] H. Hußmann, B. Demuth, and F. Finger. Modular Architecture for a Toolset Supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *≪UML≫2000 – The Unified Modeling Language, 3th Intl. Conference*, pages 278–293, LNCS 1939. Springer, 2000.

[Hes01] W. Hesse. RUP: A Process Model for Working with UML? In K. Sia and T. Halpin, editors, *Unified Modeling Language: System Analysis, Design and Development Issues*, pages 61–74. Idea Group Publishing Hershey, 2001.

[HG97] D. Harel and E. Gery. Executable Object Modelling with Statecharts. In *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society Press, 1997.

[HH08] R. Höhn and S. Höppner. *Das V-Modell XT - Grundlagen, Methodik und Anwendungen*. eXamen.press. Springer-Verlag, 2008.

[HHJ+87] C. Hoare, I. Hayes, H. Jifeng, C. Morgan, A. Roscoe, J. Sanders, I. Sorensen, J. Spivey, and B. Suffin. Laws of Programming. *Communications of the ACM*, 30(8):672–686, 1987.

[HHK+13] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, and I. Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] M. Henze, L. Hermerschmidt, D. Kerpen, R. Häußling, B. Rumpe, and K. Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, I. Schaefer, and C. Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] M. Henze, L. Hermerschmidt, D. Kerpen, R. Häußling, B. Rumpe, and K. Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] A. Haber, C. Kolassa, P. Manhart, P. Mir Seyed Nazari, B. Rumpe, and I. Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11a] A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR+11b] A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. Towards a Family-based Analysis of Applicability Conditions in Architectural Delta Models. In *Variability for You Workshop VARY'11)*, IT University Technical Report Series TR-2011-144, pages 43–52, 2011.

[HKR12] C. Herrmann, T. Kurpick, and B. Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HL02] R. Hightower and N. Lesiecki. *Java Tools for Extreme Programming*. Wiley Computer Publishing New York, 2002.

[HMSNRW16] R. Heim, P. Mir Seyed Nazari, B. Rumpe, and A. Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, 2016.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[Hop81] G. Hopper. The first bug. *Annals of the History of Computing*, 3:285–286, 1981.

[HP02] G. Halmans and K. Pohl. Modellierung der Variabilität einer Software-Produktfamilie. In M. Glinz and G. Müller-Luschnat, editors, *Modellierung 2002*, pages 63–74. GI, 2002.

[HR00] D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel, 2000.

[HR04] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] F. Huber, A. Rausch, and B. Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR10] A. Haber, J. O. Ringert, and B. Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, volume 2010-01 of *Informatik-Bericht*, pages 13 – 22. fortiss GmbH, Germany, 2010.

[HRR+11] A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] A. Haber, J. O. Ringert, and B. Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] C. Hopp, H. Rendel, B. Rumpe, and F. Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, pages 181–192, 2012.

[HRS09] P. Hruschka, C. Rupp, and G. Starke. *Agility kompakt: Tipps für erfolgreiche Systementwicklung*. Spektrum Akademischer Verlag, 2009.

[HRW15] K. Hölldobler, B. Rumpe, and I. Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HSSS96] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus - A Tool for Distributed Systems Specification. In B. Jonsson and J. Parrow, editors, *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470. LNCS 1135, Springer Verlag, 1996.

[HT90] R. Hamlet and R. Taylor. Partition Testing does not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[HU90] J. Hopcroft and J. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1990.

[Huß97] H. Hußmann. *Formal Foundations for Software Engineering Methods*. LNCS 1322. Springer-Verlag, Berlin, 1997.

[ISO92] ISO/IEC. Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN), 1992.

[IT07a] ITU-T. *SDL combined with UML, Recommendation Z.109 (06/07)*. International Telecommunication Union, 2007.

[IT07b] ITU-T. *Specification and Description Language (SDL), Recommendation Z.100 (11/07)*. International Telecommunication Union, 2007.

[IT11] ITU-T. *Message Seqeuence Chart (MSC), Recommendation Z.120 (02/11)*. International Telecommunication Union, 2011.

[JAH00] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2000.

[JK05] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2005.

[Jon96] M. P. Jones. *An Introduction to Gofer*, 1996.

[JR01] C. Jacobi and B. Rumpe. Hierarchical XP. Improving XP for Large-Scale Projects in Analogy to Reorganization Processes. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, pages 83–102. Addison-Wesley, 2001.

[JUn11] JUnit. JUnit Testframework Homepage. http://www.junit.org/, 2011.

[KCM00] S. Kim, J. Clark, and J. McDermid. The Rigorous Generation of Java Mutation Operators Using HAZOP. In J.-C. Rault, editor, *Proceedings of the 12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*. Bd. 4, Paris, 2000.

[KCS05] A. Kalnins, E. Celms, and A. Sostaks. Model Transformation Approach Based on MOLA. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Workshop: Model Transformations in Practice (MTIP), Montego*. Springer, 2005.

[KER99] S. Kent, A. Evans, and B. Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[Ker04] J. Kerievsky. *Refactorings to Patterns*. Addison-Wesley, 2004.

[KFN93] C. Kaner, J. Falk, and H. Nguyen. *Testing Computer Software, 2nd Edition*. Thompson Computer Press, 1993.

[KKP+09] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopez, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97 – Object Oriented Programming, 11th European Conference, Jyväskylä, Finnland*, LNCS 1241. Springer Verlag, 1997.

[KLPR12] T. Kurpick, M. Look, C. Pinkernell, and B. Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] C. Klein, C. Prehofer, and B. Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] T. Kurpick, C. Pinkernell, and B. Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, editors, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] H. Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] C. Klein, B. Rumpe, and M. Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRP11] D. Kolovos, L. Rose, and R. Page. *The epsilon Book*. Available at: http://www.eclipse.org/gmt/epsilon/doc/book, Dec. 2011.

[KRR14] H. Krcmar, R. Reussner, and B. Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] S. Kowalewski, B. Rumpe, and A. Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, pages 113–116. VDI Verlag, 2012.

[Krü00] I. Krüger. *Distributed System Design with Message Sequence Charts*. Doktorarbeit, Technische Universität München, 2000.

[Kru03] P. Kruchten. *The Rational Unified Process. An Introduction, Third Edition*. Addison-Wesley, 2003.

[KRV06] H. Krahn, B. Rumpe, and S. Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] H. Krahn, B. Rumpe, and S. Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] H. Krahn, B. Rumpe, and S. Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] H. Krahn, B. Rumpe, and S. Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[Küb09] S. Kübeck. *Software-Sanierung: Weiterentwicklung, Testen und Refactoring bestehender Software*. mitp Verlag, 2009.

[KW02] D. König and H. Wegener. Geniestreich oder Mogelpackung. Extreme Programming: Pro und Contra. *iX Journal*, 1/2002:94–99, 2002.

[Leh07] J. Lehmbach. *Vorgehensmodelle im Spannungsfeld traditioneller, agiler und Open-Source-Softwareentwicklung*. ibidem-Verlag, 2007.

[LF02] J. Link and P. Fröhlich. *Unit Tests mit Java. Der Test-First-Ansatz*. dpunkt.verlag Heidelberg, 2002.

[Lig90] P. Liggesmeyer. *Modultest und Modulverifikation*. B.I. Wissenschaftsverlag Mannheim, 1990.

[LKAS09] E. Legros, F. Klar, C. Amelunxen, and A. Schürr. Generic and Reflective Graph Transformations for Checking and Enforcement of Modeling Guidelines. *Journal of Visual Languages and Computing*, 20(4):252–268, August 2009.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. 2nd Edition. Springer-Verlag, Berlin, 1987.

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *International Workshop on Intelligent Signal Processing (WISP)*. IEEE, 2001.

[LOO01] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, 44(10):39–41, 2001.

[LRSS10] T. Levendovszky, B. Rumpe, B. Schätz, and J. Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[LRW02] M. Lippert, S. Roock, and H. Wolf. *Software entwicklen mit Extreme Programming*. dpunkt.verlag, 2002.

[Lud02] J. Ludewig. Modelle im Software Engineering – eine Einführung und Kritik. In M. Glinz and G. Müller-Luschnat, editors, *Modellierung 2002*, pages 7–22. GI, 2002.

[Mar09] R. C. Martin. *Clean Code - Refactorings, Patterns, Testen und Techniken für sauberen Code*. mitp Verlag, 2009.

[MB03] F. Marschall and P. Braun. Model Transformations for the MDA with BOTL. Technical report, University of Twente, 2003.

[McL06] B. McLaughlin. *Java and XML, 3rd Edition*. O'Reilly, 2006.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1997.

[MFC01] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit Testing with Mock Objects. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, pages 287–301. Addison-Wesley, 2001.

[MH00] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly Press, 2000.

[MMPH99] P. Müller, J. Meyer, and A. Poetzsch-Heffter. Making Executable Interface Specifications More Expressive. In C. Cap, editor, *JIT '99 Java-Informations-Tage 1999*, Informatik Aktuell. Springer-Verlag, 1999.

[MMR10] T. Mens, J. Magee, and B. Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[Moo01] I. Moore. Jester - a JUnit test tester. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, May 2001.

[MRR10] S. Maoz, J. O. Ringert, and B. Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] S. Maoz, J. O. Ringert, and B. Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] S. Maoz, J. O. Ringert, and B. Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] S. Maoz, J. O. Ringert, and B. Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] S. Maoz, J. O. Ringert, and B. Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] S. Maoz, J. O. Ringert, and B. Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] S. Maoz, J. O. Ringert, and B. Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] S. Maoz, J. O. Ringert, and B. Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[MT04] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30:126–139, 2004.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, 1997.

[Mye79] G. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, 1979.

[Mye01] G. Myers. *Methodisches Testen von Programmen*. Oldenbourg, München, 7.te Auflage, 2001.

[Nag79] M. Nagl. *Graph-Grammatiken: Theorie, Implementierung, Anwendungen*. Vieweg, Braunschweig, 1979.

[NM01] J. Newkirk and R. Martin. *Extreme Programming in Practice*. Addison-Wesley, 2001.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer Heidelberg, 2002.

[NS91] M. Nagl and A. Schürr. A Specification Environment for Graph Grammars. In *Proceedings of the 4th International Workshop on Graph-*

Grammars and Their Application to Computer Science, pages 599–609. Springer-Verlag, 1991.

[OB88] T. Ostrand and M. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, 1988.

[OH98] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 1998.

[OMG08] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Technical Report OMG Document formal/08-04-03, Object Management Group, April 2008.

[OMG10] OMG. OMG Unified Modeling Language: Infrastructure Specification, Superstructure Specification; formal/2010-05-03, formal/2010-05-05. Technical report, Object Management Group (OMG), May 2010.

[Opd92] W. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.d. thesis, University of Illinois at Urbana-Champaign, 1992.

[Öve00] G. Övergaard. Formal Specification of Object-Oriented Modelling Concepts. PhD Thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden, 2000.

[Pae00] B. Paech. *Aufgabenorientierte Softwareentwicklung. Integrierte Gestaltung von Unternehmen, Arbeit und Software*. Springer Verlag, 2000.

[Par90] H. Partsch. *Specification and Transformation of Programs. A Formal Approach to Software Development*. Monographs in CS. Springer-Verlag, Berlin, 1990.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 929, Springer-Verlag, 1994.

[Pau01] M. Paulk. Extreme Programming from a CMM Perspective. *IEEE Software*, 18(6):19–26, 2001.

[Pep84] P. Pepper. *Program Transformation and Programming Environments. Report on a Workshop directed by F. L. Bauer and H. Remus. NATO ASI Series F, Vol. 8*. Springer, 1984.

[PFR02] W. Pree, M. Fontoura, and B. Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[Pip02] J. Pipka. Refactoring in a Test First-World. In *Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002, May 26-30, Alghero, Italy*, pages 178–181, 2002.

[PJH$^+$01] S. Pickin, C. Jard, T. Heuillard, J.-M. Jezequel, and P. Desfray. A UML-integrated Test Description Language for Component Testing. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods. Workshop of the pUML-Group. October 1st, Toronto, Canada*, pages 208–223, LNI P–7. GI-Edition, Bonn, 2001.

[PKS02] M. Pol, T. Koomen, and A. Spilner. *Management und Optimierung des Testprozesses, 2te Auflage*. dpunkt.verlag, 2002.

[PLP01] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 12th IEEE Intl. Workshop on Rapid System Prototyping (RSP'01)*, pages 155–160. IEEE Computer, 2001.

[PP02] A. Pretschner and J. Philipps. Szenarien modellbasierten Testens. Technical report TUM-I0205, Technische Universität München, 2002.

[PR94] B. Paech and B. Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR97] J. Philipps and B. Rumpe. Refinement of Information Flow Architectures. In M. Hinchey, editor, *ICFEM'97*. IEEE CS Press, 1997.

[PR99] J. Philipps and B. Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] J. Philipps and B. Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] J. Philipps and B. Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP'97 – Object Oriented Programming, 11th European Conference, Jyväskylä, Finnland*, LNCS 1241. Springer Verlag, 1997.

[Pre00] C. Prehofer. Flexible Construction of Software Components: A Feature-Oriented Approach. Habilitation Thesis, Technische Universtität München, May 2000.

[PTLP98] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering. Technology and Practice*. SEI Series on Software Engineering. Addison-Wesley, 1998.

[PWC+95] M. Paulk, C. Weber, B. Curtis, M. Chrissis, et al. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995.

[PYvB96] A. Petrenk, N. Yevtushenko, and G. von Bochmann. Testing Deterministic Implementations from Nondeterministic FSM Specifications. In *Proceedings of the 9th International Workshop on Testing of Communicating Systems (IWTCS'96)*, pages 125–140, 1996.

[RBGW10] T. Rossner, C. Brandes, H. Götz, and M. Winter. *Basiswissen - Modellbasierter Test*. dpunkt.verlag, 2010.

[RDT95] T. Ramalingam, A. Das, and K. Thulasiraman. Fault Detection and Diagnosis Capabilities of Test Sequence Selection Methods Based on the FSM Model. *Computer Communications*, 18(2):113–122, 1995.

[Rei99] W. Reif. Formale Methoden für sicherheitskritische Software - Der KIV-Ansatz. *Informatik Forschung und Entwicklung*, 14(4):193–202, 1999.

[RFBLO01] D. Riehle, S. Fraleigh, D. Bucka-Lasse, and N. Omorogbe. The Architecture of a UML Virtual Machine. In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 327–341. ACM Press, 2001.

[RG02] M. Richters and M. Gogolla. OCL: Syntax, Semantics and Tools. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL*, pages 42–68, LNCS 2263. Springer Verlag, Berlin, 2002.

[Rin14] J. O. Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] B. Rumpe and C. Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] B. Rumpe, C. Klein, and M. Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RL04] S. Roock and M. Lippert. *Refactorings in großen Softwareprojekten. Komplexe Restrukturierung erfolgreich druchführen*. dpunkt.verlag, 2004.

[RLNS00] K. Rustan, M. Leino, G. Nelson, and J. Saxe. ESC/Java user's manual. Technical Note 2000-02, Compaq Systems Research Center, Palo Alto, CA, 2000.

[Roz99] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1999.

[RRRW15] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW12] J. O. Ringert, B. Rumpe, and A. Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] J. O. Ringert, B. Rumpe, and A. Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] J. O. Ringert, B. Rumpe, and A. Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RS01] B. Rumpe and A. Schröder. Quantitative Untersuchung des Extreme Programming Prozesses. Technical report TUM-I0110 and ViSEK/006D, Technische Universität München und Virtuelles Software Engineering Kompetenzzentrum, 2001.

[RS02] B. Rumpe and A. Schröder. Quantitative Survey on Extreme Programming Projects. In *Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002, May 26-30, Alghero, Italy*, pages 43–46, 2002.

[RSW+15] B. Rumpe, C. Schulze, M. von Wenckstern, J. O. Ringert, and P. Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[RT98] B. Rumpe and V. Thurner. Refining Business Processes. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Seventh OOPSLA Workshop on Precise Behavioral Semantics*, I9820. Technische Universität München, June 1998.

[Rum96] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum98] B. Rumpe. A Note on Semantics (with an Emphasis on UML). In *Second ECOOP Workshop on Precise Behavioral Semantics*. Technische Universität München, TUM-I9813, 1998.

[Rum01] B. Rumpe. Extreme Programming - Back to Basics? In G. Engels, A. Oberweis, and A. Zündorf, editors, *Proceedings of Modellierung 2001, 28.-30.3.2001 Bad Lippspringe*, pages Lecture Notes in Informatics, Band 1. GI-Edition, Bonn, 2001.

[Rum02] B. Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] B. Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] B. Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] B. Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] B. Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] B. Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML. Code Generation, Test Cases, Refactoring*. Springer International, 2017.

[RW11] B. Rumpe and I. Weisemöller. A Domain Specific Transformation Language. In *ME 2011 - Models and Evolution*, Wellington, New Zealand, 2011.

[RWH01] B. Reus, M. Wirsing, and R. Hennicker. A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Model. In *FASE 2001, ETAPS, Genova*, LNCS 2029, pages 300–316. Springer Verlag, 2001.

[Sax08] E. Sax. *Automatisiertes Testen Eingebetteter Systeme in der Automobilindustrie*. Carl Hanser Verlag, 2008.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF Eclipse Modeling Framework*. Addison-Wesley, 2. edition, 2008.

[Sch88] A. Schürr. Modellierung und Simulation komplexer Systeme mit PROGRES. In W. Ameling, editor, *Proc. 5. Symp. Simulationstechnik, Aachen, Germany*, volume 179 of *Informatik-Fachberichte*, pages 84–91, Heidelberg, 1988. Springer Verlag.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitätsverlag, Wiesbaden, 1991.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D)*. Springer, 1994.

[Sch04] B. Schätz. Mastering the Complexity of Embedded Systems - The AutoFocus Approach. In F. Kordon and M. Lemoine , editor, *Formal Techniques for Embedded Distributed Systems: From Requirements to Detailed Design*. Kluwer, 2004.

[Sch12] M. Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SD00] J. Siedersleben and E. Denert. Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur. *Informatik Spektrum*, 8/2000:247–257, 2000.

[SK04] S. Sendall and J. Küster. Taming model round-trip engineering. In *Proceedings of OOPSLA Workshop on Best Practices for Model-Driven Software Development*, 2004.

[Sne96] H. Sneed. Schätzung der Entwicklungskosten von objektorientierter Software. *Informatik-Spektrum*, 19:133–140, 1996.

[Som10] I. Sommerville. *Software Engineering, 9th Edition*. Addison-Wesley Longman, Amsterdam, 2010.

[Sou01] N. Soundarajan. Refactoring and Re-Reasoning. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, pages 303–319. Addison-Wesley, 2001.

[Spi88] J. Spivey. *Understanding Z*. Cambridge University Press, 1988.

[SPTJ01] G. Sunye, D. Pollet, Y. Le Traon, and J.-M. Jezequel. Refactoring UML Models. In M. Gogolla and C. Kobryn, editors, *≪UML≫2001 – The Unified Modeling Language, 4th Intl. Conference*, pages 134–148, LNCS 2185. Springer, 2001.

[SRVK10] J. Sprinkle, B. Rumpe, H. Vangheluwe, and G. Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[SSSH01] G. Succi, M. Stefanovic, M. Smith, and R. Huntrods. Design of an Experiment for Quantitative Assessment of Pair Programming Practices. In M. Marchesi and G. Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, May 2001.

[ST87] D. Sanella and A. Tarlecki. On Observational Equivalence and Algebraic Specification. *Journal of Computer and System Sciences*, pages 150–178, 1987.

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag Wien, 1973.

[Ste10] M. Steyer. *Agile Muster und Methoden: Agile Softwareentwicklung maßgeschneidert*. Entwickler.Press, 2010.

[SVEH07] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management, 2. Auflage*. dpunkt.verlag, 2007.

[SvVB02] T. Sturm, J. von Voss, and M. Boger. Generating Code for UML with Velocity Templates. In J-M. Jézéquel and H. Hußmann, editors, *≪UML≫2002 – The Unified Modeling Language: Model Engineering, Concepts and Tools, 5th Intl. Conference*. Springer, LNCS, 2002.

[SZ01] E. Sekerinski and R. Zurob. iState: A Statechart Translator. In M. Gogolla and C. Kobryn, editors, *≪UML≫2001 – The Unified Modeling Language, 4th Intl. Conference*, pages 376–390, LNCS 2185. Springer, 2001.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance*, pages 446–453, 2004.

[TB01] L. Tokuda and D. Batory. Evolving Object-Oriented Designs with Refactorings. *Journal of Automated Software Engineering*, 8:89–120, 2001.

[TDDN00] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A Meta-model for Language-Independent Refactoring. In *Proceedings ISPSE 2000*, IEEE, 2000.

[TFR02] D. Turk, R. France, and B. Rumpe. Limitations of Agile Software Processes. In *Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002, May 26-30, Alghero, Italy*, pages 43–46, 2002.

[THR+13] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[vdB94] M. von der Beeck. A Comparison of Statecharts Variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, volume LNCS 863, pages 128–148. Springer-Verlag, 1994.

[vdB01] M. von der Beeck. Formalization of UML-Statecharts. In M. Gogolla and C. Kobryn, editors, *≪UML≫2001 – The Unified Modeling Language, 4th Intl. Conference*, pages 406–421, LNCS 2185. Springer, 2001.

[Vig10] U. Vigenschow. *Testen von Software und Embedded Systems, 2. Auflage*. dpunkt.verlag, 2010.

[Viv01] F. Vivaldi. *Experimental Mathematics with Maple*. CRC Press, Boca Raton, Florida, 2001.

[vO01] D. von Oheimb. Hoare Logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[Voa95] J. Voas. Software Testability Measurement for Assertion Placement and Fault Localization. In M. Ducasse, editor, *AADEBUG, 2nd International Workshop on Automated and Algorithmic Debugging, Saint Malo, France*, pages 133–144. IRISA-CNRS, 1995.

[Völ11] S. Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[W3C00] W3C. Extensible Markup Language (XML) 1.0 (2nd edition 6 October 2000). http://www.w3.org/xml, 2000.

[Wak02] W. Wake. *Extreme Programming Explored*. Addison-Wesley, 2002.

[Wei12] I. Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[Wil01] A. Wills. Catalytic Modeling: UML meets XP. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods. Workshop of the pUML-Group. October 1st, Toronto, Canada*, pages 288–307, LNI P–7. GI-Edition, Bonn, 2001.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14:221–227, 1971.

[WKCJ00] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the Case for Pair Programming. *IEEE Software*, 17(4):19–25, 2000.

[WKS10] I. Weisemöller, F. Klar, and A. Schürr. Development of Tool Extensions with MOFLON. In H. Giese, G. Karsai, E. Lee, B. Rumpe, and

B. Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007*, LNCS 6100, pages 337–343. Springer Verlag, 2010.

[Wol99] S. Wolfram. *The MATHEMATICA Book*. Cambridge University Press, 1999.

[Woy08] R. Woywod. *Extreme Programming goes offshore - Der Einfluss der Kulturen auf den Projektmanagement-Prozess bei agilen Methoden*. Grin Verlag, 2008.

[ZPK+11] M. Zanin, D. Perez, D. Kolovos, R. Paige, K. Chatterjee, A. Horst, and B. Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.

[Zün96] A. Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme - Spezifikation, Implementierung und Verwendung*. PhD thesis, RWTH Aachen, 1996.

Index

{Hash}, 117
{ToStringVerbosity}, 118
{ToString}, 118
{equals}, 117
{location}, 246

abstract, 35
abstract factory, 119
abstraction, 270
abstraction relation, 268
acceptance test, 163, 274
action, 58
action condition, 58
activity bar
  sequence diagram, 67
agility, 25
API
  code generation, 86
artifact, 12
association, 35
  bidirectional, 106
  code generation, 106
  qualified, 110
association name, 35, 52
association role, 35
attribute, 35, 52
  derived, 100
  eager calculation, 102
  lazy calculation, 102

best practices, 12
bug-fixing costs
  XP, 16

Capability Maturity Model, 18
change of signature, 270
changing the data structure, 306
class, 35
  code generation, 116
class attribute
  object diagram, 51
CMM, 18
code generation, 75, 99
code generator, 75
Collection<X>, 43
common model ownership, 262
composition, 37
  code generation, 114
  object diagram, 54
composition link, 52
comprehension, 44
condition
  OCL, 40
  sequence diagram, 67
conformance tests, 169
constant, 35
constructive model, 75
constructive test model, 75
context
  of a condition, 40
  of preconditions/postconditions, 50
context condition
  transformation rule, 278
correctness, 163
  functional, 162
coverage metrics, 208

definedness, 279

© Springer International Publishing AG 2017
B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9

descriptive model, 75, 76
determinism, 279
development method, 12
do activity, 58
dummy, 177, 219
  code creation, 119
  code generation, 156
  simulation of time, 220
  with a memory, 222

enabledness, 58
entry action, 58
equivalence comparison, 178
error, 167
exists, 47
exit action, 58
expected test result, 168
Extreme Programming, 13

factory
  code creation, 119
failure, 164, 167
fault, 167
filter
  comprehension, 44
final, 35
final state, 57, 58
flatten
  OCL, 47
forall, 47
framework, 304

generator
  comprehension, 44

infix operator
  OCL, 42
inheritance, 35
inheritance relationship, 35
initial state, 57
instanceof, 41
instrumentation
  for tests, 173
interaction pattern, 200
interface, 35
interface implementation, 35
interpretation
  of a condition, 40
invariant, 40

JUnit, 165, 178, 181

keyword
  tag, 39

label, 7
life cycle, 57
  object, 37
link, 52
  object diagram, 51
list comprehension, 44
List<X>, 43
location, 246, 251
  in a sequence diagram, 245
logic
  lifting, 42

marker, 7
message
  sequence diagram, 67
method, 35
  code generation, 103
method specification, 40, 50
minimal loop coverage, 208
model
  closed, 270
  open, 270
model element, 39
model transformation, 265, 266, 276
  semantics, 267, 271
modeling standards, 262
modifier, 35
multiplicity, 35

navigation, 37
navigation direction, 35, 52
nondeterminism, 58
notion of observation, 272, 276

object, 52
  sequence diagram, 67
Object Constraint Language, 39
object name, 52
observation, 275, 276, 290
  external, 290
  refactoring, 272
observation invariance, 276
OCL, 39
omission, 167

oracle function, 179

pair programming, 19
parameterization
  code generation, 92
path coverage, 208
postcondition, 50, 58
  method, 40
precondition, 50, 58
  method, 40
prescriptive model, 76
primitive data type, 41
principle, 12
private, 35
protected, 35
prototypical object, 52, 54
public, 35

qualifier
  object diagram, 52
quality, 162
quality of the design, 263
quantifier, 47
query, 40

rapid prototyping, 72
readonly, 35
refactoring, 23, 257, 270, 290
  framework, 304
  rule template, 288
  singleton, 303
refactoring rule, 290
refactoring step, 290
refinement relation, 268
representation, 37
representation indicator
  ..., 37
rest run, 168
reusability, 74
robustness, 162, 163, 170
role name, 35, 52
roundtrip engineering, 110

scheduling
  with a sequence diagram, 240
schema variable, 257
  code generation, 94
script, 75
  code generation, 89
semantics
  code generation, 89, 91
sequence diagram, 66
set comprehension, 44
Set<X>, 43
side effect
  in tests, 224
side effect-freedom, 279
signature, 35
singleton
  in a test, 303
snapshot, 273
software development process, 12
source state, 58
state, 57
state coverage, 208
state invariant, 57
Statechart, 56
  oracle, 202
static, 35
stereotype, 39
  ≪trigger≫, 155
stimulus, 58
subclass, 35
substitution principle, 35
subtype, 35
success, 164
superclass, 35
superimposition method, 306
surprise, 167
SWEBOK, 11
system model, 272, 273
system run, 273

tag, 39
  {global}, 249
  {local}, 249
  {location}, 246
  {time}, 235
template, 75
termination, 279
test, 163
  refactoring, 273
  with an object diagram, 186
  with boundary value analysis, 193
  with method specification, 191
  with OCL constraint, 189
  with sequence diagram, 195
test case, 168

test coverage, 169
test data, 164, 168
test driver, 168, 177, 178
  code generation, 155
test model, 75
test object, 168
test pattern, 217, 218, 302
  class library, 230
  communication, 249
  concurrency, 238
  distribution and communication, 245, 251
  framework, 230
  object instantiation, 228
  scheduler, 238
  simulation of time, 232, 236
  singleton, 227
  thread, 237
  thread creation, 241
  timer, 237
test procedure, 168
test process, 218
test result, 168
test run
  sequence diagram, 223
test sequence, 195
test success, 168
test suite, 168, 183
test-first, 20
tests
  for equivalence classes, 193
timeline
  sequence diagram, 67
transformation, 12, 280
transformation rule
  code generation, 94
transition, 58
transition coverage, 208
trigger, 196, 200
type constructor
  OCL, 43
typeif, 41

UML virtual machine, 85
unit test, 163, 274

value
  tag, 39
verification, 301
  invariant, 279
view, 37

XP, 13