You are looking at Dirk Riehle's dissertation on Object-Oriented Framework Design (it starts on the next page). The full reference is:

Dirk Riehle. *Framework Design: A Role Modeling Approach.* Ph.D. Thesis, No. 13509. Zürich, Switzerland, ETH Zürich, 2000.

You can browse it on the web in HTML here:

- http://dirkriehle.com/computer-science/research/dissertation/index.html

The dissertation weighs in with more than 200 pages. An earlier summary is provided by the following much shorter OOPSLA 1998 paper:

Dirk Riehle and Thomas Gross. "Role Model Based Framework Design and Integration." In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '98). ACM Press, 1998. Page 117-133.

This paper can be accessed here:

- http://dirkriehle.com/computer-science/research/1998/oopsla-1998.html

(If you are a researcher, you may still want to read the dissertation as it contains crisper and more detailed definitions, examples, and validation.)

If you like this work, you may also like our work on ***Framework Development for Large Systems***

- http://dirkriehle.com/computer-science/research/1997/cacm-1997-frameworks.html

on ***Composite Design Patterns***

- http://dirkriehle.com/computer-science/research/1997/oopsla-1997.html

and on ***Design Pattern Density*** (TBP).

Dirk Riehle's publication list can be found at http://dirkriehle.com/publications.

Diss. ETH No. 13509

Framework Design

A Role Modeling Approach

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
DOKTOR DER TECHNISCHEN WISSENSCHAFTEN
(DOCTOR OF TECHNICAL SCIENCES)

Dirk Riehle
Dipl.-Inform., Universität Hamburg
born xxxx-xx-xx, citizen of xxxxxxxxxxx

Accepted on the recommendation of
Prof. Dr. Thomas R. Gross
Prof. Dr. Douglas C. Schmidt

2000

Abstract

Role modeling for framework design, as developed in this dissertation, makes designing, learning, and using object-oriented frameworks easier than possible with traditional class-based approaches.

Object-oriented frameworks promise higher productivity and shorter time-to-market for the development of object-oriented applications. These goals are achieved through design and code reuse. While many projects show that these promises can be met, failed projects also show that they are not always easy to reach. This dissertation addresses three pertinent technical problems of designing, learning, and using object-oriented frameworks: complexity of classes, complexity of object collaboration, and lack of clarity of requirements put upon use-clients of a framework.

Role modeling for framework design is an evolutionary extension of class-based modeling of frameworks. The method enhances class-based modeling with role modeling concepts. In this method, objects play roles that are described by role types. An object typically plays several roles, so that the class of an object composes several role types. Moreover, objects collaborate for several different purposes, each of which is called an object collaboration task. Such a task is described by a role model. A class model composes all relevant role models to describe how instances of its classes collaborate. Describing classes as compositions of role types and class models as compositions of role models reduces class and object collaboration complexity.

Going one step further, role modeling for framework design defines frameworks as explicit design and implementation artifacts with well-defined boundaries. A framework defines how to use it with the help of so-called free role types of free role models. A free role model provides free role types for roles that clients of a framework have to play to make proper use of the framework. Free role types are key to defining the requirements put upon use-clients of a framework. Only by acting according to free role types from a free role model may use-clients make use of framework objects. The dissertation shows how these concepts are used to design, use, and layer object-oriented frameworks.

Role modeling for framework design represents a significant improvement over current practice regarding the initially stated problems of framework design. This thesis is validated with the help of

three case studies that show how role modeling for framework design works in practice. Each of the case studies compares a traditional class-based framework design with a framework design based on role modeling. However, each case study does so from a different angle. The first case study on the Geo Object framework compares a traditional design with an enhanced role modeling design. The second case study on the KMU Desktop Tools framework shows how role modeling helps in the redesign of an existing framework and how the redesigned version compares to the old version. The third case study on the JHotDraw framework for drawing editors shows how an existing well-designed framework design can be described even better using role modeling. Finally, all three case studies reflect on the experiences made while carrying them out.

For its validation, the thesis is split up into nine sub-theses, each of which has a problem and an activity dimension. The addressed problems are class complexity, object collaboration complexity, and lack of clarity of requirements put upon use-clients. The activities are designing and redesigning a framework, learning a framework, and using a framework. For each problem/activity pair an argument is made based on the case studies. The overall validation of the thesis becomes the validation of all nine problem/activity pairs.

Role modeling for framework design combines the strengths of role modeling with those of class-based modeling while leaving out their weaknesses. It is therefore an evolutionary extension of current methods that preserves existing investments. Finally, role modeling for framework design is the first comprehensive method to make frameworks explicit design artifacts and to introduce modeling concepts for directly expressing their unique properties.

Kurzfassung

Der rollenmodellbasierte Entwurf von Frameworks, wie ihn die vorliegende Dissertation beschreibt, macht das Entwerfen, Verstehen, und Verwenden objektorientierter Frameworks einfacher, als es mit herkömmlichen klassenbasierten Entwurfsmethoden möglich ist.

Objektorientierte Frameworks dienen dazu, die Produktivität der Anwendungsentwicklung zu erhöhen und die Zeit zu reduzieren, bis eine Anwendung fertiggestellt und ausgeliefert werden kann. Man hofft, dies durch Entwurfs- und Codewiederverwendung zu erreichen. Viele Projekte zeigen, dass diese Ziele erreicht werden können. Allerdings zeigen gescheiterte Projekte ebenfalls, dass diese Ziele nicht immer einfach zu erreichen sind. Die vorliegende Dissertation behandelt drei zentrale Probleme des Entwurfs und der Verwendung von objektorientierten Frameworks: die Komplexität von Klassen, die Komplexität des Objektzusammenspiels („object collaboration“) und die fehlende Klarheit in den Anforderungen, die ein Framework an Klienten stellt, die es benutzen wollen.

Der Entwurf von Frameworks mittels Rollenmodellierung, wie ihn diese Dissertation entwickelt, ist eine evolutionäre Weiterentwicklung des klassenbasierten Entwurfs. Der Ansatz erweitert den traditionellen klassenbasierten Entwurf mit Konzepten der Rollenmodellierung. Beim vorgestellten Ansatz spielen Objekte Rollen, welche durch Rollentypen beschrieben werden. Normalerweise spielt ein Objekt mehrere Rollen, so dass die Klasse eines Objektes mehrere Rollentypen komponiert. Weiterhin dient das Zusammenspiel von Objekten üblicherweise mehreren unterschiedlichen Aufgaben, die als Objektzusammenspiel mit einer Aufgabe („object collaboration task“) bezeichnet werden. Solch ein Objektzusammenspiel mit einer Aufgabe wird durch ein Rollenmodell beschrieben. Ein Klassenmodell ist dann die Komposition aller relevanten Rollenmodelle; es beschreibt wie Exemplare der Klassen des Klassenmodells zusammenspielen. Die Beschreibung von Klassen als Kompositionen von Rollentypen und von Klassenmodellen als Kompositionen von Rollenmodellen reduziert die Komplexität von Klassen und die Komplexität des Zusammenspiels von Objekten.

In einem weiteren Schritt führt die vorgestellte Methode Frameworks als eigenständige Entwurfs- und Implementierungsartefakte ein, welche sich wohldefiniert gegen ihre Umgebung abgrenzen. Dabei

verwendet ein Framework sogenannte freie Rollentypen, um festzulegen, wie Klientenobjekte Rollen zu spielen haben, um das Framework korrekt zu nutzen. Der Einsatz von freien Rollentypen ist von zentraler Bedeutung, um die Anforderungen zu definieren, die ein Framework an seine Umgebung richtet: Klientenobjekte dürfen Framework-Objekte nur dann benutzen, wenn sie Rollen gemäss freier Rollentypen spielen. Die vorliegende Dissertation zeigt auf, wie diese Konzepte zum Entwurf, zur Verwendung, und zur Schichtenbildung von Frameworks verwendet werden.

Der Entwurf von Frameworks mittels Rollenmodellierung stellt eine signifikante Verbesserung der heute üblichen Praxis dar (in Bezug auf die oben genannten Probleme). Die Dissertation belegt diese These mithilfe von drei Fallstudien, welche illustrieren, wie die geschilderte Methode in der Praxis eingesetzt wird. Jede Fallstudie vergleicht einen herkömmlichen klassenbasierten Framework-Entwurf mit einem Entwurf auf Basis von Rollenmodellierung. Jede Fallstudie wählt dabei eine etwas andere Perspektive und ist durch einen anderen Hintergrund motiviert. Die erste Fallstudie beschreibt das Geo Object Framework, einmal als herkömmlichen klassenbasierten Entwurf und einmal als rollenmodellbasierten Entwurf. Die zweite Fallstudie beschreibt die Revision des Entwurfs des KMU Desktop Tools Framework unter Verwendung von Rollenmodellierung. Die Fallstudie vergleicht den ursprünglichen klassenbasierten Entwurf mit dem neuen rollenmodellbasierten Entwurf. Die dritte Fallstudie zeigt, wie Rollenmodellierung die Dokumentation eines existierenden Frameworks, des JHotDraw Frameworks für grafische Editoren, verbessern hilft. Alle drei Fallstudien berichten zudem über die Erfahrungen, die bei ihrer Ausführung gemacht wurden.

Für ihre Gesamtvalidierung wird die Dissertations-These in neun Einzelthesen aufgebrochen, die jeweils aus einer Problem- und einer Aktivitätsdimension bestehen. Die betrachteten Probleme sind die Komplexität von Klassen, die Komplexität des Objektzusammenspiels, und die fehlende Klarheit in den Anforderungen, die ein Framework an seine Klienten stellt. Die betrachteten Aktivitäten sind das Entwerfen und Revidieren des Entwurfs eines Frameworks, das Verstehen eines Frameworks und das Verwenden eines Frameworks. Jedes der resultierenden Problem/Aktivitäts-Paare wird einzeln betrachtet. Für jedes Paar wird begründet, warum Rollenmodellierung einen relevanten Fortschritt darstellt. Die Validierung der Dissertations-These insgesamt folgt aus der Validierung dieser neun Einzelthesen.

Der rollenmodellbasierte Entwurf von objektorientierten Frameworks kombiniert die Stärken der Rollenmodellierung mit den Stärken des herkömmlichen klassenbasierten Entwurfs und überkommt dabei viele seiner Schwächen. Der vorgestellte Modellierungsansatz ist damit eine evolutionäre Weiterentwicklung heutiger Methoden, welche existierende Investitionen wahrt. Weiterhin ist der rollenmodellbasierte Entwurf von Frameworks der erste umfängliche Modellierungsansatz, der Frameworks als eigenständige Entwurfsartefakte behandelt und Entwurfskonzepte bereitstellt, welche die spezifischen Eigenschaften von Frameworks ausdrücken helfen.

Table of Contents

Preface **xv**

1 Introduction **1**

1.1 Why object-oriented frameworks? 1

1.2 Problems with frameworks 2

1.3 Role modeling for framework design 3

1.4 Dissertation overview 4

1.5 Actors in this dissertation 6

2 Frameworks, Related Work, and Thesis Statement **7**

2.1 Overview of framework concepts 7

2.1.1 Object-oriented software architecture 7

2.1.2 Review of framework terminology 8

2.1.3 Problems with frameworks 9

2.2 Related work 10
2.2.1 Object-oriented design 10
2.2.2 Programming methods 11
2.2.3 Development methods 12
2.2.4 Role modeling concepts 13
2.2.5 Object-oriented frameworks 14
2.2.6 Review of related work 15

2.3 Thesis statement of dissertation 15
2.3.1 What is the scope of "design and documentation"? 15
2.3.2 What does "easier" mean? 17
2.3.3 Who is the subject? 18
2.3.4 Final version of the thesis 18

3 Role Modeling 21

3.1 Chapter overview 21

3.2 Object modeling fundamentals 23
3.2.1 Object and class (definition) 23
3.2.2 Value and value type (definition) 25
3.2.3 Figure class (example) 26
3.2.4 Relationships and relationship descriptions (definition) 27
3.2.5 Inheritance (definition) 30
3.2.6 Object collaboration and class model (definition) 31
3.2.7 Figure class model (example) 31

3.3 Role modeling extensions 33
3.3.1 Role and role type (definition) 33
3.3.2 Figure, Child, etc. (example) 35
3.3.3 Class (revised definition) 35
3.3.4 Choice of type specification mechanism 37
3.3.5 Object collaboration task and role model (definition) 37
3.3.6 Role constraint (definition) 38
3.3.7 Figure role models (examples) 39
3.3.8 Composing role models 42

3.3.9 Class model (revised definition) 42
3.3.10 Figure class model (revised example) 45
3.3.11 Design patterns in role modeling 49
3.3.12 Visual role model shorthands 51

3.4 Summary 52

4 Framework Design 53

4.1 Chapter overview 53

4.2 Framework design 54
4.2.1 Framework (definition) 54
4.2.2 Free role type (definition) 55
4.2.3 Built-on class (definition) 56
4.2.4 Extension-point class (definition) 57
4.2.5 Figure and Graphics framework (examples) 57

4.3 Framework use 62
4.3.1 Direct coupling through free role models 62
4.3.2 Examples of direct coupling 62
4.3.3 Properties of free role types 65

4.4 Framework extension 66
4.4.1 Domains and applications 66
4.4.2 Framework extension (definition) 66
4.4.3 Figure and SimpleFigures framework extensions (examples) 67

4.5 Framework layering 71
4.5.1 Layers and tiers 71
4.5.2 Traditional layer coupling 72
4.5.3 Role-model-based layer coupling 73
4.5.4 KidsEditor framework layering (example) 74

4.6 Framework documentation 76
4.6.1 What and when to document 76
4.6.2 How role modeling can help 77
4.6.3 A simple design documentation template 77

4.7 Summary 78

5 Extension of Industry Standards 79

5.1 Chapter overview and motivation 79

5.2 Common properties 80
5.2.1 Extending an industry standard 80
5.2.2 General requirements 82
5.2.3 Handling role types and role models 82
5.2.4 Figure framework (example) 83

5.3 Design notations 84
5.3.1 Extending design notations 85
5.3.2 Extending UML with role modeling 85
5.3.3 Extension properties 88

5.4 Programming languages 89
5.4.1 Extending programming languages 89
5.4.2 Problems of programming language extension 90
5.4.3 Extending Java with role modeling 91
5.4.4 Extending C++ with role modeling 94
5.4.5 Extending Smalltalk with role modeling 95
5.4.6 Extension properties 96

5.5 Summary 97

6 Case Study: The Geo Object Framework 99

6.1 Case study overview 99
6.1.1 Project history 99
6.1.2 The case study 100
6.1.3 Chapter structure 100

6.2 The Geo Object framework 101
6.2.1 Framework overview 101
6.2.2 Class model 101
6.2.3 Role models 102
6.2.4 Built-on classes 107
6.2.5 Example extension 107

6.3 Experiences and evaluation 109

6.3.1 Statistics of case study 109
6.3.2 Complexity of classes 111
6.3.3 Complexity of object collaboration 111
6.3.4 Clarity of requirements put upon use-clients 112
6.3.5 Reuse of experience 112

7 Case Study: The KMU Desktop Tools Framework 115

7.1 Case study overview 115
7.1.1 Project history 115
7.1.2 The case study 116
7.1.3 Chapter structure 116

7.2 The original Tools framework 116
7.2.1 Framework overview 117
7.2.2 Classes and their functionality 119
7.2.3 How to use the framework 122

7.3 Problems with the original framework 123

7.4 The redesigned Tools framework 123
7.4.1 Framework overview 124
7.4.2 Class model 125
7.4.3 Free role models 126
7.4.4 Internal role models 129

7.5 The new Environment framework 131

7.6 Experiences and evaluation 133
7.6.1 Statistics of case study 133
7.6.2 Complexity of classes 134
7.6.3 Complexity of object collaboration 135
7.6.4 Clarity of requirements put upon use-clients 135
7.6.5 Reuse of experience through design patterns 135
7.6.6 Further evolution of framework 136

8 Case Study: The JHotDraw Framework 137

8.1 Case study overview 137

8.1.1 JHotDraw history 137
8.1.2 The case study 138
8.1.3 Chapter structure 138

8.2 The JHotDraw framework 139
8.2.1 Design discussion overview 139
8.2.2 The Figure classes 140
8.2.3 The Drawing and DrawingView classes 144
8.2.4 The DrawingEditor classes 149

8.3 Experiences and evaluation 152
8.3.1 Statistics of the JHotDraw framework design 152
8.3.2 Observations from the case study 153
8.3.3 Comparison of documentation techniques 154
8.3.4 Complexity of classes 156
8.3.5 Complexity of object collaboration 157
8.3.6 Clarity of requirements put upon use-clients 158
8.3.7 Reuse of experience through design patterns 158

9 Thesis Validation 159

9.1 Thesis review and validation strategy 159

9.2 Thesis validation 161
9.2.1 Describes class as composition of role types 161
9.2.2 Breaks up relationship descriptions into role models 162
9.2.3 Makes requirements on clients explicit 163
9.2.4 Supports reuse of experience 164
9.2.5 Consolidation of validation 165

9.3 Summary (meaning of validation) 166

10 Conclusions 169

10.1 Contributions 169

10.2 Future work 170

10.3 Final conclusions 172

A References **173**

B Glossary **181**

C Design Notation **187**

C.1 Classes and role types 187

C.2 Object relationships 188

C.3 Class and role models 189

C.4 Role constraints 190

C.5 Role model shorthands 191

C.6 Frameworks 192

D Design Patterns **195**

D.1 Abstract Factory 196

D.2 Adapter 196

D.3 Bridge 196

D.4 Chain of Responsibility 197

D.5 Class Object 197

D.6 Composite 198

D.7 Decorator 198

D.8 Factory Method 199

D.9 Manager 199

D.10 Mediator 199

D.11 Metaobject 200

D.12 Null Object 200

D.13 Object Registry 201

D.14 Observer 201

D.15 Product Trader 201

D.16 Property List 202

D.17 Prototype 202

D.18 Role Object 203

D.19 Type Object 203

D.20 Serializer 204

D.21 Singleton 204

D.22 Specification 205

D.23 State 206

D.24 Strategy 206

D.25 Visitor 206

E Pointers to Further Material 209

Preface

What does the creator of a dissertation aspire to? To fulfill academic requirements for a Ph.D., for sure, but this is not the only goal. Maybe, some want to prepare for an academic career, and some want to impress their peers or their parents. My aspiration is to present a dissertation that does not only make a small but significant contribution to scientific progress, but that also provides significant help to the practicing software developer.

This dissertation addresses crucial technical problems of object-oriented framework design, a subdiscipline of software development whose importance has been increasing steadily over the last 10 years. Object-oriented software development in general, and framework-based development in particular, is gaining mass-market momentum because of Java, the emerging COBOL of the early next century (for better or worse).

UBS AG, a former employer of mine, has set up an aggressive training program for its more than thousand developers to learn Java. Naturally, these developers will work with Sun's JDK, a large object-oriented class library parts of which are best described as a set of frameworks. Some of these developers will face problems of designing frameworks, and all of them will face problems of learning and using frameworks. Three of these pertinent problems are addressed in this dissertation. I hope that the concepts from my dissertation will ease the lives of my former and current colleagues. For some it already has, as one of the case studies in this dissertation shows.

Originally, I had thought of my intended dissertation work as a kind of ordeal. Judging from what successful colleagues had told me, finding a worthwhile Ph.D. thesis, writing a dissertation, and persisting in all trials is like descending to hell, being eaten alive, and returning to the surface while trying to maintain a normal life. Nothing of this has happened to me (including the normal life). I managed to nail down the thesis proposal in 1997, continue with my research and its consolidation in 1998, and finish writing up the dissertation in 1999. One reason for this is that I could build on my earlier research work dating back to 1994, even though originally I had not intended to do so.

However, the most important reason for succeeding is the guidance and support I received from my advisor, Thomas Gross. I have learned immensely from him. He taught me how to find direction in Ph.D. thesis work, drill down to the essentials, and make a dissertation out of it. I am highly grateful to him for making my Ph.D. thesis work such a smooth and rewarding experience.

Also, of importance to this work is the supportive environment I found at Ubilab, the IT research laboratory of UBS AG, where I worked until the end of March 1999. I appreciate my colleague's interest in my work and their morale support, for which I would like to thank them very much. In particular, the creative atmosphere in the Software Engineering group, originally with Kai-Uwe Mätzel, and later with Erica Dubach and Hans Wegener supported my ascent to prevail on top of this dissertation. I wish to thank them all.

I have collaborated with many people over the last years, and in one way or another, they have influenced my thinking. Of particular importance to me are the discussions I had with Dirk Bäumer, Daniel Megert, and Wolf Siberski. I would like to thank them very much. In a similar vein, I would like to thank Walter Bischofberger, Gregory Hutchinson, Birgit Rieder, Bruno Schäffer, and Heinz Züllighoven.

The dissertation went through a number of releases ("I released early, I released often"). Ralph Johnson commented on the OOPSLA 1998 release, Dirk Bäumer commented on the Christmas 1998 release, nobody commented on the February 1999 release, and Wolf Siberski commented on the May 1999 release. Moira Norrie and Douglas Schmidt commented on the final release through the feedback I got at my doctoral examination. Thomas Gross commented on all of the releases. I am indebted to all of them for their insightful comments that helped me improve this dissertation.

Dirk Riehle

Zurich, Switzerland. June 1999.

Mannheim, Germany, February 2000.

1

Introduction

Frameworks are of key importance for developing large-scale object-oriented software systems. They promise higher productivity and shorter time-to-market through design and code reuse. However, many projects report that this promise is hard to fulfill: the design of object-oriented frameworks is all but well understood. This introductory chapter sheds some light on why this is so, and presents several key problems. It thereby poses the research questions that have driven the work presented in this dissertation, and gives an overview of how and where these questions are answered in this work.

1.1 Why object-oriented frameworks?

Object-oriented frameworks promise *higher productivity* and *shorter time-to-market* of application development *through design and code reuse* (than possible with non-framework based approaches).

Object orientation comprises object-oriented analysis, object-oriented design, and object-oriented programming. Using a small set of concepts (objects, classes, and their relationships), developers can model an application domain (analysis), define a software architecture to represent that model on a computer (design), and implement the architecture to let a computer execute the model.

None of these activities (analysis, design, and implementation), nor the resulting models, are trivial. To carry them out effectively, developers have invented additional concepts that represent the conceptual entities they are dealing with. One such key concept is the object-oriented framework.

An object-oriented framework is a reusable design together with an implementation [JF88, CIM92, Lew95, FS97, FSJ99]. The design represents a model of an application domain or a pertinent aspect thereof, and the implementation defines how this model can be executed, at least partially. A good

framework's design and implementation is the result of a deep understanding of the application domain, usually gained by developing several applications for that domain. The framework represents the cumulated experience of how the software architecture and its implementation for most applications in the domain should look like. It leaves enough room for customization to solve a particular problem in the application domain.

Developers who apply a framework reuse its design and implementation. They do so to solve an application problem that falls into the domain modeled by the framework. By reusing the design, application developers customize a (hopefully) well-understood software architecture to their own specific application problem. This helps them get the key aspects of the architecture right from the beginning. By reusing the implementation, application developers get up to speed more quickly.

Through design and code reuse, frameworks help developers achieve higher productivity and shorter time-to-market in application development.

Designing and implementing object-oriented frameworks is hard. Typically, it requires several iterations to get a framework "right" (which might mean nothing more than that it gets a pause before the evolution of domain requirements leads to yet another redesign and re-implementation of the framework).

In contrast to non-framework based application development, a framework requires additional up-front investments. If a framework is bought, it requires money to buy it and time to learn it. If a framework is developed in-house, it requires time and resources, both to develop it, and later to teach it or to learn it.

However, the promise of a significant increase in productivity and reduction in time-to-market makes the investment worthwhile in most cases. Today, every large object-oriented development project I know of uses frameworks in one way or the other. Yet, while there is no way around frameworks in large-scale object-oriented software development, they are all but well understood and sometimes do not live up to their promises.

1.2 Problems with frameworks

What is wrong with object-oriented software development based on frameworks? No single answer can be given. Most projects that fail do so for a multitude of reasons, with poor design and implementation quality of frameworks being only one of them. Yet, frameworks can significantly contribute to project success and ensure flexibility and evolvability of applications. It is therefore worthwhile to investigate current problems in framework development and search for solutions.

Case studies [FK97], current practice [BBE95], as well as my own experiences with frameworks in whose development I have participated, which I have used, or which I have reviewed suggest the following key problems (these are elaborated in Chapter 2):

- *Class complexity.* Classes define the behavior of objects, their instances. Objects collaborate in multiple contexts, for multiple purposes, exhibiting task-specific behavior. For complex objects, the definition of this behavior in a single flat class interface is inadequate, and better mechanisms that describe the different aspects of objects, are needed.
- *Complementary focus on classes and collaborations.* Objects collaborate with each other, for different purposes. Much of the complexity of frameworks goes into designing and implementing the object collaboration behavior. By assigning responsibilities to individual objects, the focus on overall collaborative behavior is lost. Thus, mechanisms to describe collaboration behavior are needed.

- *Object collaboration complexity*. The overall collaborative behavior of framework objects and their collaboration with client objects may become complex. To make the overall object collaboration easier to understand and to manage, it needs to be broken up into independent pieces. Means to describe task-specific collaborative behavior of objects are needed, as well as mechanisms to compose these pieces of collaborative behavior to define the full object collaboration.
- *Difficulties with using a framework.* It is easy to use a framework in ways unforeseen and not intended by their original designers. In particular, the requirements that a framework puts upon its clients are frequently unclear and unspecified. Framework misuse causes constant work-arounds for the client and may easily lead to unstable and buggy code. Thus, mechanisms are needed that help prevent the (mis-)use of frameworks in ways not intended by the framework developers.

These are the problems addressed in this dissertation. Of course, these problems are only a subset of technical problems with frameworks. Yet, they are a particularly pertinent kind of problems.

These problems suggest a *separation of concerns* approach: because complexity is high, it needs to reduced. Reduction of complexity is achieved by breaking up the framework up into (re-)composable pieces. Each of the pieces can then be analyzed, designed, and understood individually.

1.3 Role modeling for framework design

Role-based object-oriented modeling (or, for short, role modeling), is such a *separation of concerns* approach. This dissertation presents an approach for designing object-oriented frameworks using role modeling. The basic role modeling concepts are role, role type, object collaboration task, and role model. The next paragraphs, summarized from Chapter 3, shortly explain the concepts.

Objects do not occur in a vacuum. Rather, each object collaborates with other objects: it does so by playing *roles.* A role is an observable behavioral aspect of an object. While playing roles, an object collaborates with other objects, usually for several different purposes at once. Each such well-defined purpose of object collaboration is an *object collaboration task.* The composition of all object collaboration tasks becomes the overall *object collaboration.*

A *role type* describes each role an object may play. A role type is a regular type specification. Each object collaboration task is described by a *role model.* A role model uses role types to describe how an object in an object collaboration task must behave (play a role) if it wants to properly carry out its part of the work. The role types of a role model relate to each other using regular object relationship descriptions. The role model is effectively a specification of the set of possible valid object collaboration tasks.

A *class* defines the behavior of objects, their instances. A class is the composition of several role types, each of which is taken from a role model and assigned to one or more classes. The composition of role types forms the class type. Classes and role models are complementary: a role model focuses on one particular task of object collaboration, ignoring others, and a class focuses on how roles played in different tasks come together in one kind of object.

A *class model* describes the overall collaboration of objects. A class model is a set of classes that are related with each other through role models and class inheritance. The object relationship descriptions between the classes can be derived from the relationship descriptions between the role types from the role models. This way, role models serve as the interconnecting glue between classes, not only showing the overall structure, but the individual object collaboration tasks that make up the class model. Effectively, with the help of classes, the class model composes the different role models to define what a valid overall object collaboration is.

This dissertation extends the basic modeling approach to support the design of frameworks, leading to the *role modeling for framework design* approach. The next paragraphs, summarized from Chapter 4, shortly explain the involved concepts.

A *framework* is a class model that defines the collaboration of (framework) objects (and their client objects with respect to the framework). Next to the framework-internal role models, a framework defines so-called *free role models* that provide a bridge between the framework client and the framework. *Free role types* are those role types of a free role model that are to be picked up by client classes. They specify how client objects of the framework must behave to make proper use of framework objects. Only by playing roles defined by free role types may client objects make use of a framework.

Frameworks build on other frameworks by picking up these other frameworks' free role types and assigning them to one or more of their own classes. This mechanism of making use of frameworks by free role models is applied recursively to view systems as layers of frameworks and framework extension stacked on top of each other. Framework extensions are a set of subclasses of framework classes that add new free role models. Through the use of free role models new clients may make use of the extended framework functionality as well.

How does this approach help in overcoming the problems stated above?

- *Class complexity.* The complexity of a class is reduced by breaking its interface up into distinct role types. Experience shows that it is easier to understand the parts first and then to compose them rather than trying to understand the whole all at once.
- *Complementary focus on classes and collaborations.* The explicit focus on object collaborations and class models rather than just class hierarchies ensures that both perspectives are taken into account and represented in the design of a framework.
- *Object collaboration complexity.* The complexity of the overall collaborative behavior of framework objects is reduced by breaking it up into object collaboration tasks and describing it as the composition of role models. Again, understanding the parts first makes understanding the whole easier.
- *Difficulties using a framework.* The concept of free role model lets developers specify succinctly how a framework is to be used by use-relationship based clients. Free role models help to prevent misuse that would otherwise occur easily if the requirements put upon clients are not clarified.

Role modeling for framework design is evolutionary in nature rather the revolutionary. It is used as an addition to traditional class-based modeling. Role modeling does not try to replace class-based modeling but rather to refine it and improve over it where necessary. Developers who use this approach do not have to throw away existing investments, but may selectively apply the approach where necessary.

1.4 Dissertation overview

The dissertation comprises three main parts. The first part describes the role modeling approach and how it is applied to framework design. The second part presents three case studies, each one focussed on evaluating different aspects of the approach. The third and final part reiterates the claims posed by the dissertation, and validates them using qualitative arguments and experiences gained from the case studies.

The first part on role modeling for framework design comprises Chapters 2 to 5.

- *Frameworks, related work, and dissertation thesis.* Chapter 2 puts object-oriented frameworks into the context of software architecture. It reviews related work on object-oriented software architecture and frameworks and lists remaining problems. Based on these problems, it develops the dissertation thesis.
- *Role modeling.* Chapter 3 introduces the role modeling foundations, on which Chapter 4 on framework design builds. Chapter 3 shows how the traditional class-based modeling approach can be extended with the concepts of role type and role model, and how existing concepts need to be revised.
- *Framework design.* Chapter 4 builds on the role modeling foundation from Chapter 3. It develops a comprehensive definition of the concept of framework and discusses its context. It covers framework definition, framework use, and framework extension. It also discusses framework layering and documentation.
- *Extension of industry standards with role modeling concepts.* Chapter 5 shows how current industry standards can be extended with role modeling concepts. Such an extension lets developers use role modeling with current standards. The chapter provides extensions of UML, Java, C++, and Smalltalk.

The second part presents the case studies. It comprises Chapters 6 to 8.

- *The Geo Object framework.* Chapter 6 presents the Geo Object framework as a case study. This framework is the root framework of the Geo system, a distributed object system based on a metalevel architecture. The Object framework provides core abstractions like Object and Class common to many industrial systems.
- *The KMU Desktop Tools framework.* Chapter 7 presents the KMU Desktop Tools framework as a case study. The framework is used to build software tools for desktop applications. It is based on the Tools and Materials Metaphor approach to software development.
- *The JHotDraw drawing editor framework.* Chapter 8 presents the JHotDraw framework for drawing editors as a case study. JHotDraw is based on HotDraw, which is a widely known and mature framework used to build drawing editor applications.

The third part validates the claim and draws conclusions from the dissertation. It comprises Chapters 9 and 10.

- *Validation of dissertation thesis.* Chapter 9 reviews the experiences made in the case studies, supports them with qualitative arguments, and validates the dissertation thesis.
- *Conclusions.* Chapter 10 reviews the dissertation as a whole. It explains the consequences of the dissertation results, both from a narrow perspective (the thesis claim), and a wider perspective.

Finally, Appendices A to E provide additional material.

- *References.* Appendix A lists the references of work referred to by the dissertation.
- *Glossary.* Appendix B provides a glossary of key terms defined by the dissertation.
- *Notation guide.* Appendix C provides a notation guide to role modeling for framework design.
- *Design patterns.* Appendix D presents common design patterns cast in a role model form.
- *Further material.* Appendix E provides pointers to further material.

For comprehensive understanding, the theory chapters must be read first. Then individual case studies may be read, followed by the validation of the dissertation thesis. Those who want to get an impression only may restrict their reading to Chapter 3 and 4 and to one or more case studies. Alternatively, they may directly jump into a case study, using Appendix C, the notation guide, as a minimal introduction to role modeling for framework design.

1.5 Actors in this dissertation

For properly identifying actors in this dissertation, I use the following conventions:

- “we” identifies the reader and the author (you and me),
- “the project team” identifies the team described in Chapter 6,
- “the redesign team” identifies the team described in Chapter 7.

“The project team” and “the redesign team” are only used in their respective chapter. They are sometimes abbreviated as “the team”.

2

Frameworks, Related Work, and Thesis Statement

Object-oriented frameworks are cohesive design and implementation artifacts. Frameworks typically serve to implement (larger-scale) components, and are implemented using (smaller-scale) classes. This chapter describes the framework concept and its purpose in the context of object-oriented software architecture. Then the chapter identifies a set of key problems with object-oriented frameworks, and discusses to which extent related work addresses these problems. Finally, the thesis of this dissertation is defined and explained.

2.1 Overview of framework concepts

This section first puts frameworks into the context of object-oriented software architecture, then reviews common and established terminology, and finally provides a list of key problems that haunt today's application development based on frameworks.

2.1.1 Object-oriented software architecture

Object-oriented software architecture is based on objects and classes as the primitive building blocks. Current practice distinguishes three levels of system granularity: the class level, the framework level, and the component level. (Sometimes a fourth component framework level is added, but it is ignored here, because it does not add anything important to the discussion of object-oriented frameworks.)

On the smallest level of granularity, a system can be designed using classes, whose instances collaborate with each other. A class defines a well-defined and bounded chunk of behavior and state, based on

a domain concept that it represents. Objects are instances of classes, and thereby represent instances of a domain concept.

For small systems, objects and classes are sufficient means for describing the architecture of a system. However, when a system gets bigger, more and more classes get involved in its architecture, and higher-level abstractions are needed that let developers design and implement systems.

Systems of medium size can be described as a set of collaborating frameworks and framework extensions. A framework is a cohesive design and implementation artifact [JF88, CIM92, Lew95, FS97, FSJ99]. It represents a specific domain or an important aspect thereof as a reusable design of abstract classes (or interfaces) and concrete classes, together with a set of implementations. Good implementations are reusable, and may or may not be readily instantiable.

A good framework has well-defined boundaries, along which it interacts with clients, and an implementation that is hidden from the outside. Frameworks and framework extensions are a key part of medium to large-scale software development [BCG95, BGK+97], but even they have an upper limit of coping with complexity.

On a large-scale level of granularity, a system can be described as a set of collaborating components, each of which may have been built from one or more object frameworks [WSP+92, Szy98]. A component is a well-defined technical artifact with interfaces to other components. It may or may not have been implemented using object-oriented frameworks-in case of an object-oriented system it typically is.

There are many more software architecture related issues like process architecture, code bundling, and build systems. However, they do not add to the discussion and are therefore omitted from it.

2.1.2 Review of framework terminology

Frameworks model a specific domain or an important aspect thereof. They represent the domain as an abstract design, consisting of abstract classes (or interfaces). The abstract design is more than a set of classes, because it defines how instances of the classes are allowed to collaborate with each other at runtime. Effectively, it acts as a skeleton, or a scaffolding, that determines how framework objects relate to each other.

A framework comes with reusable implementations in the form of abstract and concrete class implementations. Abstract implementations are abstract classes that implement parts of a framework abstraction (as expressed by an abstract class or interface), but leave crucial implementation decisions to subclasses. They do so using the principle of *Design by Primitives*. Design by Primitives bases a class implementation on a small set of primitive operations that are left open for implementation through subclasses. Concrete subclasses implement these operations so that they can be instantiated and used without further subclassing. A detailed discussion of the use of interfaces, abstract classes, and concrete classes in object-oriented design is presented in [RD99a, RD99b] (for German readers: [Rie97d, Rie97e]).

When designing an application, a developer may choose to use an existing framework, because it models parts of the application domain well. There are two primary ways of using a framework: either by use-relationships or by inheritance. Consequently, there are two kinds of clients: use-relationship based clients and inheritance-based clients (use-clients and extension clients).

A use-client object instantiates one or more framework classes and uses the objects for its purposes. A framework, whose classes can be readily instantiated, and which can be used as-is, is called a black-box framework [JF88]. An extension client class subclasses framework classes according to its needs. It thereby customizes the general domain model represented by the framework to its specific application needs. The new subclasses are used by use-clients of the specific application. A framework that

can be extended using subclassing is called a white-box framework [JF88]. Most real-world frameworks combine black-box with white-box issues and are therefore called gray-box frameworks.

Typically, applications use not only one framework, but several. An application may tie together a framework for handling user-interfaces, another framework for the banking domain, and yet another framework for handling persistence and database access.

An application framework is a framework that ties together a set of existing frameworks to cover most aspects of a certain type of application. Strictly speaking, an application framework is just another framework of a similar size as the frameworks it uses. Most developers, however, when speaking of an application framework, refer to all involved frameworks, the primary application framework as well as all the other frameworks it is based on (an application framework thereby may be viewed as something like a composite framework).

In software development based on application frameworks, applications (or systems) become extensions of the application framework. They reuse the software architecture of this particular type of application as defined by the application framework and its domain frameworks.

There are a number of key advantages to be gained from using application frameworks. The primary technical advantage is that they provide design and code reuse. The larger and better an application framework, the more design and code reuse becomes possible. Also, systems based on frameworks are easier to maintain, because most key design and implementation decisions are localized in one place, the framework.

The primary business advantages of design and code reuse are higher developer productivity and shorter time-to-market of new applications. In addition, applications tend to be less buggy (they are reusing mature implementations), and tend to look more homogenous (a suite of applications built from the same framework has the same architecture and similar implementations).

2.1.3 Problems with frameworks

Software development in general, and object-oriented framework-based application development in particular, is a non-trivial undertaking that comes with many problems. These include technical issues, social and communication issues, project management issues, and developer organization issues.

This dissertation focuses on technical problems of design complexity. Fortunately, the problems presented here can be addressed largely independently of social, managerial, and organizational problems as they emerge in real-world software development.

Based on four large case studies, Fichman and Kemerer note that a key challenge of adopting object technology is the steep learning curve it takes until developers can get to work productively [FK97]. Bischofberger, Birrer, and Eggenschwiler note that beyond basic object-oriented software development, object-oriented frameworks require an even higher up-front learning investment [BBE95]. In terms of resources, it takes time and good developers until a project or an organization can start using a framework successfully. Frameworks have a steep learning curve, because developers not only have to understand single isolated classes, but abstract designs of several classes whose instances collaborate for many different purposes.

What makes up this increased complexity of using object-oriented frameworks over the basic object paradigm? The case studies cited above, current practice, and my own observations point to the following problems:

- *Class complexity*. Classes define the behavior of objects, their instances. Objects collaborate in multiple contexts, for multiple purposes, exhibiting task-specific behavior. For complex objects, the definition of this behavior in a single flat class interface is inadequate, and better mechanisms that describe the different aspects of objects, are needed.

- *Complementary focus on classes and collaborations.* Objects collaborate with each other, for different purposes. Much of the complexity of frameworks goes into designing and implementing the object collaboration behavior. By assigning responsibilities to individual objects, the focus on overall collaborative behavior is lost. Thus, mechanisms to describe collaboration behavior are needed.
- *Object collaboration complexity*. The overall collaborative behavior of framework objects and their collaboration with client objects may become complex. To make the overall object collaboration easier to understand and to manage, it needs to be broken up into independent pieces. Means to describe task-specific collaborative behavior of objects are needed, as well as mechanisms to compose these pieces of collaborative behavior to define the full object collaboration.
- *Difficulties with using a framework.* It is easy to use a framework in ways unforeseen and not intended by their original designers. In particular, the requirements that a framework puts upon its clients are frequently unclear and unspecified. Framework misuse causes constant work-arounds for the client and may easily lead to unstable and buggy code. Thus, mechanisms are needed that help prevent the (mis-)use of frameworks in ways not intended by the framework developers.

The first three list items are problems with understanding and learning an existing framework. All four bullet list items are also problems with using a framework, as understanding it precedes using it.

These problems are a result of the complexity of designing and implementing a framework in the first place. They arise in the initial (and continued) development of a framework. Developers do not have (yet) adequate means to tackle these problems in the design and implementation process.

Any framework that is being used in real-world projects continuously evolves, because its underlying domain keeps changing [Bäu98]. The design is in constant danger to deteriorate, and the implementation is in constant danger to get convoluted. After a few evolutionary steps, redesign and reimplementation of the framework become a necessity. In our fast-paced time this must be carried out as effectively as possible.

The thesis of this dissertation is that the listed problems of designing and understanding frameworks can be eased significantly through the use of a role modeling approach to framework design. Role modeling for framework design is an addition of the traditional class-based modeling approach. Before the thesis is laid out in more detail, however, the next section reviews related work and how it addresses the presented problems.

2.2 Related work

Related work can be characterized along three dimensions: work on object-oriented design and frameworks, work on modeling languages and methods for (possibly persistent) object-oriented systems, and work on separation of concerns techniques for object-oriented design and programming.

2.2.1 Object-oriented design

Objects interact in several different contexts at once. An early understanding of this observation can be traced back to Smalltalk that provides developers with method categories to group methods into [GR89] and Objective-C that lets developers specify different protocols of objects [Cox87]. In Smalltalk, each method category can be devoted to one particular aspect of the class, and it can be viewed independently of the other aspects. However, Smalltalk provides a few standardized method catego-

ries, most notably “Accessing”, which suggest that method categories are more a convenience function than a means of class design.

Programming languages like C++ and Java make it possible for classes to have different interfaces through the use of multiple inheritance [Str94, AG96]. The presence of interfaces and multiple inheritance explicitly acknowledges the need to view objects from different perspectives. Also, industry component models like COM or CORBA [Box98, Sie96] and academic component models like those of Wright, Darwin, and Rapide [All97, MDEK95, LKA+95] all allow components to have multiple interfaces.

Also, it is now widely understood that object collaboration is a major design issue and should be explicitly focused on. The idea of object collaborations goes back to Beck and Cunningham [BC89] and Wirfs-Brock et al. [WWW90]. However, the seminal paper is [HGG90], which introduces the concept of contract as a formal description of object collaboration behavior. A contract abstracts from individual objects and focuses on the collaborative behavior of objects. Contracts can be extended and refined, using a specialization mechanism.

2.2.2 Programming methods

For a few years now, researchers have been working on new techniques for making object systems more readily composable from pieces than possible before. The understanding that objects and classes need better code composition mechanisms than use and inheritance relationships is driving most of this research.

In his work on the *Demeter* method, Lieberherr criticizes the tendency of classes to hard-code their relationships [LH89, Lie95]. Class structures are hard to change if these structures are embedded in the class implementations. He therefore suggests to abstract from concrete class structures and to define constraints on class structures only. These constraints do not describe a specific class structure but rather a family of class structures that all fulfill the constraints. Object-oriented programs are generated from these constraints and from a set of propagation patterns. A propagation pattern describes how to carry out a specific task in terms of traversing object structures and calculating results. The full program is generated by a tool that takes the class structure constraints and propagation patterns, derives a class structure, and assigns implementations to classes that reflect both the tasks defined by the propagation patterns and the chosen class structure.

In their work on *subject-oriented programming*, Harrison and Ossher also point out that an object may be viewed from many different angles [HO93, OKH+95]. Starting with this observation, they derive a method that lets developers compose new applications from existing applications, where each application may have an arbitrarily complex class model. They introduce composition operations for classes and methods, and provide developers with tools to specify how to compose different applications. This approach distinguishes itself from the other approaches in that it uses the traditional concepts of classes and methods as its basic building blocks, and can be applied to existing applications.

In his work on the *DASCO* method, Rito-Silva describes techniques for the development of concurrent distributed systems [Rit97]. He views applications as the result of a composition of different frameworks, where each framework addresses one particular technical aspect of a system (for example, a particular type of object synchronization or concurrency control). Composing classes from these frameworks using multiple inheritance defines the final application, in which all different aspects come together.

In their work on *aspect-oriented programming*, Kiczales et al. address the problem of “tangled code”, a manifestation of the feature interaction problem [KLM+97, Lop97]. Tangled code is the phenomenon that in many class implementations different technical issues come together. The implementation of a single operation of a class might have to address issues of synchronization and concurrency, logging and security, and others, before focusing on the primary domain task of the operation. Kiczales et

al. suggest that it is desirable to describe each of these aspects in a programming language of its own, rather than using a generic common language. A dedicated tool, the Aspect Weaver, composes these different programs for the different technical aspects, and generates a single resulting program. Introducing dedicated programming languages lets developers define programs dedicated to one particular technical aspect. The approach thereby lets developers separate concerns and "untangle" class implementations. The approach promises higher flexibility and simpler maintainability and evolution.

In their work on *role-oriented programming*, VanHilst and Notkin view object-oriented designs as compositions of object collaboration descriptions [Van97, VN96]. They describe object collaborations as a set of roles, which objects in the collaboration play. They provide means to compose these object collaborations. Their composition mechanism are parameterized types (templates in C++), for which they provide elaborate handling guidelines.

In the context of database programming, Albano et al. describe how to *implement data models with roles* [ABGO93]. Their database programming language Fibonacci lets developers describe data models in terms of roles objects play. Objects are fully hidden behind roles and programmers never get to see an object except through a mediating role. Roles are described by role types, which are organized in type hierarchies. For these type hierarchies, the usual subtyping mechanisms apply.

Database programming languages focus more on data structures and object-lifecycles than behavior specification. Hence, Fibonacci is strong in the area of dynamically acquiring and loosing roles, as may happen during the lifetime of an object. This is helpful for easing schema evolution, one of the hard problems of database design and programming. However, this is not a topic of this dissertation.

2.2.3 Development methods

OOram is a full-fledged software development method that covers all relevant activities of the development process [Ree96, RAB+92]. It was developed by Trygve Reenskaug et al. and has been in use for many years. OOram is of particular importance to this dissertation, because it served as one of its sources of inspiration.

OOram takes a different approach to modeling than traditional class based modeling. It focuses on object collaborations, which it describes using role models. Objects in a collaboration play a role, which is described using a type. Classes are irrelevant on this modeling level; they serve as implementation concepts only.

A role model describes just one particular object collaboration purpose, and is therefore composed with other role models to fully determine object behavior and hence class implementations. The composition is called role model synthesis, and is supported by dedicated tools. A full object-oriented system emerges as the composition of many role models. There is no intermediate concept (like the framework concept), but only role models of growing complexity.

Andersen's dissertation follows up on OOram [And97]. He rehabilitates the concept of class, which he defines to model the information content of objects by abstracting from individual objects. Andersen also emphasizes that role models are instance-level models that do not abstract from individual objects but are always bound to specific objects. In contrast to this view, this dissertation views role models as a specification of object collaboration tasks that may fit a possibly infinite set of object collaborations at runtime.

Finally, Andersen's work provides a formal foundation for the role model synthesis process of OOram. This formal foundation is of particular importance for this dissertation, because the dissertation does not present a new type specification mechanism but rather relies on existing ones. Anderson's mechanism is one possible specification mechanism that can be used to formally specify role models as used in this dissertation.

UML is currently becoming the dominating modeling language in the object-oriented world [UML97a, UML 97b]. In UML, relationships between classes can be defined that are tagged at each end with a so-called role name, a simple string that indicates the role an instance of the other class in the relationship may play. It is possible to add additional specification using general UML mechanisms, but no specific role modeling support is provided.

Next to basic role support on relationships between classes, UML also provides the concept of Collaboration that lets developers describe how objects are to collaborate for a specific task. The authors of UML write that Collaborations serve to represent different things, including use-cases and OOram role models. Collaborations let developers specify structure, but not behavior. Its expressive power is largely equivalent to basic OOram role models. However, it is lacking more advanced features like role model synthesis (composition of collaborations). This is where method extensions come into play.

Catalysis is a recent software development method for component-based software development with UML. It is being developed by D'Souza and Wills [DW98]. It is based on UML and tries to leverage its modeling capabilities as far as possible. Therefore, it does not introduce new concepts like role and role model, but rather uses the concepts of interface and class diagram.

However, D'Souza and Wills use class diagrams and interfaces much like Reenskaug uses role models and roles, except that they do not view their role–model–alike class diagrams as instance-level models, but rather as type-level models. They use multiple inheritance as the composition mechanism for interfaces. They support interface specification and composition through lightweight formal modeling based on pre-/postconditions and invariants.

However, much like OOram, Catalysis does not introduce an explicit framework concept as used in this dissertation. Rather, they scale up using UML concepts only, which does not provide a dedicated framework concept. Catalysis uses the term framework, but it refers to a single class diagram dedicated to one particular purpose, much like a role model.

Object Role Modeling (ORM, [Hal96]) is a data modeling technique that extends ER modeling with role modeling concepts. Similarly to UML, it lets developers tag the end of associations with role names, thereby letting them indicate what the purpose of an entity type at an association end is. As explained by Halpin in [Hal98], the differences between ORM and UML are on a fine-grained level, for example in the details of how to attach meaning to roles. However, unlike UML, ORM provides no means to specify behavior, as possible with UML collaborations. Like most data structure oriented modeling techniques, ORM falls short when it comes to specifying collaborative behavior in object-oriented design.

2.2.4 Role modeling concepts

Some related work adds directly to the role concept as a key modeling concept without trying to develop a full-fledged design or programming method. Depending on the researchers' background, such related work has a specific focus like conceptual modeling, runtime configuration, or database design.

Kristensen and Osterbye introduce the role concept to the Scandinavian tradition of conceptual modeling and programming [KO96a]. The role meta-concept is viewed as a specialization of the more general concept meta-concept. Operations that are applicable to the concept meta-concept therefore become applicable to the role meta-concept, for example, classification, aggregation, and specialization. The definition of a role is the result of an abstraction process from similar domain phenomena (which are not necessarily domain objects). Roles can be aggregated to form aggregate roles, and they can be specialized to form new derived roles. In Kristensen's and Osterbye's terminology, role instances are attached to objects. While objects provide intrinsic properties, roles attach extrinsic properties to objects. Properties are both operations and attributes. Kristensen and Osterbye call the resulting concept instance conglomerate (object + attached roles) a subject.

Gottlob et al. [GSR96] discuss a role concept for better supporting the life-cycle of objects in object-oriented databases. Long-lived objects, in particular those stored in a database, may have a complex life-cycle. Gottlob et al. describe the lifecycle of an object as a succession of roles the object may take on. At any one time, an object is an instance of exactly one class, but it may take on several roles. Roles are modeled in role hierarchies, which are type hierarchies. An object that is in the particular state of playing a certain role acts according to the role's type definition in the role hierarchy. Gottlob et al. provide a Smalltalk implementation that lets an object dynamically acquire and drop roles, even if they are of the same role type. The Smalltalk implementation keeps the different roles synchronized.

Wieringa et al. also use roles for modeling object life-cycles [WJS95]. Their conceptual modeling approach is similar to Gottlob et al.'s and comprises role class hierarchies and dynamic role playing. However, where Gottlob et al. provide a Smalltalk implementation of roles, Wieringa et al. provide a formal specification of the metamodel behind roles and classes using order-sorted logic.

Also closely related to Gottlob et al.'s approach is the concept of role objects, as discussed by Bäumer et al. [BRSW00]. A role object is an object that extends a core object for use in a specific context. The core object provides all functionality that is independent of a specific context, and a role object provides all functionality that is needed in one specific context. A core object may provide many different role objects so that it can operate in many different contexts in parallel. Of particular importance to this dissertation is that role objects may be used to integrate frameworks. One framework may define a core concept like Person, and further frameworks may define role objects that extend the core concept for their particular use [BGK+97, RG98]. Then, role objects serve as a dynamic bridge between frameworks. They are used to adapt a framework to unforeseen requirements.

The focus of the related work presented in this subsection is on the individual object and its roles. None of the work addresses the collaborative behavior of objects through its roles, and none of the work applies it to the level of framework design.

2.2.5 Object-oriented frameworks

The seminal paper on framework concepts is [JF88], where Johnson and Foote pick up the framework concept and discuss many of its properties. They introduce the concepts of white-box and black-box frameworks. They distinguish between classes, types, and protocols. The protocol of an object is the set of methods (operations) that it understands. They use protocol as a synonym for type and argue that it is to be treated differently from a class. A class implements a particular protocol, which points to recent considerations of classes as implementation concepts only (rather than modeling concepts, as defined by their original SIMULA-67 inventors [DH72]).

Frameworks were soon recognized as a means for achieving large-scale reuse. The Taligent frameworks are a notable effort for building systems based on frameworks. This effort is also well published, even though commercially it failed in the end. In a series of books [CP95, Tal95], Taligent tried to introduce framework concepts and a lightweight terminology into the marketplace. They introduce the concept of ensemble to mean the set of application-specific subclasses of frameworks that make up an application. An ensemble draws on several frameworks at once. Furthermore, as already suggested by Johnson and Foote through the introduction of white-box and black-box frameworks, they make a clear distinction between the different interfaces clients may use of a framework, which they call composition-focused and inheritance focused, respectively.

Frameworks are usually used in the larger context of object-oriented software architecture. Because frameworks are always focused on a particular domain, they establish domain-specific software architectures (DSSA's). Bäumer describes an example of such a DSSA for interactive desktop applications [Bäu98, BGK+97]. He discusses a U-form layering structure of application systems, with applications on top of business section frameworks on top of business domain frameworks, on top of desktop/technology frameworks, on top of foundation frameworks. The layering structure takes on a U-

form, because layering is not strict, and any higher-layer framework makes use of the desktop/technology and the foundation framework layers. These two base layers form the containing borders of the jar-like U-Form. Therefore, this DSSA employs non-strict layering, a common property of object-oriented systems.

Of particular interest is Bäumer's adaptation of the connector concept from software architecture [SG96] to object-oriented frameworks. He shows that frameworks are frequently connected using (instantiations of) design patterns as connecting elements. The results of this dissertation support the importance of this observation, and provide a more thorough basis for its discussion in the form of role models. Role models may be design pattern instantiations (but need not!), and one of their purposes is to connect frameworks with clients or other frameworks.

Further reports on successfully using object-oriented frameworks support the reuse claim [BCG95, SBF96]. [Lew95] is a first compendium of concrete examples of frameworks from all kinds of domains. Another forthcoming book by Fayad, Schmidt, and Johnson provides even more study material [FSJ99].

2.2.6 Review of related work

The recent interest in separation of concerns approaches suggests that today's design and implementation techniques need further improvement. The work presented in this dissertation is based on role modeling, which is one of the separation of concerns approaches. None of the related work, be it on individual design concepts or full design methods, on general programming or database programming, or on frameworks or databases, utilizes role modeling for framework design. However, this is the core them of this dissertation.

2.3 Thesis statement of dissertation

This subsection explains and refines the dissertation thesis. The initial thesis statement is:

> **Thesis statement (initial version)**
>
> Role modeling for framework design makes the design and documentation of object-oriented frameworks easier than is possible with traditional class-based approaches.

Analysis of the initial thesis statement leads to three questions that need further discussion:

- What is the scope of "design and documentation"?
- What does "easier" mean? Can it be made more specific?
- Who is the subject that benefits from an increased ease of use?

The next subsections examine each question in turn.

2.3.1 What is the scope of "design and documentation"?

First, we need to distinguish between the different activities and the tangible and intangible artifacts of framework-based application development. Figure 2-1 shows a diagram of these activities and arti-

facts. This diagram has been developed to explain the thesis and serves illustration purposes only. It is not a complete description of development activities, dependencies, and artifacts.

In Figure 2-1, tangible and intangible artifacts are shown in large fonts, and activities relating artifacts are shown in small font.

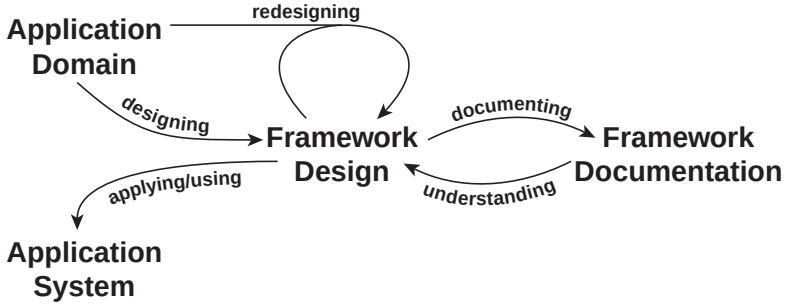

Figure 2-1: Activities and artifacts during framework design and application.

The diagram shows four artifacts: application domain, application system, framework design, and framework documentation. It shows the distinction between artifacts and activities well. For example, framework design is an artifact, while designing is an activity. Of the artifacts, only documentation is fully tangible. The application domain, the application system, and the framework design artifacts are partially or fully abstract.

It is important to distinguish between a framework's design and its documentation. The framework design comprises all the ideas and concepts involved in a framework. The framework documentation makes them explicit as far as possible, but is never able to fully reveal them. Rather, the documentation describes those aspects of the design that are most important for developers to know who try to understand the framework.

The artifacts are related by activities. Designing a system in an application domain leads to a design. Documenting the design leads to documentation. Reading documentation and working with a framework leads to understanding the framework design. Applying the framework leads to an application. Etc.

Role modeling for framework design is a method that supports human activities. It is also reflected in the artifacts, for example in the documentation, but its primary purpose is to support humans in carrying out activities based on these artifacts. Therefore, the degree to which role modeling helps with these activities is the primary measure of its utility.

The diagram shows five activities: designing, redesigning, applying/using, documenting, and understanding.

Let us assume that someone versed in role modeling for framework design carries out these activities. Also, let us assume that role modeling has already been used for the existing frameworks and their documentation. Then, any redesign activity becomes a design activity that takes both the application domain and an existing design into account. Because the existing framework design is already a result of designing for the application domain, no qualitative difference exists, and redesign can safely be subsumed under design.

Moreover, documenting a framework using role modeling is an activity by which developers select certain aspects of a framework and document them using whatever scheme seems appropriate, here role modeling. The decision of which aspect is considered important is independent of the deployed technique. Therefore, the documentation activity is largely independent of the role modeling approach.

This discussion leaves us with designing, applying/using, and understanding object-oriented frameworks. The refined thesis statement now becomes:

Thesis statement (first intermediate version)

Role modeling for framework design makes the following activities easier to carry out than is possible with traditional class-based approaches:

- designing and redesigning a framework,
- learning a framework from its documentation,
- using a framework that is already understood.

What about frameworks that have not been designed and documented with role modeling in mind? Can role modeling help here as well? As I have demonstrated, role modeling can be used as an analytical means to reengineer existing designs [Rie97]. Naturally then, the new design can be expressed well using role modeling.

However, such a reengineering activity is also an example of the more general design activity for object-oriented frameworks. Here, the application domain is the underlying domain of the existing framework, and the reengineering activity becomes a design activity for the application domain taking the purpose and structure of the existing framework into account. Therefore, reengineering is also subsumed under design.

2.3.2 What does “easier” mean?

“Easier” is used in the sense of “more effective” in handling the problems of framework-based application development as discussed earlier in Section 2.1.3.

The refined thesis statement of the dissertation now becomes:

Thesis statement (second intermediate version)

Role modeling for framework design makes the following activities easier to carry out than is possible with traditional class-based approaches:

- designing and redesigning a framework,
- learning a framework from its documentation,
- using a framework that is already understood.

The following problems are addressed and their severity is reduced:

- complexity of classes,
- complexity of object collaboration,
- clarity of requirements put upon use-clients.

What remains to be done is to define an evaluation strategy that lets us compare the effectiveness of the traditional class-based approach with role modeling for framework design as presented in this dissertation.

The phrase “[problem] severity is reduced” does not lay claim to a quantitative measure. In principle, showing only a tiny improvement over the existing situation would validate the dissertation thesis. While a correct solution, this is certainly not a satisfying solution. However, no quantitative measure or metric is used to validate the claims; rather, the dissertation uses a case study based approach.

If the dissertation were to measure the improvement in ease of use, it would have to introduce a metric that measures the reduction in complexity as experienced by a user. In principle, this means setting up and conducting a psychological experiment for framework design and use tasks of non-trivial complexity. Such an experiment is beyond the scope of this dissertation and can only be carried out by trained psychologists.

A less aggressive alternative would be to define quantitative metrics of complexity in framework design. Such metrics could then be calculated for a traditional framework design and for a framework design based on role modeling. A "better" resulting metric value can then be interpreted as a validation of the dissertation thesis. The problem of this approach is that it is difficult to show that a metric adequately reflects the design complexity as perceived by a human. Again, psychological experiments would be needed to show this.

As a consequence, I resort to using a qualitative approach based on case studies to validate the dissertation thesis. Chapters 6 to 8 present several case studies and Chapter 9 evaluates the thesis based on experiences made in the case studies.

2.3.3 Who is the subject?

Role modeling for framework design makes life easier for expert framework developers and users. It is tempting to conclude that role modeling helps novices and experts alike. However, I have no hard evidence that role modeling works for novices to the same extent that it does work for experts. Role modeling for framework design is a more comprehensive method than traditional class based modeling. It requires experience with framework design and use. If this experience is not given, role modeling for framework design may be too demanding a method for some developers.

Therefore, the thesis statement of the dissertation only claims that the presented approach helps expert developers that already understand class-based modeling and framework design well. This is well in line with the understanding that the presented approach is an extension of the existing class-based modeling approach, rather than a replacement. One needs to understand the fundamental modeling concepts first, before adding new ones.

2.3.4 Final version of the thesis

After the review of these three issues with the initial thesis statement, the final statement now becomes:

Thesis statement (final version)

Role modeling for framework design makes the following activities easier to carry out for the expert framework developer and user than is possible with traditional class-based approaches:

- designing and redesigning a framework,
- learning a framework from its documentation,
- using a framework that is already understood.

The following problems are addressed and their severity is reduced:

- complexity of classes,
- complexity of object collaboration,
- clarity of requirements put upon use-clients.

The three activity dimensions designing/redesigning, learning, and using a framework all relate equally well to the three problem dimensions complexity of classes, complexity of object collaboration, and clarity of requirements put upon use-clients. The result is a 3 x 3 matrix of activity/problem pairs, as shown in Table 2-1.

(activity, problem) matrix	*designing and redesigning a framework*	*learning a framework from its documentation*	*using an already understood framework*
complexity of classes	Validity to be shown.	Validity to be shown.	Validity to be shown.
complexity of object collaboration	Validity to be shown.	Validity to be shown.	Validity to be shown.
clarity of requirements put upon use-clients	Validity to be shown.	Validity to be shown.	Validity to be shown.

Table 2-1: The dissertation thesis broken up into nine sub-theses.

The thesis validation of Chapter 9 uses this matrix as the basis of the validation strategy. For each pair, an argument is made on why the specific problem is reduced in its severity when carrying out the associated activity. The validation of the overall thesis becomes the sum of the validations of the individual sub-theses.

The remainder of this work explains and validates the dissertation thesis.

3

Role Modeling

Object-oriented modeling is based on the concepts of object, class, and their relationships. This chapter extends these concepts with new role modeling concepts. It starts out with the definition of concepts like object, class, and class model, to which it adds the definitions of the concepts of role, role type, role model, and role constraint. This leads to an extended object modeling terminology, based not only on objects and classes, but on roles and role types, as well as class models and role models. The concepts for framework design, defined in the following chapter, build on this foundation.

3.1 Chapter overview

This chapter presents two categories of modeling concepts:

- *Traditional object modeling concepts.* This part defines the concepts object and value, class, class type, and value type, association and association description, aggregation and aggregation description, inheritance, object collaboration, and class model. This section provides the object modeling basis.
- *New role modeling concepts.* This part defines the concepts role, role type, role constraint, object collaboration task, and role model. It also revises and extends the object modeling basis to better support role modeling. It presents a novel perspective on the role concept, as needed for framework design and documentation.

Appendix B, the glossary, lists all concept definitions in alphabetical order. Appendix C, the notation guide, summarizes the visual notation used in the dissertation. The visual notation is derived from UML, with additions only for those concepts not known to UML.

This and the next chapter use the design of a graphical figure framework as a running example. This chapter discusses the class model only; the next chapter discusses its definition as a framework and its use by clients like drawing editors.

Graphical figure objects are part of a drawing, as shown by and manipulated through a drawing editor. Figure 3-1 shows the graphical user interface of such an example drawing editor. At the center of the figure is a drawing area, which contains several graphical figures.

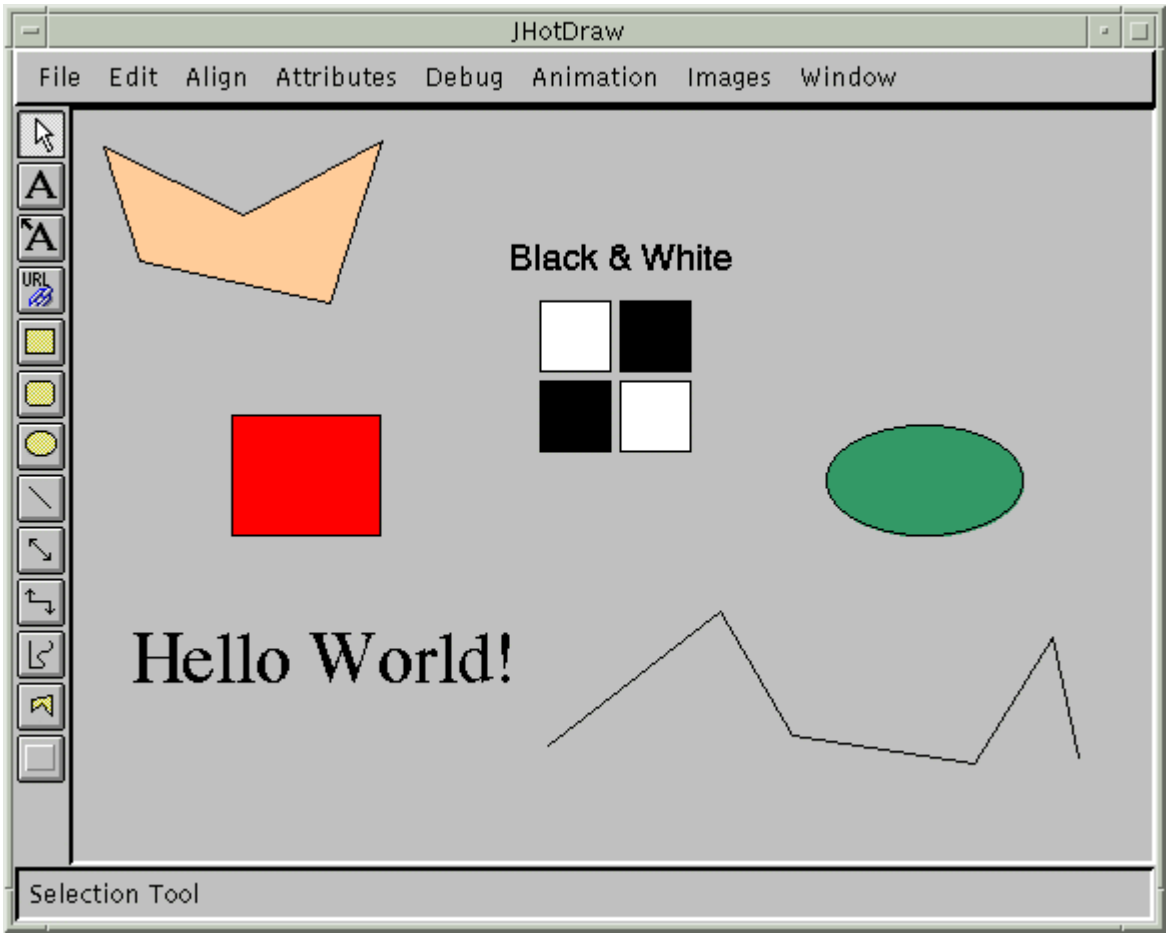

Figure 3-1: Screenshot of an example drawing editor (JHotDraw).

Graphical figures may be generic figures like polygon, rectangle, circle, and text figures. They may also be domain-specific figures like UML-style object, interface, association, and aggregation figures as used in a UML diagram editor, or like animal or circus figures as used in a drawing editor for children.

Some of the figures are *composite graphical figures.* A composite graphical figure is a figure that is composed from further embedded figures (it aggregates these sub-figures). Examples of composite graphical figures are predefined figures like the UML-style class or interface figures, which are build from polygon, rectangle, and text figures, and user-defined figures like those composed by a grouping mechanism as found in drawing editors. Thus, a composite graphical figure can be realized as a hierarchy of figure objects.

Figure 3-2 illustrates the runtime object hierarchy for the graphical figures of the drawing from Figure 3-1.

Section 3.2 defines the basic object modeling terminology. It uses the graphical figure class hierarchy indicated above as its illustrating example. The hierarchy is presented as a class model.

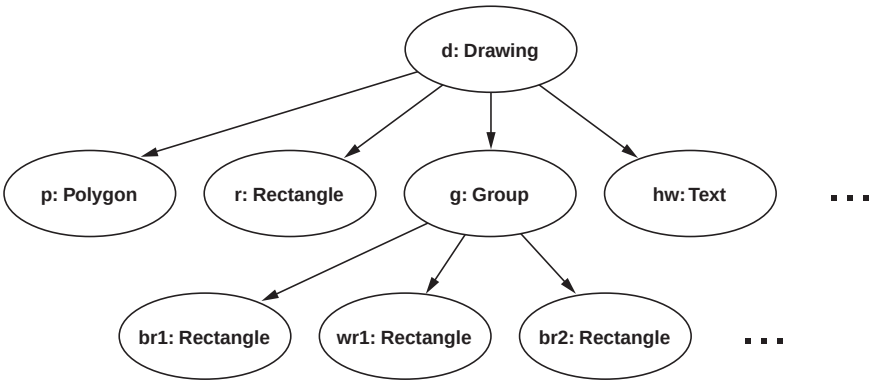

Figure 3-2: Object hierarchy of graphical figures.

Section 3.3 defines the role modeling extension of the object modeling basis. It enhances the graphical figure example with roles and role models to show the structure of object collaborations.

The concepts are presented in a programming language independent way. They do not reflect the full complexity of today's programming languages. Chapter 5 shows how the modeling concepts map on other notations and programming languages like UML, Java, C++ and Smalltalk.

3.2 Object modeling fundamentals

Objects and classes are the traditional modeling concepts of object-oriented software systems. This section defines and illustrates the concepts of object, class, class type, operation, value, value type, and interpretation function. It also defines the concepts of association and association description, aggregation and aggregation description, and inheritance. Finally, it defines the concepts of object collaboration and class model. Some definitions are preliminary and are extended in the subsequent sections when the role modeling concepts are introduced.

This section is not meant to provide a comprehensive object modeling terminology. It only defines the most fundamental concepts, on which the subsequent role modeling concepts build. It omits most of the complexity found in industrial-strength modeling approaches like UML [UML97a, UML97b] or OML [FHG98], which provide a much larger variety of modeling concepts than those presented here.

3.2.1 Object and class (definition)

The concepts of object and class can be defined from a technical or a conceptual perspective. We use the technical definitions, and view the conceptual definitions as a description of the purpose of objects and classes.

Definition 3-1: Object

An object is an opaque runtime entity of a system that provides state and operations to query and change that state. An object has a lifecycle: It is created, may change over time, and is possibly deleted. Objects can be identified unambiguously; identity is an intrinsic property of every object.

An object represents a concrete or abstract phenomenon from a domain. The domain may be a non-technical application domain like banking or insurance, or it may be a technical application domain like multithreading, synchronization, or distribution. This second modeling-oriented definition of objects as representations of phenomena complements the first definition of objects as encapsulated state with operations.

Examples of objects are all kinds of graphical figures, for example polygons, rectangles, triangles, and circles, but also higher-level graphical figures, like UML-style class, interface, association, and inheritance figures.

Figure 3-3 shows the visual representation of the object concept as used in this work. In such a diagram, an object typically has a name, like “aFigure”, and provides its class name, like “Polygon”, set in parenthesis below the object name.

Figure 3-3: Example of an object called p of class Polygon.

An object is an instance of a class. The class defines the properties of its instances.

Definition 3-2: Class

A class is the definition of a (possibly infinite) set of objects, called its *instances*. A class defines the behavior of its instances using a *class type.* A *class type* is a type, specified using an appropriate type specification mechanism.

With each class, exactly one class type is associated. The class type defines the operations applicable to an instance of the class in terms of the effects these operations have on its state and in terms of the values returned to an object calling an operation.

A class is the result of an abstraction process from several similar objects from a domain. It describes the common properties of all its instances and ignores any properties that are irrelevant for the modeling task at hand. Again, this second definition of class as the abstraction from similar recurring phenomena complements the first definition of class as the definition of the common properties of a set of objects.

Examples of classes are PolygonFigure, RectangleFigure, and TextFigure. Some visual instances are shown in Figure 3-1 as polygons, rectangles, texts on a drawing area, and some regular instances are shown in Figure 3-2 as objects in an object diagram. Similarly, more complex examples are the UmlClassFigure and UmlAssociationFigure classes, as they are needed for a UML-diagram drawing editor.

An object may be an *immediate* (or *direct*) instance of a class or not. An immediate instance is an object that conforms to a class without obeying further specified properties. An object that conforms to a class but that is not an immediate instance of this class typically is an instance of a subclass of this class (see Section 3.2.5).

A class may be *abstract* or *concrete*. An abstract class cannot have immediate instances. A concrete class may have immediate instances. The aforementioned PolygonFigure, RectangleFigure, and TextFigure classes are concrete classes. The class Figure, which represents all properties common to figure objects, is an abstract class, of which no direct instances may exist.

Figure 3-4 shows the visual representation of the class concept. If the class name is set in Italics, the class is an abstract class. If not, it is a concrete class. The body of the visual class representation shows parts of its class type, as defined below.

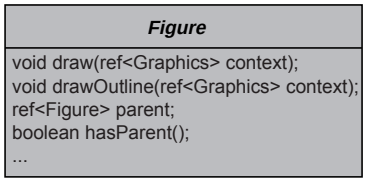

Figure 3-4: Example of a class, here the Figure class.

The concept of class is understood and used here as a modeling concept. On an implementation level, a design-level class may be represented using any appropriate mechanism. In Java, for example, design-level classes are frequently expressed using Java interfaces, and Java classes are used to provide implementations of the design-level classes.

The class type can be specified using an appropriate type specification mechanism, for example Liskov & Wing [LW93a, LW93b, LW94] or Abadi & Cardelli [AC96]. The only precondition is that the chosen specification mechanism must provide a proper composition operator on types. This is needed, as shown in the Section 3.3, to define a class type as the composition of several role types.

Figure 3-5 shows the visual representation of the type concept.

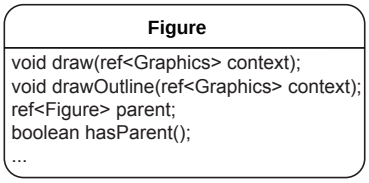

Figure 3-5: Example of a type, here the class type of Figure.

Class types, role types, and value types (see below) are expressed and shown using the same visual symbol.

3.2.2 Value and value type (definition)

Objects and values are complementary modeling concepts of equal importance [Mac82, BRS+98].

Definition 3-3: Value

A value is an atomic entity from the abstract and invisible universe of values. A value cannot be perceived directly, but only through occurrences of its representations. The representations are interpreted by means of *interpretation functions*. These interpretation functions return further (occurrences of representations of) values; they do not change the value.

In contrast to objects, values do not live in time: they are not created, do not change, and are not destroyed.

A value is always bound to an *attribute* of an object. An attribute is a name/value pair that puts a label, the attribute name, on some aspect of the object's state space, the attribute value, so that for a given object, the attribute's value can be named, assessed, and changed.

Examples of values are integers, strings, but also domain-specific values like account numbers, security tickets, and monetary amounts. Values of particular importance to object systems are object references, because they enable objects to communicate. The so-called primitive values like integers, strings, and object references are typically directly supported by a modeling notation or a programming language, while domain-specific values need to be defined by programmers.

Definition 3-4: Value type

A value type is a type that specifies a set of values together with the interpretation functions applicable to representations of members of this set.

Value types (or type constructors for a specific kind of value type) may be directly provided by a modeling language, like integer, string, and object reference. Or, value types may be domain-specific and introduced by programmers, like Color, 2DPoint and 3DPoint, etc.

For a given object attribute, an *attribute type* is defined, which determines the set of possible values that can be assigned to the attribute. An attribute type is always a value type.

There are no universal context-independent criteria to decide whether some domain phenomenon should be modeled as an object or a value. The purpose of the model and the suitability of its possible implementations always drive such a decision.

3.2.3 Figure class (example)

As explained, a class defines the state and behavior of its instances. It does so with the help of a *class type*. In the simplest case, a class type is a set of attributes, a set of constraints on the attribute values, a set of transitions between allowed states, and a set of operations that trigger state transitions. The constraints on the attributes determine the subspace of the overall state space defined by the cross product of the attributes' value types. Next to this simple scheme, more elaborate type specification mechanisms can be used.

For the purposes of this work every type specification mechanism is suitable, as long as it provides a well-defined composition operation on types. To better illustrate the examples, and to simplify their discussion, we use a simple type specification mechanism. A type is described as a set of operations, a set of attributes, and an informal annotation that describes its meaning (either as comments in the specification text or in the main body of the respective section). The possible state transitions are implied by the operation definitions.

For example, the class type of class Figure defines an attribute with the name "parent" that is of type "object reference to Figure object". Also, class Figure may have an attribute called extent of value type rectangle, with the constraint that the rectangle must have a positive non-zero extent. (Please note that the value type rectangle is a mathematical concept, hence a value type, while the class Rectangle-Figure is a graphical figure class, which may not only have an attribute extent of type rectangle, but many other attributes, like fill color, line stroke color, or line stroke width.)

Specification 3-1 describes the class Figure and its class type. Next to the Figure class, it uses a Graphics class that represents the drawing area. The specification uses Java syntax-alike constructs. The uncommon syntax "ref<Figure>" stands for the value type "object reference to Figure object". Thus, "ref" is a type constructor for object reference value types. This pseudo-code serves illustration purposes only. It is not based on a full-fledged specification or programming language.

```
class Figure {
  // Provide basic domain functionality of figures.
  // Figures have origin, extent. Can be drawn, moved, and resized.
  // Figures have a handle by which they are manipulated.
  point origin;
  rectangle extent;
  void draw(ref<Graphics> context);
  void drawOutline(ref<Graphics> context);
  void place(point location);
  void move(int dx, int dy);
  void resize(int handle, int dx, int dy);
```

```
  // Provide parent reference; may be null for root.
  // Parent object must be of type Figure.
  ref<Figure> parent;
  boolean hasParent();
  void setParent(ref<Figure> parent);

  // Manage objects registered as dependents.
  // Notify them about state changes of the figure.
  // Dependents must be of type FigureObserver.
  collection<ref<FigureObserver>> dependents;
  boolean hasObserver(ref<FigureObserver> observer);
  void addObserver(ref<FigureObserver> observer);
  void removeObserver(ref<FigureObserver> observer);

  // Generically manage properties of figure.
  // Examples properties are fill color, frame color, line stroke.
  // Properties may be any kind of Object.
  collection<ref<Object>> properties;
  void hasProperty(string name);
  void getProperty(string name);
  void setProperty(string name, ref<Object> property);
  void unsetProperty(string name);

  ... More definition.
}
```

Specification 3-1: Example specification of Figure class.

This example class defines its instances' state space together with operations to change it. The existence of an attribute in the class type definition implies an operation to get the attribute's value, but not necessarily an operation to set it. A set-operation needs to be specified explicitly.

3.2.4 Relationships and relationship descriptions (definition)

At runtime, objects are connected with each other. Such a connection may be implemented in different ways. On the modeling level, we only need to express that we allow for a certain type of connection between objects.

A connection between two objects is called an object relationship. For the object modeling basis, we consider two types of object relationships: object associations and object aggregations. These are runtime entities stating that two objects are connected with each other in a particular way.

An object relationship description defines what a valid object relationship is. We consider object association descriptions and object aggregation descriptions. These relationship descriptions can be annotated to add meaning and more precisely specify the set of allowed relationships according to that relationship description.

Object associations may be unidirectional or bi-directional. Object aggregations are always unidirectional. However, this information is effectively an annotation of the relationship description that is used to help implementing the model. The overall graph of object relationship descriptions is undirected, independently of the specification of individual relationship descriptions.

3.2.4.1 Association and association description (definition)

Objects relate to each other by means of object associations.

Definition 3-5: Object association

An *object association* is a pair of objects (x, y), stating that an object x holds a reference to another object y of which it may or may not make use.

Holding a reference means that a parameter of an invocation of an operation of object x or that an attribute of object x has the reference to y as its value. An example of an object association is (aText, aUmlClass), which indicates that the object aText is associated with the object aUmlClass (for example, by means of its parent attribute).

Figure 3-6 shows the visual representation of a unidirectional object association.

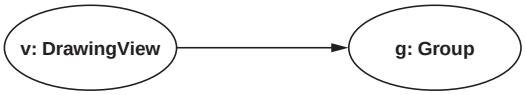

Figure 3-6: Example of an association between two objects.

Object associations may be bi-directional, which means that for a given association (x, y), the inverse (y, x) exists. This is visually shown through the omission of the arrowhead, see Figure 3-7.

Figure 3-7: Example of a bi-directional association between two objects.

Classes prescribe how their instances may relate to each other by means of object association descriptions.

Definition 3-6: Object association description

An *object association description* is a pair of types (X, Y) that determines possible runtime object associations. An association between two objects (x, y) conforms to the association description if x is of type X or a subtype of X, and if y is of type Y or a subtype of Y.

When speaking of an association description between two classes, the class types of the classes are meant.

Object association descriptions between classes are specifications of how instances of the classes may relate to each other at runtime. A specific object association is called a valid object association with respect to an object association description if it conforms to the description.

Figure 3-8 shows the visual representation of an object association description for unidirectional object associations, and Figure 3-9 shows the visual representation of an object association description for bi-directional object associations.

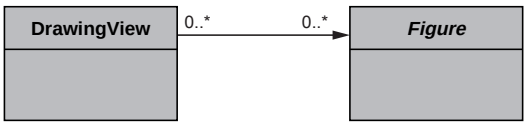

Figure 3-8: Example of a unidirectional object association description.

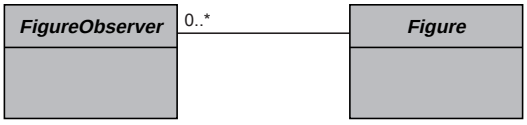

Figure 3-9: Example of a bi-directional object association description.

Being unidirectional or being bi-directional are annotations on the first or second element of the pair representing the object association description. They add meaning to the association description that can be used in the implementation of the design. However, the underlying graph is still undirected.

Object association descriptions (and object aggregation descriptions, see below) are annotated with further information. Each annotation is tied to one end of the description. Examples of annotations are the name of the description, the cardinality of the source or target, and possible existence dependency constraints.

For example, Figures 3-8 and 3-9 annotate the association descriptions with the cardinality of runtime object associations. If no cardinality is given, a default value of 0..1 is assumed.

3.2.4.2 Aggregation and aggregation description (definition)

Objects also relate to each other by means of aggregation.

Definition 3-7: Object aggregation

An *object aggregation* is a pair of objects (x, y), stating that an object x aggregates an object y as a part of it. To aggregate an object means to control it, not only to make use of it, but to determine its lifetime and accessibility as well.

One object may aggregate several other objects, but may itself be aggregated at maximum once. Object aggregations form acyclic object hierarchies. Odell gives an in-depth discussion of the concept of aggregation [Ode98]. An example of an aggregation is (aUmlClass, aText), which indicates that the object aUmlClass aggregates the object aText (for example to display the class name).

Figure 3-10 shows the visual representation of an aggregation. It is distinguished from an association through the diamond on the side of the aggregating object.

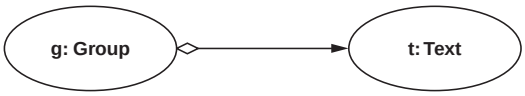

Figure 3-10: Example of an aggregation between two objects.

Classes prescribe how their instances may relate to each other by means of aggregation descriptions.

Definition 3-8: Object aggregation description

An *object aggregation description* is a pair of types (X, Y) that determines possible runtime object aggregations. An aggregation between two objects (x, y) conforms to the aggregation description if x is of type X or a subtype of X, and if y is of type Y or a subtype of Y.

When speaking of an association description between two classes, the class types of the classes are meant.

Object aggregation descriptions between classes are specifications of how instances of the classes may aggregate each other at runtime. A specific object aggregation is called a valid object aggregation with respect to an object aggregation description if it conforms to the description.

Figure 3-11 shows the visual representation of an aggregation description.

Like object association descriptions, aggregation descriptions are not aggregations but rather descriptions of them. Aggregation descriptions provide information about the allowed runtime aggregations like cardinality of the aggregated objects, etc. In an aggregation description, the cardinality annotation

of the aggregating class is always 0..1; thus the annotation can be omitted in a figure depicting the aggregation description.

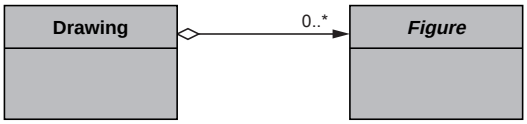

Figure 3-11: Example of an aggregation description between two classes.

3.2.5 Inheritance (definition)

Classes may inherit from each other. Inheritance is understood here as a modeling concept to define specialization/generalization hierarchies rather than as a construct to reuse code.

Definition 3-9: Inheritance

An *inheritance* is a pair of classes (X, Y) such that any instance of class Y can be substituted in a context where an instance of class X is expected.

X is called the *superclass* of Y and Y is called the *subclass* of X.

Conceptually speaking, the concept modeled by class Y is a specialization of the concept modeled by class X and the concept modeled by class X is a generalization of the concept modeled by class Y. The pair (Figure, RectangleFigure) is an example of inheritance. It indicates that RectangleFigure is a subclass of Figure.

Wegener and Zdonik discuss in more detail what "substituted in a context where an instance of class X is expected" means [WZ88]. Further definitions of substitutability have been given by Liskov [Lis88], Liskov and Wing [LW93a, LW93b, LW94], Abadi and Cardelli [AC96], and others.

We use single inheritance only. Thus, for a given inheritance (X, Y), there may not be another inheritance (Z, Y) with $Z \neq X$. (This will seem less a restriction, once role types are introduced.)

Figure 3-12 illustrates the visual representation of the inheritance concept.

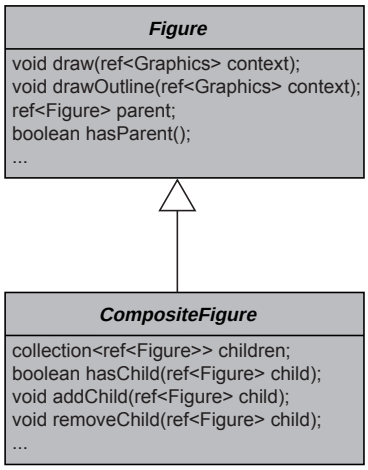

Figure 3-12: Example of an inheritance between two classes.

According to the abstract superclass rule, a class should not inherit from a concrete class [Hür94]. Thus, in a class inheritance hierarchy all leaf classes should be concrete classes, and all other classes are abstract classes.

In practice, this rule is often relaxed, and concrete classes inherit from concrete classes. If done right, this is not a problem, because the concrete superclass actually assumes two different roles: one as an abstract superclass for subclasses and one as a concrete class that is readily instantiable by use-clients.

3.2.6 Object collaboration and class model (definition)

For modeling object systems, we must define what a system is. Moreover, we are not interested in just about any system, but only those considered valid, because only they provide both expected and useful behavior.

At runtime, an object system is a set of objects that relate to each other by means of object associations and aggregations. Which of the possibly infinite number of object systems is a valid one and which is not? Object collaborations and class models provide the necessary means to better define what a valid system is.

Definition 3-10: Object collaboration

An *object collaboration* is a set of objects that relate to each other by object relationships.

An *object system* is an object collaboration. An object collaboration or an object system is said to be valid if it conforms to a class model.

Definition 3-11: Class model

A *class model* is a set of classes that relate to each other by inheritance and object relationship descriptions. The class relationship graph must be non-partitioned.

The class relationship graph is the graph of all object relationship descriptions, which may be either object association or object aggregation descriptions. The graph is undirected and non-partitioned.

A *class model* acts as a specification of a (possibly infinite) set of runtime object collaborations. Its purpose is to define what a valid object collaboration is and what not. If an object collaboration is an element of the set of object collaborations specified by the class model, it is said to conform to the class model, and hence is said to be a valid object collaboration.

3.2.7 Figure class model (example)

We can now present a first class model. We use the classes Figure, PolygonFigure, RectangleFigure, TextFigure, CompositeFigure, GroupFigure, UmlClassFigure, and UmlAssociationFigure. Figure and CompositeFigure are abstract classes, while all other classes are concrete classes.

Figure 3-2 presents an example object collaboration. A few non-composite as well as a few composite objects can be seen. To understand the structure of this object hierarchy, the classes Figure and CompositeFigure need to be defined. Class Figure is described in the previous class example subsection, and class CompositeFigure is described in Specification 3-2.

```
class CompositeFigure extends Figure {
  // Manage child figure objects.
  // Child figures must be of type Figure.
  collection<ref<Figure>> children;
  boolean hasChild(ref<Figure> child);
  void addChild(ref<Figure> child);
  void removeChild(ref<Figure> child);

  ... More definition.
}
```

Specification 3-2: Example specification of CompositeFigure class.

The class CompositeFigure defines what it means to be a composite figure object: it may have child objects, aggregated by the very composite object. Therefore, all classes whose instances are composite figure objects inherit from CompositeFigure.

The classes PolygonFigure, RectangleFigure, and TextFigure are direct subclasses of Figure. Thus, they represent non-composite figure objects. The classes GroupFigure, UmlClassFigure, and UmlAssociationFigure are subclasses of CompositeFigure. Thus, they represent composite figure objects.

Figure 3-13 shows the resulting class model.

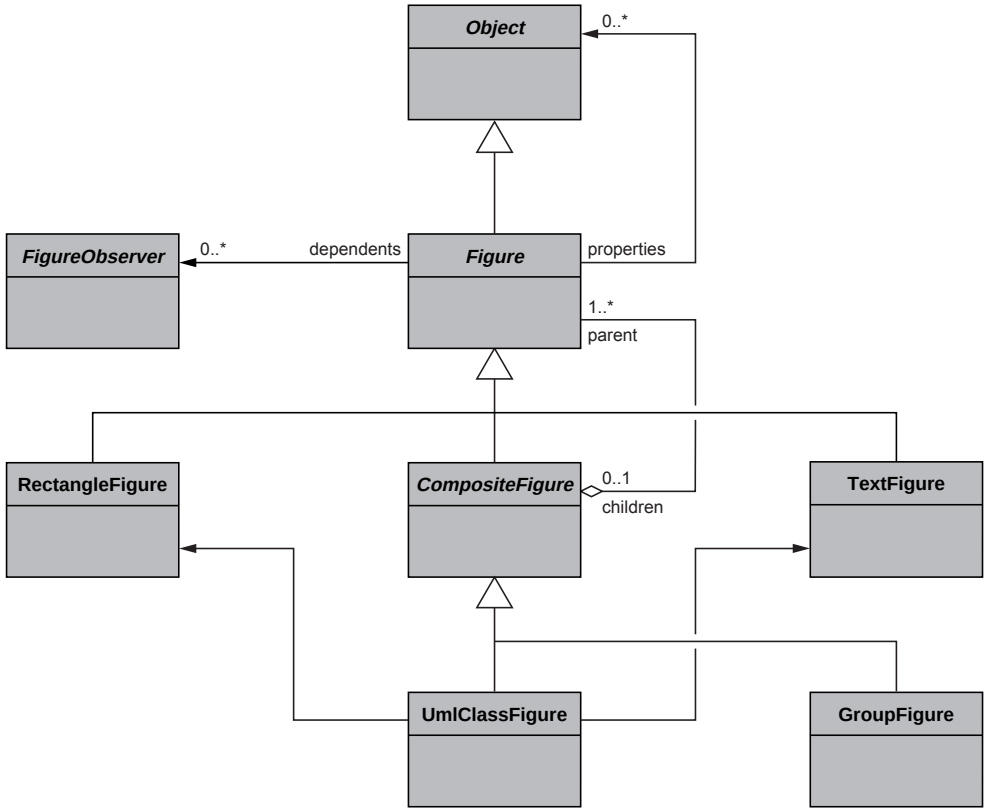

Figure 3-13: Class model of Figure example.

Figure 3-13 expresses the concept of composite figure object by means of the object aggregation description between the CompositeFigure and Figure classes. The possibility to traverse an object hierarchy from the leaves up is expressed by the parent association description from Figure to CompositeFigure. The object associations between a graphical figure and its dependents are shown by an association description between Figure and FigureObserver. Finally, an object association description between Figure and Object shows how a figure provides its properties to clients.

The association descriptions between the composite figure classes UmlAssociationFigure and UmlClassFigure and the primitive figure object classes PolygonFigure, RectangleFigure and TextFigure are omitted.

3.3 Role modeling extensions

This section extends the concepts defined in the previous section with role modeling concepts. It introduces new concepts and revises old ones.

Object collaborations are described as interacting objects that play roles to provide predefined behavior. Because an object may play several roles in an object collaboration, the collaboration itself is broken up into pieces, called object collaboration tasks. Each object collaboration task is described by a role model, and a role type from the role model describes each role in the collaboration task. A role model describes the collaboration task as a set of role types and their relationships, independent of other collaboration tasks.

A role model describes the set of valid object collaboration tasks, much like a class model describes the set of valid object collaborations. Also, a class model is broken up into different role models, much like an object collaboration is broken up into collaboration tasks. The collaboration tasks compose to become the full object collaboration, and the role models compose to become the full class model. Role models thereby separate the different concerns involved in a class model and reduce the complexity of designing and understanding it.

This section defines the concepts role and role type, role constraint, object collaboration task and role model. The class and class model concepts are extended to integrate smoothly with the role modeling concepts. The Figure example is revised in light of the new modeling concepts.

3.3.1 Role and role type (definition)

Roles are observable behavioral aspects of objects. A role is described by a role type.

Definition 3-12: Role

A *role* is an observable behavioral aspect of an object.

An object, which provides a particular role, is said to *play* that role.

An object may play several roles at once (which every non-trivial object does). The roles, which an object plays, interact, and an operation called in the context of one role can easily lead to the object acting in the context of another role. The set of roles an object is playing is called the object's *role set.*

A role represents the behavior of an object with respect to a specific object collaboration task. Objects as phenomena from a domain typically behave in many different ways, acting in different use-contexts. Therefore, in each context, an object plays a different role. The role is determined by the view the client holds on the object playing the role.

Each graphical figure object provides a set of roles to clients. Among these are roles that let clients draw the object, traverse the object hierarchy, make the object persistent, and inform dependent objects about state changes.

A role type abstracts from the behavior of similar roles.

Definition 3-13: Role type

A role type is a type that defines the behavior of a role an object may play. It defines the operations and the state model of the role, as well as the associated semantics.

An object's behavior is defined by the composition of all role types of all roles it may play. Different objects may play a role that is defined by the same role type. Thus, a role type is (in principle) independent of a particular object or class.

A role type is the result of an abstraction process from similar behavioral aspects of objects from a domain, much like classes are the abstraction from structurally and behaviorally similar objects. In contrast to a class that fully defines an object, a role type only defines one possible behavioral aspect of an object.

A role type is visually represented as a type. For example, Figure 3-14 shows the Figure role type.

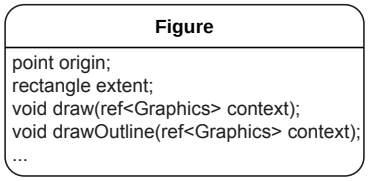

Figure 3-14: Example of a role type.

Examples of role types are the type definitions of the roles mentioned above. These role types might be named Figure, Child, and Subject. The Figure role type defines the domain-specific behavior of figure objects, like its capability to draw itself on some drawing area. It ignores other aspects like its child behavior in an object hierarchy.

If a textual specification of a role type is given, it is not necessary to repeat the type specification as part of the visual representation in a diagram. It is sufficient to provide the name. Figure 3-15 shows this convenience shortcut for the Figure, Child, and Subject role types.

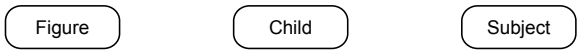

Figure 3-15: Example role types.

Some role types may have no operations associated with them. For example, client role types frequently specify only behavior of objects, but no operations. (A client acts according to its role type specification, but it does not offer operations for other objects to call back.)

Definition 3-14: No-operation role type (no-op role type)

A no-operation role type is a role type that defines no operations.

Still, behavioral semantics may be associated with such no-operation role types. This is sometimes the case, but not always. But even if no semantics are associated with a no-operation role type, the role type still serves a useful purpose as a handle for a role played by an object.

Definition 3-15: No-semantics role type

A no-semantics role type is a no-operation role type that defines neither state nor behavior.

No-semantics role types are used to designate objects whose references are passed around but which are not made use of in the context of the current role model. No-semantics role types can be attached to a class after it has been defined.

For pragmatic purposes, diagrams identify no-operation role types by a mark in their upper right corner. Figure 3-16 shows an example.

Figure 3-16: Example no-op role type.

The importance of no-operation and no-semantics role types will become apparent in the context of role models (see Subsection 3.3.5) and class models (see Subsection 3.3.9).

3.3.2 Figure, Child, etc. (example)

Specification 3-3 describes the Figure, Child, and Subject role types:

```
// Provide basic domain functionality of figures.
// Figures have origin, extent. Can be drawn, moved, and resized.
// Figures have a handle by which they are manipulated.
roletype Figure {
  point origin;
  rectangle extent;
  void draw(ref<Graphics> context);
  void drawOutline(ref<Graphics> context);
  void place(point location);
  void move(int dx, int dy);
  void resize(int handle, int dx, int dy);
}

// Provide parent reference; may be null.
// Parent object must be of type Parent.
roletype Child {
  ref<Parent> parent;
  boolean hasParent();
  void setParent(ref<Parent> parent);
}

// Manage objects registered as dependents.
// Notify them about state changes of the figure.
// Dependents must be of type Observer.
roletype Subject {
  collection<ref<Observer>> dependents;
  boolean hasObserver(ref<Observer> observer);
  void addObserver(ref<Observer> observer);
  void removeObserver(ref<Observer> observer);
}
```

Specification 3-3: Example specification of Figure class.

Please note that a role type in its definition refers to other types, and that these other types need not be class types. As discussed below, these other types are likely to be other role types from a role model. Examples are the definition of the Child role type that refers to a Parent role type and the definition of the Subject role type that refers to an Observer role type.

3.3.3 Class (revised definition)

With the introduction of the role and role type concept, we can now enhance the definition of the class and inheritance concepts. The object concept has already been enhanced in the previous subsection, stating that with any object, a set of roles is associated.

Definition 3-16: Class (revised)

A class is the definition of a (possibly infinite) set of objects, called its *instances*. A class defines a non-empty set of role types, a composition function, and a class type. The composition function, applied to all role types, results in the class type.

The class type specifies the behavior of the instances of the class.

The set of role types of a class is called the class' *role type set.* The role type set of a class determines which kind of roles its instances may play at runtime. An object x of class X may only play a role r if the role's type R is an element of the role type set of X.

Specification 3-4 defines the Figure class in terms of role types.

```
roletype Figure { ... } // For definition, see above.
roletype Child { ... } // For definition, see above.
roletype Subject { ... } // For definition, see above.

// Generically manage properties of figure.
// Examples properties are fill color, frame color, line stroke.
// Properties may be any kind of Object.
roletype Provider {
  collection<ref<Object>> properties;
  void hasProperty(string name);
  void getProperty(string name);
  void setProperty(string name, ref<Object> property);
  void unsetProperty(string name);
}

class Figure extends Object {
  roletype Figure;
  roletype Child;
  roletype Subject;
  roletype Provider;
  ... More definition.
}
```

Specification 3-4: Example specification of Figure class, including role types.

The role type set of the Figure class comprises the Figure, Child, Subject, and Provider role types. The role types are deliberately defined independently of the class. The class only makes use of them by specifying that they are elements of its role type set. The enclosing definition of a role type is always a role model (see below).

A class inherits the role type set of its superclass. Thus, with increasing distance from the root class Object, the size of the role type set increases monotonously. Assume that class Object is defined as follows:

```
class Object {
  roletype Readable; // Object can be read from a passive form.
  roletype Writable; // Object can be written to a passive form.
  roletype Instance; // Object provides metainformation.
  roletype Clonable; // Object can be cloned.
  roletype Comparable; // Object can be compared for equality.
  roletype Key; // Object can act as a key for dictionaries.
}
```

Specification 3-5: Example specification of Figure class.

The cardinality of the object role type set is 6. With Figure being a direct subclass of Object, the cardinality of its role type set is 6 plus the number of new role types it defines itself (of which we have seen 4 so far: Figure, Child, Subject, and Provider).

For the purposes of this dissertation, there is no need to introduce a subtyping relationship between role types. Thus, class inheritance is not affected, and role types of a class remain unchanged by subclasses.

3.3.4 Choice of type specification mechanism

A class type is the composition of all the role types from the class' role type set. The class defines a composition function that carries out this composition. Effectively, this is the integration of the state model of the role types to form a larger state model for the class.

This dissertation does not introduce a new type specification, but assumes that any mechanism is acceptable that fulfills the following properties:

- The mechanism lets developers express a type in an object-oriented fashion.
- The mechanism provides a concept of substitutability based on concept specialization.
- The mechanism lets developers specify dynamic behavior of objects.
- The mechanism lets developers compose (role) types to form new derived (class) types.

These properties are the result of the new class definition. Most of the aforementioned approaches address these issues and could be chosen as a type specification mechanism for role modeling (for example, [LW93a, LW93b, LW94]).

Another solution is not to choose a particular type specification mechanism, but to rely on a mainstream modeling language or programming language. (These are typically so weak with respect to type specification that they are considered here as not having one.) Developers then define the composition function of the role types implicitly. They realize the composition function through a class implementation.

3.3.5 Object collaboration task and role model (definition)

Object collaborations typically fulfill several tasks in parallel. The same objects collaborate to achieve several different things. Given any individual object, the role types of the object's roles express the behavior needed for the different tasks.

Definition 3-17: Object collaboration task

An *object collaboration task* is an object collaboration and a set of roles objects play in the collaboration. The object relationship graph must be non-partitioned.

An object collaboration task represents a single-purpose activity of objects in an object collaboration, which they carry out by playing the roles defined by the task.

An object collaboration task is said to be valid, if it conforms to a role model.

Definition 3-18: Role model

A *role model* is a set of role types that relate to each other by object relationship descriptions and role constraints. The role type relationship graph must be non-partitioned.

The role type relationship graph is the graph of all object relationship descriptions.

A role model defines a (possibly infinite) set of valid object collaboration tasks. A collaboration task from this set is said to conform to the role model. As discussed below, role models compose to become class models, and object collaboration tasks compose to become object collaborations.

Role models are the place where role types are defined. For example, the role model that describes the figure object hierarchy is called FigureHierarchy. Part of its definition are the role types Child and Parent, as defined above. Classes have to import these role types from the role model to put them into their role type set. We use the common dot-notation to identify role types: Child becomes FigureHierarchy.Child and Parent becomes FigureHierarchy.Parent in the context of a class.

The role type relationship graph in a role model may not be partitioned, so that there is a path from every role type to every other role type. This ensures that the role model is a cohesive model rather than a set of unrelated role types.

Before we provide an example, we need to define the concept of role constraint that lets us define how roles come together in an object (or not).

3.3.6 Role constraint (definition)

An object may play several roles at once, as defined by the role type set of its class. A role, an object plays, may require another role from that same object within the given object collaboration task. Or, a role may require that the same object within the collaboration task never plays another role. Or, two roles might mutually require each other. Finally, two roles may not have any requirements with respect to each other.

Such constraints are expressed as role constraints. (It is role constraint rather than role type constraint, because this descriptive means refers to roles objects play in an object collaboration task rather than to role types from a role model).

Definition 3-19: Role constraint

A *role constraint* is a value from the set {role-implied, role-equivalent, role-prohibited, role-dontcare}. For every given pair of role types (R, S) from a role model one such value is defined.

Role constraints are scoped by an object collaboration task. They only constrain the role-playing of objects within such a task. As a consequence, role constraints are only specified within the context of one role model.

The meaning of the role constraints is as follows:

a) A *role-implied* value for a pair of role types (R, S) defines that an object playing a role r defined by role type R is always capable of playing a role s defined by role type S. That is, role r implies role s. This relationship is transitive.

b) A *role-equivalent* value for a pair of role types (R, S) defines that an object playing a role r defined by role type R is always capable of playing a role s defined by role type S, and vice versa. That is, role r and role s imply each other. This relationship is symmetric and transitive.

c) A *role-prohibited* value for a pair of role types (R, S) defines that an object playing role r defined by role type R may not play role s defined by role type S within a given collaboration task. That is, role r prohibits role s for the task. This relationship is symmetric and transitive.

d) A *role-dontcare* value for a pair of role types (R, S) defines that an object playing a role r of role type R has no constraints with respect to another role s of role type S within the given collaboration task. The role s may or may not be available together.

Figure 3-17 shows the visual representation of role constraints. Each possible role constraint is depicted through its own visual symbol.

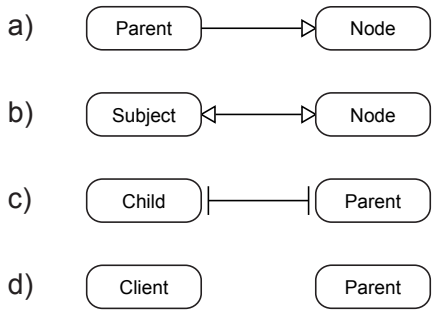

Figure 3-17: Examples of role constraints between role types.

Case a) shows a role-implied constraint, case b) shows a role-equivalent constraint, case c) shows a role-prohibited constraint, and case d) shows a role-dontcare constraint.

3.3.7 Figure role models (examples)

As a first example of a role model, consider the collaboration task of a figure object with its client. It can be described by a simple role model that defines the role types Figure and Client. The Figure role type depends on further types from a yet unspecified Graphics role model, which it imports.

Specification 3-6 describes the Figure role model:

```
// Figure requires Graphics role model.
import Graphics.*;

rolemodel Figure {
  // Provide basic domain functionality of figures.
  // Have origin, extent. Can be drawn, moved, and resized.
  roletype Figure {
     point origin;
     rectangle extent;
     void draw(ref<Graphics> context);
     void drawOutline(ref<Graphics> context);
     void place(point location);
     void move(int dx, int dy);
     void resize(int handle, int dx, int dy);
     ... More definition.
  }

  // The Client role type provides no operations.
  // However, it must behave properly when using Figure objects.
  roletype Client {
     ... // Specification of behavior with respect to using figures.
  }

  constraints {
     (*, *) = role-dontcare;
  }

  ... More definition, e.g., associations, cardinalities.
}
```

Specification 3-6: Example specification of Figure role model.

The wildcard ‘*’ in the role constraints part of the role model is a placeholder for any role type from the model. Thus, (*, *) expands to { (Figure, Client), (Client, Figure) }.

The Client role type is a no-operation role type. Within a role model, such no-operation role types define roles of objects that are not referenced and used by other objects (hence, no operations needed). However, a Client object still acts according to the no-operation Client role type.

The Client and Figure role types of the Figure role model are referenced from the outside using the common dot-notation for qualifying names: Figure.Client and Figure.Figure.

Figure 3-18 shows the Figure role model. Each role type in the role model prominently shows its name in large font and set below it, in parenthesis and a smaller font, the role model name. The role model name is added to distinguish role types from different role models in the context of class models (see below).

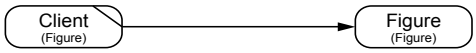

Figure 3-18: The Figure role model.

Another role model example is the collaboration between parent and child figure objects in the hierarchy. A parent figure manages its child figures, and a child figure provides access to its parent object.

Specification 3-7 shows the FigureHierarchy role model.

```
rolemodel FigureHierarchy {
  // Provide parent reference.
  roletype Child {
     ref<Parent> parent;
     boolean hasParent();
     void setParent(ref<Parent> parent);
  }

  // Manage child figures.
  roletype Parent {
     collection<ref<Child>> children;
     boolean hasChild(ref<Child> child);
     void addChild(ref<Child> child);
     void removeChild(ref<Child> child);
  }

  // Configures parent object with children.
  roletype Client {
     // Leave it to parent to set itself to child.
     // No further specification.
  }

  // A child may not be its own parent.
  // A child may not configure its parent.
  constraints {
     (Child, *) = role-prohibited;
     (*, Child) = role-prohibited;
     (Client, Parent) = role-dontcare;
     (Parent, Client) = role-dontcare;
  }

  ... More definition, e.g., associations, cardinalities.
}
```

Specification 3-7: Example specification of FigureHierarchy role model.

Figure 3-19 shows the FigureHierarchy role model.

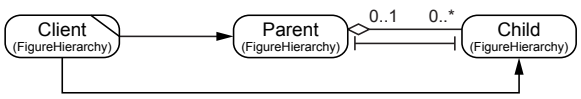

Figure 3-19: The FigureHierarchy role model.

Yet another example is the collaboration task for notifying clients about state changes of a figure object. Clients may depend on the state of the figure object and need to be informed about changes to it.

A previous subsection defines the role types FigureObserver and Subject. This subsection redefines them in the context of a role model. In particular, their names are adapted to avoid confusion.

Specification 3-8 defines the FigureObserver role model.

```
rolemodel FigureObserver {
  // Manage all dependent objects.
  // Notify them about state changes of the figure.
  // Dependents must be of type Observer.
  roletype Subject {
     collection<ref<Observer>> dependents;
     boolean hasObserver(ref<Observer> observer);
     void addObserver(ref<Observer> observer);
     void removeObserver(ref<Observer> observer);
  }

  // Provide callback operations for subject.
  roletype Observer {
     void update(ref<Subject> source, ref<Event> event);
  }

  ... Event definition.

  constraints {
     (*, *) = role-prohibited;
  }

  ... More definition, e.g., associations, cardinalities.
}
```

Specification 3-8: Example specification of FigureObserver role model.

Figure 3-20 shows the FigureObserver role model.

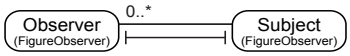

Figure 3-20: The FigureObserver role model.

As a final example, consider a role model used to pass client requests along a chain of objects. The client requests are converted into Request objects first, which are then forwarded from one element in the chain to its succeeding element. A figure object uses this mechanism to notify its parent about a client request that it could not handle.

Specification 3-9 describes the FigureChain role model:

```
rolemodel FigureChain {
  // Predecessor in a chain of figure objects.
  // Provides operation to generically handle requests.
  roletype Predecessor {
     ref<Successor> successor;
     void setSuccessor(ref<Successor> successor);
     void forwardRequest(ref<Request> request);
  }

  // Accept, queue, and dispatch requests.
  roletype Successor {
     void handleRequest(ref<Request> request);
     void handleDeleteRequest(ref<DeleteRequest> request);
     void handleInvalidateRequest(ref<InvalidateRequest> request);
  }

  ... Request definition.
```

```
  constraints {
     (*, *) = role-prohibited;
  }

  ... More definition, e.g., associations, cardinalities.
}
```

Specification 3-9: Example specification of FigureChain role model.

Figure 3-21 shows the FigureChain role model.

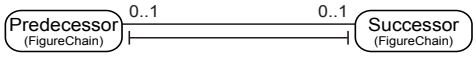

Figure 3-21: The FigureChain role model.

Please note that these role models are concrete design artifacts, and not design patterns. (They are instances of design patterns.) It is a conscious modeling decision to make the FigureHierarchy role model have a Client role type, but to omit this role type from the FigureChain role models. These role models are defined for and used in the Figure class model defined below, where the FigureHierachy role model is used to configure the object hierarchy at runtime. From the object hierarchy, the object chain is determined, and no option for chain configuration is given.

The role models Figure, FigureHierarchy, FigureObserver, and FigureChain all generically relate to figure objects. They are therefore used by the abstract Figure and CompositeFigure classes. In addition, role models specific to a certain figure class can be defined.

For each concrete figure class like RectangleFigure or UmlClassFigure, there is a role model that describes the figure specific collaboration with its clients. Also, there is typically a role model that defines how to create a new instance. This leads to role models RectangleFigure and RectangleFigureCreation, UmlClassFigure and UmlClassFigureCreation, etc. They are omitted here, because they are fairly simple, and do not add to the discussion.

3.3.8 Composing role models

It might seem natural to introduce a composition function for role models. Composed role models would be the composition of smaller role models, which could either be atomic or other composed role models. OOram, for example, is based on a role modeling composition scheme called role model synthesis [Ree96]. However, OOram, as originally defined, has no concept of class on the modeling level.

Role modeling as defined in this dissertation directly takes the step from atomic role models to classes and class models. I have found no need for intermediate composed role models. None of the case studies presented in Chapters 6 to 8 require the introduction of composed role models. Therefore, no such role model composition function is introduced.

Composition is always carried out in the context of a class model, where role types are assigned to classes, and where the classes provide the composition function to compose role types.

3.3.9 Class model (revised definition)

This subsection revises the definition of class model given earlier. Class models are understood as compositions of role models, in which classes provide and compose role types defined by the role models.

Definition 3-20: Class model

A class model is a set of classes and a set of role models. The classes relate to each other by inheritance and object relationship descriptions between role types. The class relationship graph must be non-partitioned.

The purpose of a class model does not change: it serves to model a set of valid object collaborations. The revised definition simply adds the concept of role model to describe how the classes relate to each other. In addition to using classes, inheritance, and object relationship descriptions, a class model uses role models for the description of the set of valid object collaborations.

Effectively, a class model composes the role models. Developers assign role types to classes and define how the role types are composed to form the class type.

Between the classes, the relationships, and the role models, a set of constraints applies:

- At least one role type from one of the role models must be in the role type set of a class. Otherwise, the class or the role model would not be connected with the rest of the class model.
- The role type set of a class must fulfill the role constraints set up by the role models. This requirement avoids invalid object collaborations (see section on assigning role types to classes).
- The object relationship descriptions between role types must match those between classes. In particular, the cardinality of the descriptions between role types must sum up to those of the classes (see below).
- The relationship graph between the classes may not be partitioned, so that there is a path from every class to every other class in the model. Otherwise, a class would not be connected with the rest of the model.

Each role model describes one particular object collaboration task of the valid object collaborations defined by the class model. The role model composition leads to the class model, which in turn does not only define one or several object collaboration tasks, but the full set of possible object collaborations.

3.3.9.1 Context and scope of definitions

We can now review how the concepts introduced so far relate to each other.

Everything is defined in the context of a class model. It is the outermost concept. In the context of a class model we define role models and classes. Role types are defined in the context of role models. Classes only import the role types; they do not define them.

Role types are defined independently of classes, because their primary purpose is to show how an object behaves within a specific collaboration task. Thus, they are always a part of a role model that describes the (set of valid) object collaboration tasks.

Defining role types and role models independently of a specific class lets us introduce reusable role models that can be used by different class models. An example is the ObjectProperty role model that defines the role types Client, Provider, and Property, and that can be used by different class models (see revised class model example).

3.3.9.2 Cardinality of object relationship descriptions

The cardinality of object relationship descriptions (associations and aggregations) between role types must sum up to the overall cardinality of object relationship descriptions between the classes of the role types.

Given a pair of classes A and B, let R_A be a role type of A and R_B a role type of B. For any object relationship description (R_A, R_B) that is defined by a role model, a cardinality is defined for R_A and R_B. Then the following must hold:

- The sum of the cardinality associated with all R_A's from the set of relationship descriptions (R_A, R_B) represents the cardinality of the object relationship description of class A with respect to class B.
- The sum of the cardinality associated with all R_B's from the set of relationship descriptions (R_A, R_B) represents the cardinality of the object relationship description of class B with respect to class A.

Most class models in this dissertation omit the object relationship descriptions between classes, because they can be derived from the object relationship descriptions between role types. Should there be a case where this statement does not seem to hold, a role model is probably missing. (Such a role model may be the simple role model of a Manager object being its own Client and managing a set of Elements in a collection.)

3.3.9.3 Assigning role types to classes

A class provides a set of role types to determine the possible roles its instances may play. How do role constraints between role types affect how a class may put role types into its role type set?

The use of the abstract superclass rule (a concrete class may not have subclasses) significantly eases the definition of these assignment constraints:

- A role-dontcare constraint has no effect on how a class may provide certain role types or not.
- A role-prohibited constraint also has no effect on how a class may provide some role types, because role constraints are restricted to single object collaboration tasks only. For different tasks, an object may well play roles defined by role types between which a role-prohibit constraint exists.

  For example, the role-prohibited constraint between the Observer and Subject role type of the FigureObserver role model denotes that an object may not observe itself. However, an object may well observe another object and be observed by yet another object. Hence a class may provide both role types. This second case uses two different collaboration tasks, even though they are instances of the same role model.
- A role-implied constraint between two role types (R, S) puts a constraint on the class hierarchy of a class X that provides role type R. Another class Y now provides role type S. The role constraint is maintained if one of the following cases holds:
  - X = Y;
  - Y is a superclass of X;
  - Every concrete subclass of X provides S (either directly or indirectly).

  These three cases ensure that for a given object that plays a role defined by R, a role defined by S is available.
- A role-equivalent constraint between two role types (R, S) puts a constraint on the class hierarchy of a class X that provides role type R. Assume another class Y, which provides role type S. The role constraint is maintained if one of the following cases hold:
  - X = Y;
  - if Y is a superclass of X, and if every concrete subclass of Y provides R (directly or indirectly);

– if X is a superclass of Y, and if every concrete subclass of X provides S (directly or indirectly).

These three cases ensure that for a given object that plays a role defined by R, a role defined by S is available, and that for a given object that plays a role defined by S, a role defined by R is available.

As a general rule, no constraint can prevent that a class provides a certain role type. However, role-implied and role-equivalent enforce the joint provision of role types according to the above specification.

3.3.10 Figure class model (revised example)

We can now describe the Figure class hierarchy as a class model based on role models. So far, we have described the class model without role models (Figure 3-13). We also have defined the role models Figure, FigureHierarchy, FigureObserver, and FigureChain.

We further need the ObjectProperty role model, which determines the collaboration of a client with a figure and its property objects. It defines the role types Client, Property, and Provider (of property). Specification 3-10 illustrates the ObjectProperty role model.

```
rolemodel ObjectProperty {
  // Anything may work as a property.
  // Thus, this is a no-semantics role type serving as a handle.
  // It can be assigned to any class, even after the class has been defined.
  roletype Property {}

  // Generically manage properties of figure.
  // Examples properties are fill color, frame color, line stroke.
  // Properties may be any kind of Object.
  roletype Provider {
     collection<ref<Property>> properties;
     void hasProperty(string name);
     void getProperty(string name);
     void setProperty(string name, ref<Property> property);
     void unsetProperty(string name);
  }

  // Get and set Properties to Provider.
  roletype Client {
     // Do not set Provider as a Property to itself.
     // No further specification.
  }

  // A Provider may not be have itself as a Property.
  // A Client may not be a Property.
  constraints {
     (Property, *) = role-prohibited;
     (*, Property) = role-prohibited;
     (Client, Provider) = role-dontcare;
     (Provider, Client) = role-dontcare;
  }

  ... More definition, e.g., associations, cardinalities.
}
```

Specification 3-10: Example specification of Figure class model.

Also, we have not defined the class-specific role models RectangleFigure, PolygonFigure, etc. These are simple binary role models, which define a Client and a Figure role type. The Figure role type defines basic services for the Client role type. The specification given below omits the details.

Specification 3-11 describes the Figure class model.

```
// Figure requires ObjectProperty and Graphics role model.
import Object.ObjectProperty;
import Graphics.Graphics;

classmodel Figure {
  rolemodel Figure { ... }
  rolemodel FigureHierarchy { ... }
  rolemodel FigureObserver { ... }
  rolemodel FigureChain { ... }

  class Figure extends Object {
     roletype Figure.Figure;
     roletype FigureHierarchy.Child;
     roletype FigureObserver.Subject;
     roletype FigureChain.Predecessor;
     roletype ObjectProperty.Provider;
     roletype Graphics.Client;
     ... More definition.
  }

  class CompositeFigure extends Figure {
     roletype FigureHierarchy.Parent;
     roletype FigureChain.Successor;
     ... More definition.
  }

  rolemodel RectangleFigure { ... }
  rolemodel RectangleFigureCreation { ... }

  class RectangleFigure extends Figure {
     roletype RectangleFigure.Figure;
     roletype RectangleFigureCreation.Creator;
     roletype RectangleFigureCreation.Product;
     ... More definition.
  }

  rolemodel PolygonFigure { ... }
  rolemodel PolygonFigureCreation { ... }
  class PolygonFigure extends Figure { ... }

  rolemodel TextFigure { ... }
  rolemodel TextFigureCreation { ... }
  class TextFigure extends Figure { ... }

  rolemodel GroupFigure { ... }
  rolemodel GroupFigureCreation { ... }
  class GroupFigure extends CompositeFigure { ... }

  rolemodel UmlClassFigure { ... }
  rolemodel UmlClassFigureCreation { ... }
  class UmlClassFigure extends CompositeFigure { ... }

  rolemodel UmlAssociationFigure { ... }
  rolemodel UmlAssociationFigureCreation { ... }
  class UmlAssociationFigure extends CompositeFigure { ... }

  ... More definition, e.g., associations, cardinalities.
}
```

Specification 3-11: Example specification of Figure class model.

Figure 3-22 shows the class model using role models. The figure shows class-level role types for the first time. Class-level role types are role types of the class object (see discussion below). They look like regular (instance-level) role types, but are shown with a rectangle rather than an oval as the bounding box.

A role type, which is put on top of a class, is an element of that class' role type set. The object relationship descriptions between classes are omitted to avoid cluttering up the figure. They can be derived from the relationship descriptions between the role types.

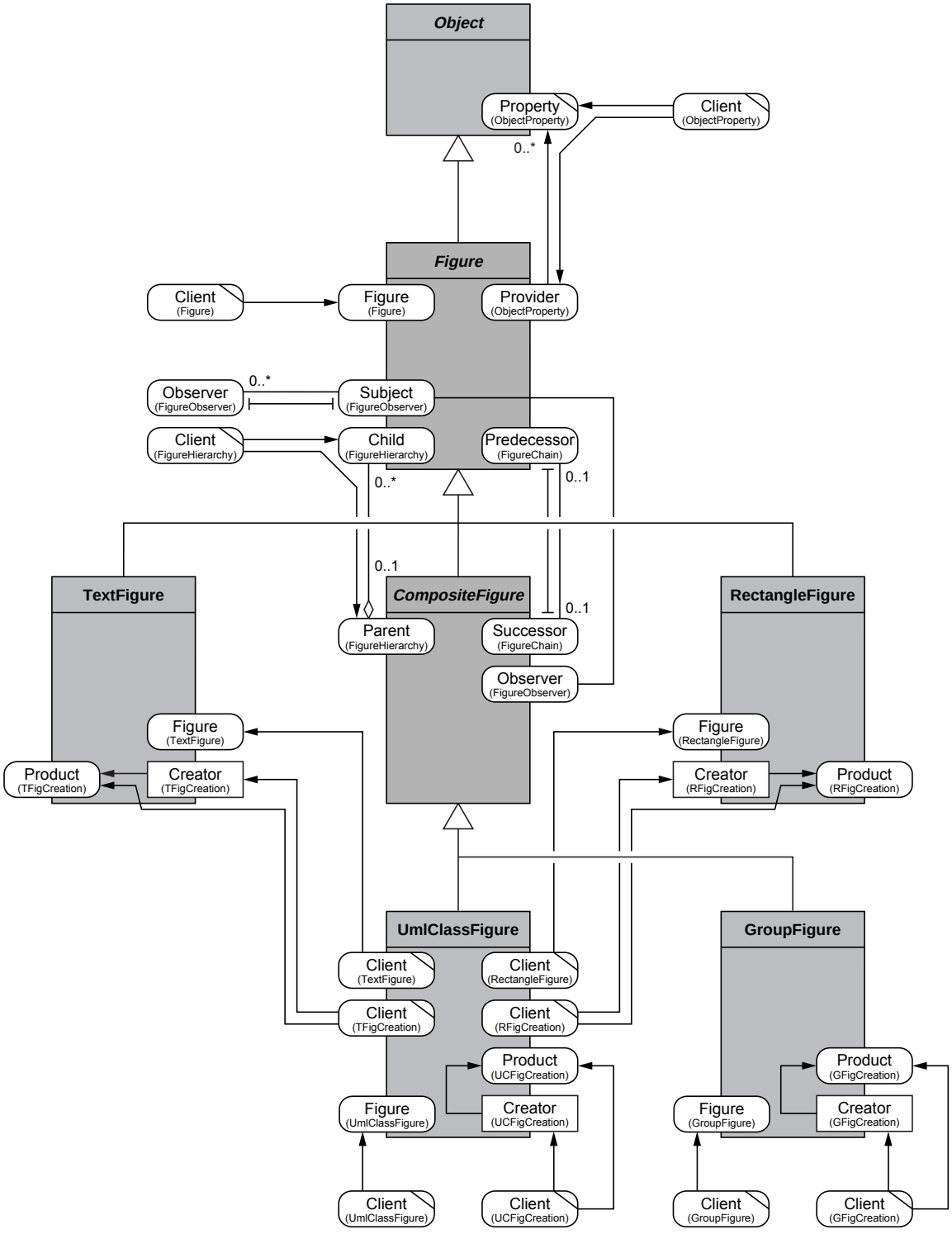

Figure 3-22: Class model of the example Figure class model, including all role models.

Figure 3-22 demonstrates a few interesting issues.

- We can see how a class assembles a set of role types and composes them. For example, the Figure class provides the Figure, Subject, Child, Provider, and Predecessor role types.
- Also, no object relationship descriptions between classes are needed, because they can be derived easily from the relationship description between role types.

- All important role models deal with the key classes Figure and CompositeFigure. The less important subclasses only provide simple binary domain-specific role models like RectangleFigure.
- The average subclass introduces two role models. One that provides its primary domain functionality (like RectangleFigure), and one that lets clients create instances of the class (like RectangleFigureCreation).

The Figure class model raises some issues that need more discussion.

3.3.10.1 Roles of class objects

Class objects are objects and therefore play roles. The class of a class is called its metaclass. The role types of a metaclass need a dedicated textual and visual notation to not screw up specifications and diagrams.

Textually, we tag the metaclass' role type using the keyword static, and put them next the role types of the class. Visually, in a class model, we draw the bounding box of a static role type as a rectangle, rather than as an oval. This distinction lets us put static role types right next to regular role types.

Consider the GroupFigureCreation role model for creating a GroupFigure object. A Client object calls the new operation on a Creator, the class object. The Creator creates the new Product object that it returns to the Client. The Creator and Product role types are made up by the different constructor and initialization operations.

Specification 3-12 shows the textual specification of the GroupFigureCreation role model.

```
rolemodel GroupFigureCreation {
  roletype Client {
     // No constraints.
  }

  roletype Creator {
     GroupFigure new();
     GroupFigure new(collection<ref<Figure>> elements);
     ... Possibly more.
  }

  roletype Product {
     initialize(collection<ref<Figure>> elements);
  }

  ... More definition, e.g., associations, cardinalities.
}

class GroupFigure extends CompositeFigure {
  static roletype GroupFigureCreation.Creator;
  roletype GroupFigureCreation.Product;
  ... More role types.
}
```

Specification 3-12: GroupFigure class and GroupFigureCreation role model.

Figure 3-23 shows the GroupFigureCreation role model visually.

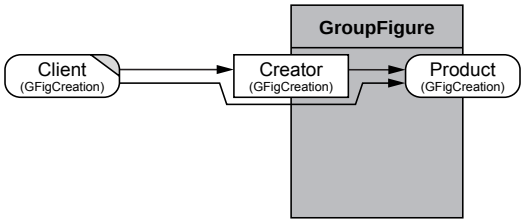

Figure 3-23: GroupFigure class with GroupFigureCreation role model.

Depending on the programming language used to implement this role model, different language features can be deployed to make it more convenient to use. For example, in Java, unless there are more specific requirements, Creator.new() should be mapped on new GroupFigure(), which may then internally call initialize() with an empty collection.

3.3.10.2 Boundaries of class model

The class model of Figure 3-22 presents only a part of what would be a realistic class model for a drawing editor. Basic graphical figure classes, like CircleFigure or TriangleFigure are missing. Also, domain-specific graphical figure classes are missing, like UmlInterfaceFigure, UmlInheritanceFigure, or UmlCollaborationFigure, as in the case of the UML drawing editor example.

If we added these classes, the class model would grow until it encompassed the whole system. The result would be a single large class model, which would be very inconvenient to use. Therefore, concepts to partition such a class model are needed. Categories, subsystems, and frameworks are examples of such concepts. As this work focuses on frameworks only, we omit the other concepts.

The next chapter shows how the class model can be partitioned into one framework and two framework extensions. The framework comprises the Figure, CompositeFigure, and GroupFigure classes. The first framework extension part encompasses the fundamental figure classes, and the second framework extension encompasses the application-specific figure classes.

3.3.11 Design patterns in role modeling

This dissertation is not about design patterns. However, design patterns permeate framework design. The Figure class model example shows several design pattern instances. The case studies in Chapter 6 to 8 show even more design pattern applications in the context of object-oriented frameworks. Therefore, it is helpful to show how design patterns relate to role models. Knowing patterns in their role model form lets us communicate design examples much faster than is possible without patterns.

3.3.11.1 Design patterns and design templates

A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts [RZ96]. An object-oriented design pattern is a pattern in the domain of object-oriented design. A pattern is frequently described as a problem/context/solution triple [GHJV95, BMR+96]. According to [GHJV95, p. 407], “design patterns identify, name, and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design. They identify classes, instances, their roles, collaborations, and the distribution of responsibilities.”

Despite much recent work on design patterns, many misunderstandings about patterns remain [Vli98]. Perhaps the most common and most harmful misunderstanding is to take the structure diagram of a design pattern description, for example from [GHJV95], as a rigid definition of the pattern. The structure diagram and the description of its participants are *an illustration of one common form of the pattern.* By no means does the structure diagram represent the one and only form of the pattern.

It is helpful to distinguish between the concept of design pattern and design template. A design pattern is an abstract idea that defies formalization and therefore precise definition. However, for any given pattern, we can define an infinite number of design templates that can be represented in a formal way. The specification of such a design template can be carried out using any appropriate formalism, making code generation and conformance checking of implementations possible.

A pattern is an abstract idea that can be illustrated in many different ways and that can be instantiated in an infinite number of ways. We can illustrate a design pattern using role models as much as we can

illustrate it using the class diagrams in [GHJV95]. We can also use both. The choice of a particular modeling technique depends on how well the presentation conveys the pattern idea to its readers.

In [Rie96a] I discuss the advantages of role modeling over traditional class-based modeling for illustrating design patterns. Almost all design patterns, in particular those from [GHJV95] can be illustrated using role models. The role model form of a pattern lets developers more easily understand the pattern structure and apply it in a new design than possible with traditional class-based modeling.

3.3.11.2 The Composite pattern as a role model (example)

The extended role modeling technique lets us illustrate design patterns in a more comprehensive way than traditional class-based modeling lets us do.

Consider the Composite design pattern [GHJV95]. The Figure class model uses an instance of this pattern in the form of the FigureHierarchy role model.

Figure 3-24 shows an illustration of the Composite pattern in role model form. It has two parts. The core part is the Client/Child/Parent role model that constitutes the Composite pattern. The second part is a separate NodeClient/Node role model that represents a domain functionality role model. This second role model is grayed out in the figure, because it does not directly belong to the pattern.

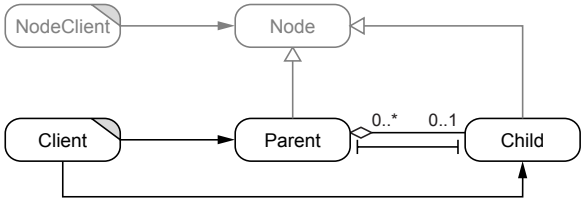

Figure 3-24: Illustration of the Composite pattern using role models.

The NodeClient/Node role model represents the domain functionality of the domain the pattern is applied in. The Client/Child/Parent role model represents the core idea of the pattern, namely that a Parent object can have several distinct Child objects and that the Parent object gets configured with its Child objects by a Client. The role-implied look-alike symbols transfer their role modeling meaning to the pattern level. It should be noted, however, that no precise semantics are underlying this pattern illustration. Here, the role-implied constraints state that any Child object and any Parent object always must be able to play the Node role.

Figure 3-25 shows the most common class model of the Composite pattern. It also shows how the role models are applied to this class model, thereby clarifying the different responsibilities of the classes.

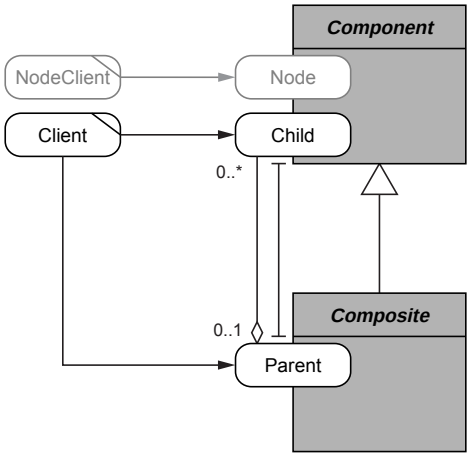

Figure 3-25: Illustration of the most common class model of the Composite pattern.

To drive home the point that role models, in conjunction with class models, illustrate design patterns more comprehensively than traditional class-based modeling, Figure 3-26 shows another class model that illustrates the Composite pattern equally well.

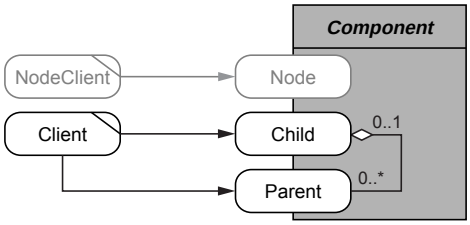

Figure 3-26: Illustration of an alternative class model for the Composite pattern.

Both class models can be formalized as design templates. They represent the pattern idea in a specific form. The first case, shown in Figure 3-25, covers the majority of the pattern applications, but the second case, shown in Figure 3-26, can also be found.

3.3.11.3 Further design patterns

In addition to the Composite pattern, the Figure example uses the Observer, Chain of Responsibility, and Property List pattern. They occur as the FigureObserver, FigureChain, and ObjectProperty role models.

Appendix D presents these and many other patterns in role model form. The case studies of Chapter 6 to 8 draw heavily on these patterns to describe framework designs.

3.3.12 Visual role model shorthands

The textual and visual presentation of class models can become complex, in particular if many role models are involved. However, conciseness of presentation is important, because it helps communicate the design structure more effectively than possible by a lengthy presentation (which is also needed).

Sometimes the same types of role models are composed in a class in always the same way. It is very inconvenient to elaborately specify and draw these compositions in all their detail. Common recurring compositions should be captured using a shorthand notation.

Consider a Drawing class. In a drawing editor application, there might be exactly one instance of the Drawing class, which represents the whole drawing. Assume that there is one central place where to get this object, and that this is the class object of the Drawing class. Therefore, the Drawing class offers a simple access operation to it (DrawingSingleton role model). Moreover, the Drawing object is instantiated only when the first access occurs. Thus, the Drawing class object plays the Client and Creator roles in a DrawingCreation role model.

Figure 3-27 shows the resulting visual specification.

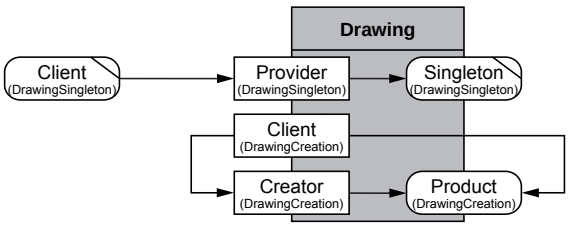

Figure 3-27: Drawing class with DrawingSingleton and DrawingCreation role models.

Because this kind of role model composition is such a common case that we abbreviate it using a shorthand. Figure 3-28 shows how the shorthand symbol looks like.

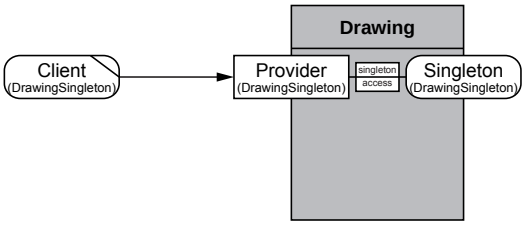

Figure 3-28: Drawing class with collapsed DrawingSingleton and DrawingCreation role model.

Unless a precise role type specification is required, it is sufficient for all practical purposes to use the terminology set up in Figure 3-28 to discuss this design. The case studies of this dissertation make use of this shorthand to save time and space and to avoid unnecessarily lengthy discussions.

Appendix C, the notation guide, presents further shorthands.

3.4 Summary

This chapter describes fundamental object and role modeling concepts needed for framework design. These concepts form the building blocks of all further concepts to come in this dissertation. They are needed to define higher-level concepts for framework design.

First, the chapter presents a concise summary of traditional class-based modeling concepts. It then enhances these concepts with new and adapted role modeling concepts. The new role modeling concepts add to the existing concepts and do not replace them.

The overall method becomes an evolutionary extension of existing object modeling approaches. They add to them where necessary rather than trying to replace them. Existing designs and documentations are not invalidated. However, using role modeling, they can be defined much more precisely and with greater detail.

The next chapter on framework design builds on this foundation to better describe object-oriented frameworks.

4

Framework Design

An object-oriented framework is a model of a particular domain or an important aspect thereof. It provides a reusable design and reusable implementations to clients. This chapter describes framework design and use from a role modeling perspective. A framework is described as a class model, whose free role types determine how it is to be used by use-relationship-based clients. A set of extension-point classes determines how the framework can be extended by inheritance. The chapter uses these definitions and results to discuss framework layering and framework documentation from a role modeling point of view.

4.1 Chapter overview

Chapter 2 puts framework into the wider perspective of software architecture. According to that perspective, object-oriented frameworks are design artifacts on a level above classes and role types. We therefore need explicit modeling concepts to represent them.

This chapter builds on the role modeling foundations laid in the previous chapter. It defines frameworks as class models that are used by its environment in specific ways. The environment of a framework comprises its use-relationship-based (black-box) clients, its inheritance-based (white-box) extension clients, and classes the framework builds upon for its own design and implementation.

The chapter is divided into five major parts.

- *Framework design.* This part introduces the concept of framework, its constituting parts, and their definitions. It provides the basis for the following subsections.

- *Black-box framework use.* This part discusses the use of frameworks from a black-box use-relationship-based perspective. It shows how role models serve as bridges between use-clients and a framework.
- *White-box framework extension.* This part shows how frameworks are customized for specific application domains using inheritance as extension mechanism.
- *Framework layering.* This part applies the role modeling concepts for framework design to the layering of class models, be they frameworks or application-specific framework extensions.
- *Framework documentation.* This final part draws some conclusions on how to document complex frameworks using role modeling.

The results of this chapter let us model object-oriented software systems with frameworks as elementary building blocks that provide well-defined interfaces to their environment.

4.2 Framework design

Frameworks are cohesive design and implementation artifacts. This section defines what a framework is and lists the hooks by which it is embedded into its environment. The environment comprises use-relationship-based (black-box) clients, inheritance-based (white-box) extension clients, and further classes the framework builds upon. Sections 4.3 and 4.4 discuss each aspect of a framework's environment in more detail.

4.2.1 Framework (definition)

A framework is a model of a particular domain or an important aspect thereof. A framework may model any domain, be it a technical domain like distribution or garbage collection, or an application domain like banking or insurance. A framework provides a reusable design and reusable implementations to clients.

Definition 4-1: Framework

A *framework* is a class model, together with a free role type set, a built-on class set, and an extension-point class set.

The core of a framework is a class model, as defined in Chapter 3. It is defined in terms of classes and role models. A role model may either be newly defined by the framework, or be imported from another framework, or be imported from a class library.

- The *free role type set* of a framework comprises those free role types of the class model that stem from role models that are defined by the framework. The free role type set excludes free role types from role models that are imported by the framework (see definition and discussion below).
- The *built-on class set* of a framework comprises those classes from other class models, frameworks, or framework extensions, which framework classes build upon (see definition and discussion of built-on classes below). They use free role types and free role models to do so.
- The *extension-point class set* of a framework comprises those framework classes, from which new classes of a framework extension may inherit (see definition of extension-point class set below).

Whether a role type is a free role type or not, and whether a class is an extension-point class or not, cannot be derived from some intrinsic properties of the role types or classes, nor can it be derived from the domain being modeled. It always is a conscious modeling decision.

A framework defines how objects collaborate with each other to represent the domain being modeled. A framework captures those aspects of a domain that are considered invariant over a set of object collaborations that represent concrete domain situations. A framework, as any other model, always focuses on specific aspects of the domain and ignores those deemed irrelevant for the modeling task.

The three key concepts of free role type, built-on class, and extension-point class serve to define how a framework connects to its environment.

Use-client classes are framework-external classes that use free role types to define how their instances make use of framework objects. Built-on classes are framework-external classes that are utilized by the framework for implementing its services. Extension-point classes serve to define how framework extensions custom-tailor the general domain model of the framework to an application-specific model.

As a consequence, a framework has two types of clients and may be a client of further frameworks itself:

- *Use-client class.* A use-client class is a framework-external class whose instances make use of framework objects via use-relationships.
- *Extension class.* An extension class is a subclass of an extension-point class of the framework. It makes use of a framework class via inheritance.

Each framework class may itself be a use-client class of a lower-layer framework class, for example a built-on class. Also, a framework class may inherit from an extension-point class of another framework and thereby become an extension class of the other framework. (In single rooted systems, this is always the case, with the Object framework being the only exception.)

Frameworks have been characterized as being either black-box or white-box [JF88]. A black-box framework is a framework that is used by means of object composition, and a white-box framework is a framework that is extended using inheritance. Most real-world frameworks are “gray-box” frameworks, allowing both uses of a framework.

4.2.2 Free role type (definition)

The free role types of a framework define how use-clients may make use of a framework. The set of free role types of a framework defines the full set of services that clients may access.

Definition 4-2: Use-client object

A *use-client object of a framework* is a framework-external object that makes use of one or more framework objects in an object collaboration task.

A framework-external object is an object that is an instance of a class that is not defined by the framework. (It may be an instance of a class from another framework, though.)

Definition 4-3: Use-client class

A *use-client class of a framework* is the class of a use-client object. It is connected to the framework through one or more role models.

Making use of a framework is defined in terms of the role models that connect client classes with framework classes. Client classes take on free role types of the framework's free role models.

Definition 4-4: Free role type

A *free role type of a framework* is a role type of a framework-defined role model that may be picked up by use-client classes by putting it into their role type sets.

A free role type can be represented as an interface or a protocol, given a programming language that offers these concepts. However, free role types, which are no-operation or even no-semantics role types, usually are not explicitly represented on the implementation level.

Definition 4-5: Free role model

A *free role model of a framework* is a framework-defined role model that has one or more free role types.

Definition 4-6: Free role type set

The *free role type set of a framework* is the set of all free role types of a framework.

Using a framework this way is called black-box use, because clients connect to the framework using object relationships only. A use-client class determines the roles its instances may play through the free role types it takes on. The section on black-box use of frameworks discusses free role types and their use further.

4.2.3 Built-on class (definition)

Built-on classes are classes the framework relies on to implement its services. The framework reuses these framework-external classes, which may stem from any kind of class model, that is class libraries, frameworks, or application-specific framework extensions.

Definition 4-7: Built-on object

A *built-on object of a framework* is a framework-external object that a framework object makes use of in an object collaboration task.

Definition 4-8: Built-on class

A *built-on class of a framework* is the class of a built-on object. It is connected to the framework through one or more role models.

Definition 4-9: Built-on class set

The *built-on class set of a framework* is the set of all built-on classes of the framework.

The framework class is said to *build upon* the framework-external (built-on) class.

How do framework classes make use of built-on classes? The built-on class is part of a (built-on) class model that defines the role model through which another class may use the built-on class. Such a role

model is always a free role model of the class model being built upon, and the role type the framework class takes on is always a free role type of this role model.

A built-on class set is better suited for describing a framework's dependencies than a "built-on role model set," because it helps prevent unexpected behavior of instances of built-on classes. By relying on a specific class rather than a role type, a framework class is given a full behavioral specification of a built-on class rather than just one partial aspect as described by a single role type.

4.2.4 Extension-point class (definition)

A white-box framework serves as scaffolding for framework extensions. A framework extension is a set of classes, some of which inherit from framework classes. The framework models the domain on an abstract level, and a framework extension customizes this general domain model for a particular application. Framework extensions are discussed in a later section. Here, extension-point classes are defined, which are the (hook) classes that a framework extension relies upon.

Definition 4-10: Extension-point class

An extension-point class is a framework class that may be subclassed by framework-external classes.

Definition 4-11: Extension-point class set

The extension-point class set of a framework is the set of all extension-point classes of the framework.

The definition of extension-point classes is crucial for reusing classes through inheritance. Experience shows that only those classes, which have been prepared for reuse, can actually be reused in an effective way. A developer of a class must take into account how a new subclass inherits from the class when designing it. Only those classes that have been prepared for being reused should be declared extension-point classes of a framework.

The subsection on framework extension (see below) defines what framework extensions are and how they are handled in the context of framework design and layering.

4.2.5 Figure and Graphics framework (examples)

This section presents the Figure and the Graphics frameworks, as taken from Chapter 3.

4.2.5.1 Figure framework

The framework's core consists of the classes Figure, CompositeFigure, and GroupFigure. These classes are independent of any particular drawing editor application, and therefore serve well as part of a framework.

Specification 4-1 describes the framework.

```
// Figure imports the Common.ObjectProperty
// and Graphics.Graphics role model.
import Common.ObjectProperty;
import Graphics.Graphics;

framework Figure {
  public rolemodel Figure { ... }
  public rolemodel FigureHierarchy { ... }
```

```
  public rolemodel FigureObserver { ... }
  protected rolemodel FigureChain { ... }

  public abstract class Figure extends Object {
     roletype Figure.Figure;
     roletype FigureHierarchy.Child;
     roletype FigureObserver.Subject;
     roletype FigureChain.Predecessor;
     roletype ObjectProperty.Provider;
     roletype Graphics.Client;
     ... More definition.
  }

  public abstract class CompositeFigure extends Figure {
     roletype FigureHierarchy.Parent;
     roletype FigureChain.Successor;
     ... More definition.
  }

  rolemodel GroupFigure { ... }
  rolemodel GroupFigureCreation { ... }

  public class GroupFigure extends CompositeFigure {
     roletype GroupFigure.Figure;
     roletype GroupFigureCreation.Creator;
     roletype GroupFigureCreation.Product;
     ... More definition.
  }

  // Free role type set of framework.
  freeroletypes {
     Figure.Client;
     FigureHierarchy.Client;
     FigureObserver.Observer;
     ObjectProperty.Client;
     GroupFigure.Client;
     GroupFigureCreation.Client;
  }

  // Extension-point class set of framework.
  extensionpoints {
     Figure;
     CompositeFigure;
  }

  // Built-on class set of framework.
  builtonclasses {
     Graphics.Graphics;
  }
}
```

Specification 4-1: Specification of the Figure framework.

Figure 4-1 shows the visual representation of the framework. A light-gray background identifies free role types, while regular non-free role types have a white background.

The specification and its visual representation show all three parts of the framework.

- *Class model.* The class model of the framework comprises the classes Figure, CompositeFigure, and GroupFigure. It also defines several role models like Figure, FigureHierarchy, and FigureObserver. In addition, the Figure framework imports the Common.ObjectProperty and Graphics.Graphics role model.
- *Free role type set.* The free role type set of the framework comprises the role types Figure.Client, FigureHierarchy.Client, FigureObserver.Observer, ObjectProperty.Client, GroupFigure.Client, and GroupFigureCreation.Client. The free role types define how the framework is to be used, as illustrated below.

- *Extension-point class set.* The extension-point class set of the framework comprises the classes Figure and CompositeFigure. Extension classes of a framework may inherit only from extension-point classes. GroupFigure is not an extension-point class, because it is a concrete class not prepared for subclassing.
- *Built-on class set.* The built-on class set of the framework contains the Graphics class from the Graphics framework. The Figure class builds upon it through the Graphics.Client role type.

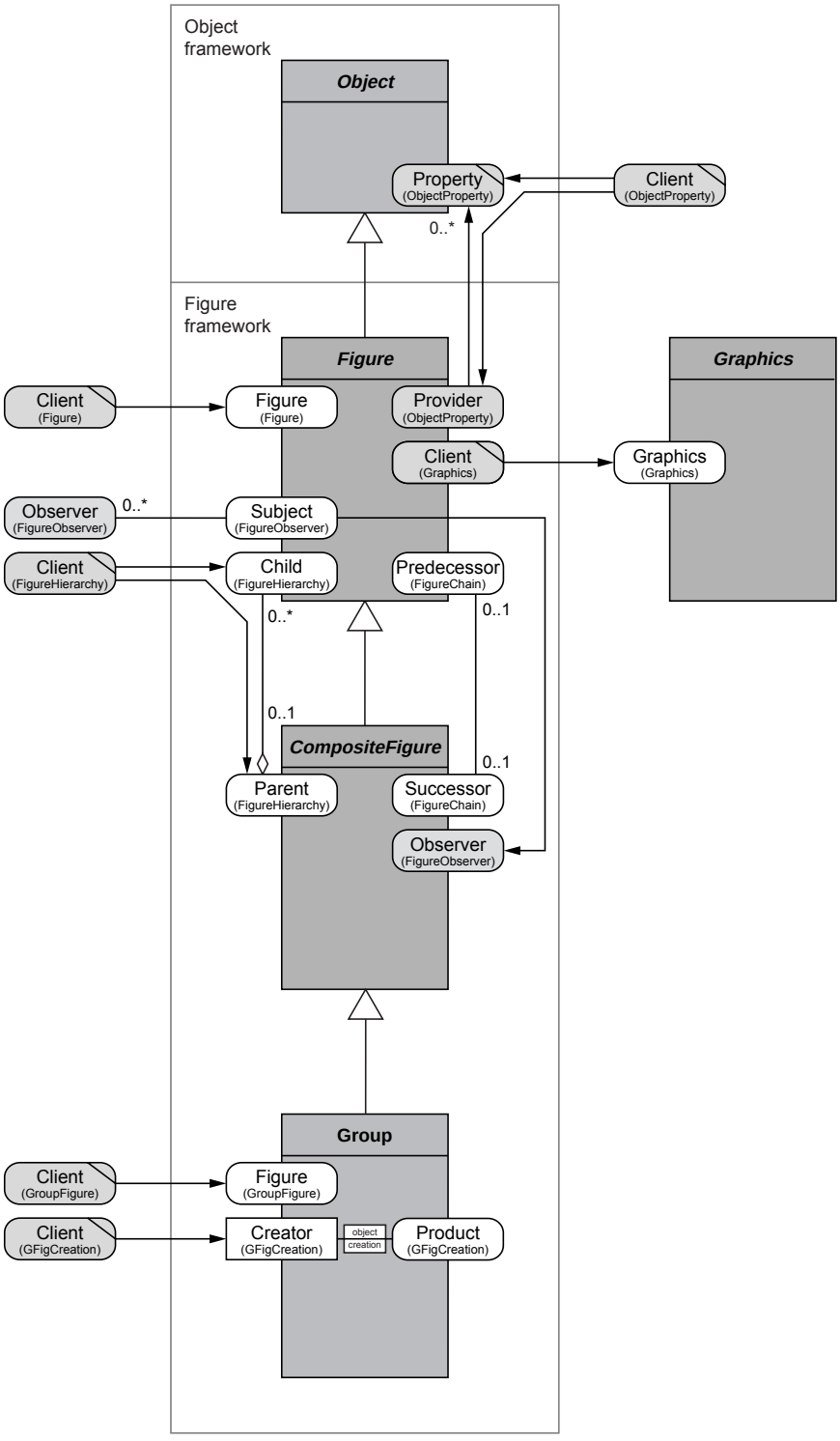

Figure 4-1: Class model of the Figure framework.

From the Graphics framework point of view, Figure is a use-client class. Figure uses the Graphics framework by putting one of its free role types, the Graphics.Client role type, into its role type set. Thus, frameworks can recursively build on each other.

4.2.5.2 Graphics framework

As a second framework example, consider the Graphics framework. It provides the Graphics, Image, Font, and Polygon class. The Graphics class lets clients draw figures and text onto a graphics context. A graphics context may be anything from a drawing area on the screen up to a printing device. The Image, Font, and Polygon classes are used to represent images, fonts, and polygons.

Each of the different tasks involved is described as a role model. Specification 4-2 shows the resulting definition of the Graphics framework.

```
framework Graphics {
  public rolemodel Graphics { ... }
  public rolemodel Clipping { ... }
  public rolemodel Imaging { ... }
  public rolemodel Texting { ... }
  public rolemodel Polylining { ... }

  public abstract class Graphics extends Object {
     roletype Graphics.Graphics;
     roletype Clipping.Graphics;
     roletype Imaging.Imager;
     roletype Texting.Texter;
     roletype Polylining.Polyliner;
     ... More definition.
  }

  public rolemodel ImageCreation { ... }

  public abstract class Image extends Object {
     roletype Imaging.Image;
     roletype ImageCreation.Creator;
     roletype ImageCreation.Product;
     ... More definition.
  }

  public rolemodel FontCreation { ... }

  public abstract class Font extends Object {
     roletype Texting.Font;
     roletype FontCreation.Creator;
     roletype FontCreation.Product;
     ... More definition.
  }

  public rolemodel PolygonCreation { ... }

  public class Polygon extends Object {
     roletype Polylining.Polygon;
     roletype PolygonCreation.Creator;
     roletype PolygonCreation.Product;
     ... More definition.
  }

  // Free role type set of framework.
  freeroletypes {
     Graphics.Client;
     Clipping.Client;
     Imaging.Client;
     Texting.Client;
     Polylining.Client;
     ImageCreation.Client;
     FontCreation.Client;
     PolygonCreation.Client;
  }
```

```
   // Extension-point class set of framework.
   extensionpoints {
      Graphics;
   }

   // Built-on class set of framework.
   builtonclasses {
      // Empty set (uses native API).
   }
}
```

Specification 4-2: Specification of the Graphics framework.

Figure 4-2 shows the class model of the Graphics framework.

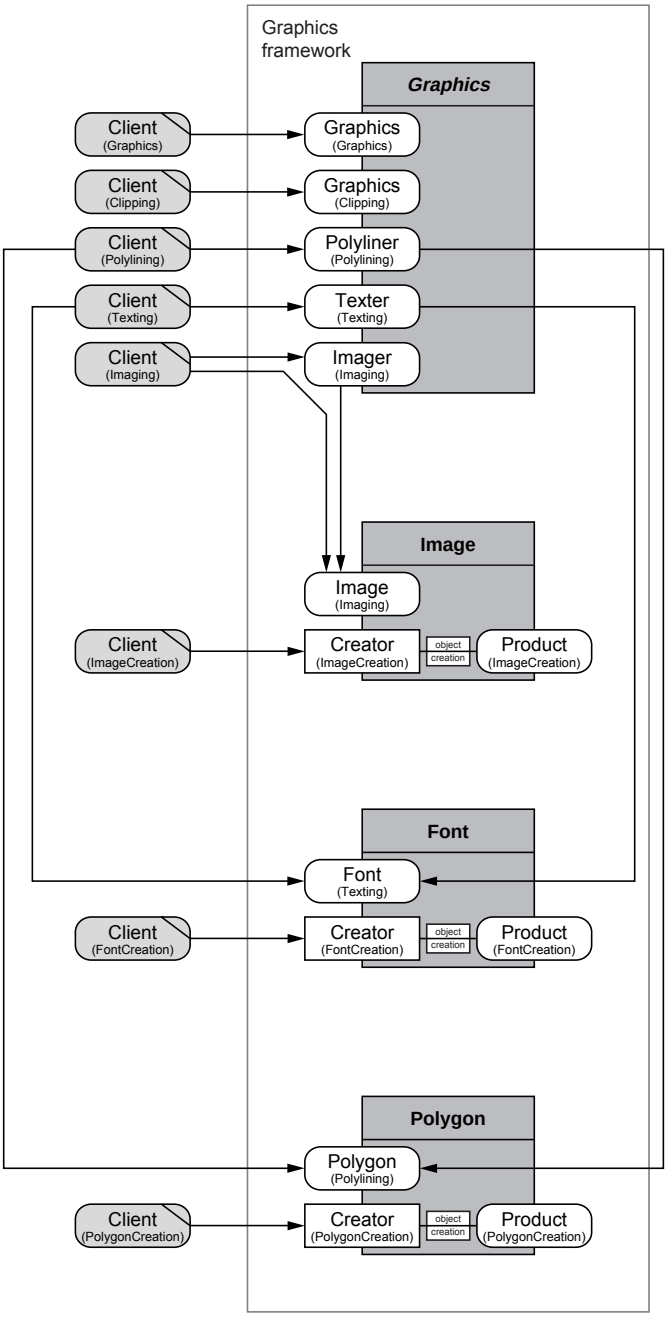

Figure 4-2: Class model of the Graphics framework.

The class model of the Graphics framework comprises the classes Graphics, Image, Font, and Polygon. Its free role type set comprises the Client role types of the creation role models and the Client role types of the domain functionality role models. Its built-on class set is empty, because the classes are implemented using a native API. The extension-point class set comprises the Graphics class only.

4.3 Framework use

Use-client objects of a framework use framework objects during object collaboration tasks described by free role models. Free role types from these role models determine the behavior of a use-client object. Free role types and free role models are the key (primitive) means to describe how clients may make use and eventually do make use of a framework.

4.3.1 Direct coupling through free role models

A use-client class makes use of a framework by putting one or more of its free role types into its role type set. *There is no other way of using a framework.* By accepting a role type, the class declares that its implementation and consequently the behavior of its instances conforms to the role type specification.

Use-client classes are coupled by static typing with the framework class of their interest, which is why we call this coupling mechanism *direct coupling* (as opposed to role object coupling, discussed below).

Effectively, *a free role model acts as the bridge between a client and a framework.* It represents the contract to which both the use-client and the framework promise to conform. The use-client-side and the framework-side role types represent the hooks by which the role model ties in the different classes.

Free role models, used this way, let framework developers specify the behavior required of client classes. This helps to avoid framework misuse. The strength of this help depends on how expressive the type specification mechanism is and how strongly conformance of an implementation to a role type or class specification can be ensured or checked for.

The class-based equivalent to using role types is to define classes that formalize use-client behavior, and then to make use-client classes inherit from them. This approach has the disadvantage of either requiring multiple inheritance or having an explosion of the number of classes. It is much better to use role types, which are precisely those lightweight entities that define one particular aspect of connecting to a framework, and nothing else. It is up to the use-client class to pick up those role types relevant to it.

This type of coupling can be applied recursively between frameworks. In particular does a framework build on its built-on classes through the use of free role models. A framework class picks up a free role type from a free role model through which it wants to connect to a built-on class. From the built-on class' point of view, the framework class is a use-client class, and the free role model acts as the bridge between them.

4.3.2 Examples of direct coupling

This subsection considers two examples of direct coupling:

- *Coupling of Editor, Figure, and Graphics framework.* The Editor use-client class is directly coupled with the Figure framework, which in turn is directly coupled with the Graphics framework.
- *Coupling of Editor and KidsEditor extension with Figure and KidsFigures extension.* The KidsEditor use-client class is directly coupled with the KidsFigures extension of the Figure framework.

4.3.2.1 Editor use-client of Figure framework

For the first example, consider the basic handling of figures through an Editor application class. The Editor class is a use-client class of the Figure class. Figure 4-3 shows how it ties in with the framework using its free role types. Also, the framework diagram shows how the Figure class ties in with the Graphics class, using a free role model. The free role models are highlighted in the diagram through a surrounding red box (dark gray box in a gray-scale printout).

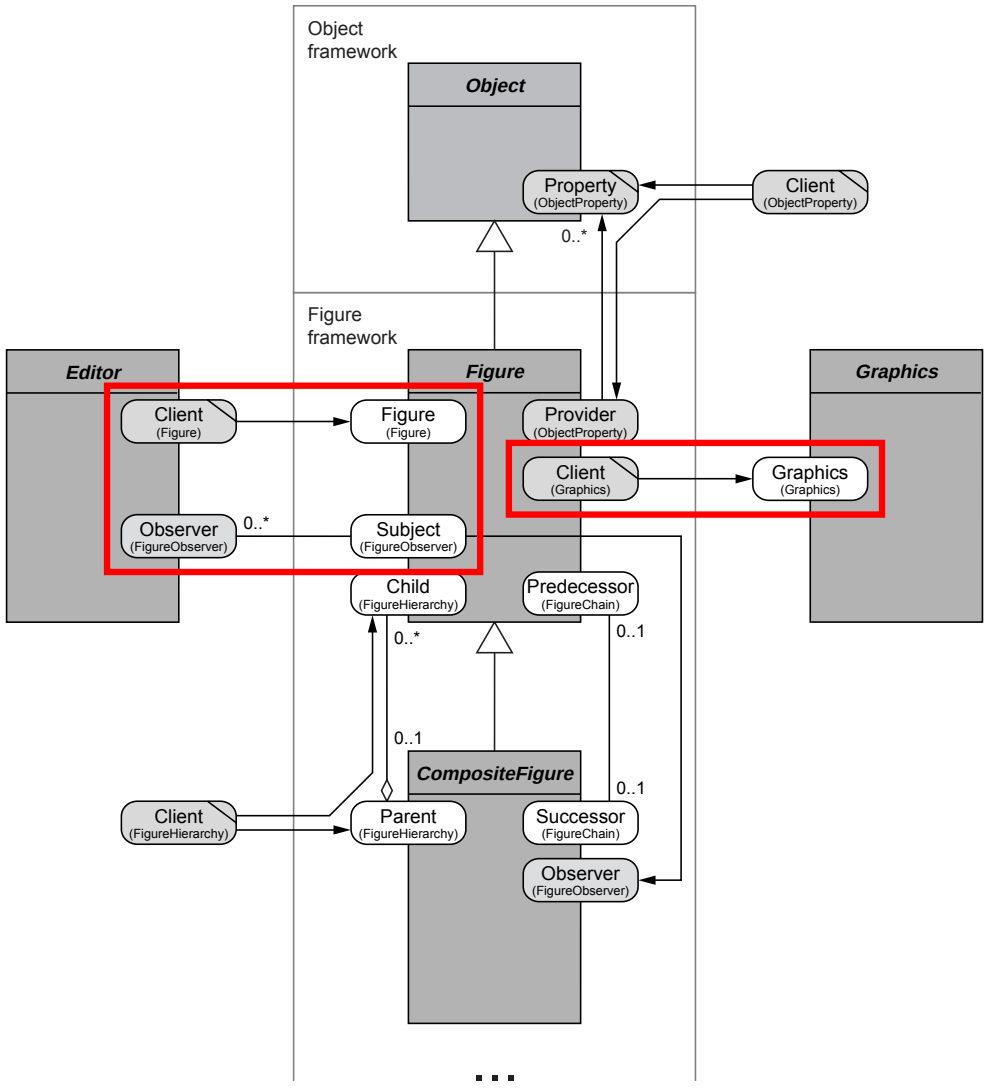

Figure 4-3: Editor example of making use of the Figure framework.

As can be seen, class Editor picks up the Figure.Client and the FigureObserver.Observer role type. Thus, the Figure role model directly couples the Editor use-client class with the Figure framework.

Also, the Figure class picks up the Graphics.Client role type. Thus, the Graphics role model directly couples the Figure framework with the Graphics framework.

4.3.2.2 KidsEditor use-client of Figure framework

As the second example, consider a drawing editor for children, called KidsEditor. It makes use of the Figure framework to provide children with basic graphical figures like rectangles and circles, but also with more complex attention-grabbing figures like “aquarium figures” and “circus figures”. Aquarium figures show animated fishes, and circus figures show juggling acts.

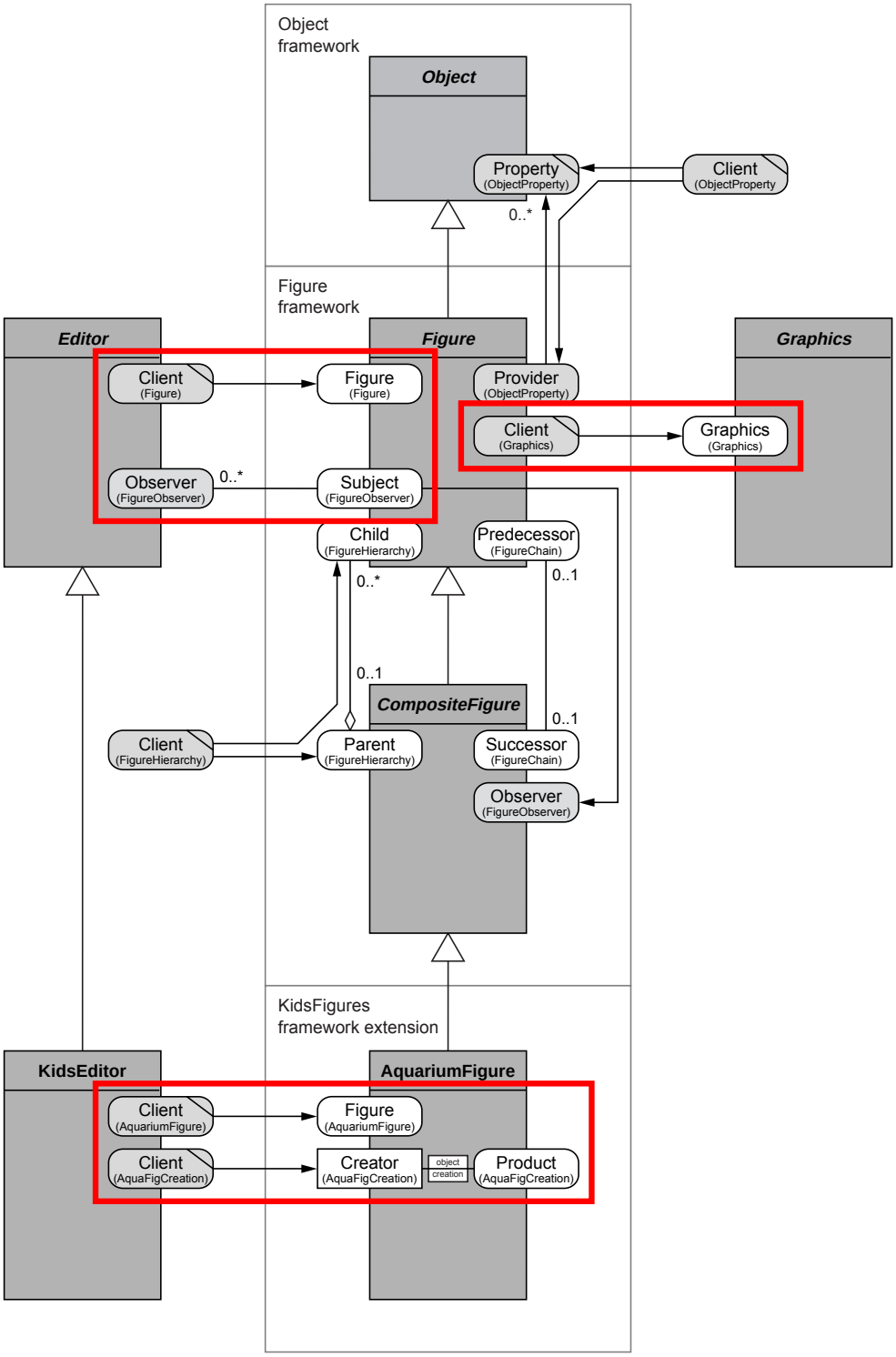

Figure 4-4: KidsEditor example of using the Figure framework and the KidsFigures framework extension.

These complex figures are instances of classes like AquariumFigure and CircusFigure. They are extension classes provided as parts of the KidsFigure framework extension of the Figure framework. (See the section on white-box framework extension for the definition of framework extensions.)

Here, the focus is on how a use-client class makes use of the Figure framework in the face of inheritance and framework extensions.

The KidsEditor class both picks up new free role types and inherits free role types of its Editor superclass. The full direct coupling of class KidsEditor with the Figure framework and its KidsFigures extension is therefore defined by the inherited role types Figure.Client and FigureObserver.Observer, and by the newly acquired role types AquariumFigure.Client, AquariumFigureCreation.Client, etc.

4.3.3 Properties of free role types

Free role types can be classified along two dimensions: the extent to which they constrain a class taking on the role type; and whether they come with operations that must be provided by the class (callback role types). These two dimensions are not orthogonal, both can be considered to be "just" an issue of proper type specification. However, in practice it is helpful to distinguish these two issues.

- *Free no-semantics role type.* A free role type that imposes no constraints on a class that picks it up is a no-semantics role type. Its specification is empty. Typically, vanilla Client role types are of this kind. However, one needs to be aware of hidden assumptions that are obvious to the experienced framework user (and are therefore not specified as part of the role type), but which are non-obvious to a novice.

  An example of a free no-semantics role type is the Figure.Client role type that lets use-client objects use Figure objects in arbitrary ways (within the context of the Figure role model). This example assumes that a Figure object is self-sufficient and does not rely on the client to adhere to some specification, in case of which the free role type would be a constraining role type.

- *Constraining free role type.* A constraining role type is a role type that comes with a non-empty behavioral specification. Typically, the specification requires that an object adhering to this role type must observe some protocols when collaborating with other objects, for example it may call some operations only in a certain order (like calling hide only after a preceding draw) [HS88].

  An example frequently found are complex initialization protocols, in which some operations may only be called by clients if other operations have been called before to set up parts of the object being initialized.

Traditionally, initialization protocols or other specifications are attached to the class whose instances are to be initialized according to that protocol. It is interesting to note that the role modeling view of specifying behavior leads to attaching the behavioral specification to the client of a class being initialized. This reflects more adequately that observing protocols are constraints on clients rather than on the class whose instances are to be initialized using this protocol.

- *Free role type with operations (callback role type).* Not every role type, be it constraining or no-semantics, must come with operations of its own. Only the minority of free role types has operations. Figure.Client has no operations. However, FigureObserver.Observer has.

  Typically, the operations of a free role type are callback operations that a framework or a class model requires to connect back to its clients. Instances of the Observer pattern (like the FigureObserver role model) probably account for the majority of such role models.

Role types with operations serve to transfer control back from the framework to a client. This control transfer is an important design issue, leading to the definition of callback role types.

Definition 4-12: Callback role type

A *callback role type* is a free role type of a framework that has a non-empty set of operations.

Callback role types are important, because having to define operations has the most visible impact on the implementation of a framework's class model in a particular programming language. For example, in Java a callback role type is typically represented as an interface, so FigureObserver.Observer becomes the Java interface FigureListener. Free role types that are not callback role types usually do not require an interface of their own.

4.4 Framework extension

Next to black-box use-relationship-based clients, frameworks also have white-box inheritance-based clients. Framework extensions are sets of classes that inherit from framework classes. Doing so, they customize the domain concepts represented by the framework classes to a more specific use, for example, in a specific application. This section defines what framework extensions are and how they are used.

4.4.1 Domains and applications

As discussed, a framework models a specific domain or a pertinent aspect thereof. The domain model represented by a framework may be specific enough to be directly usable by some use-clients. In this case, the framework is used as a black-box framework, and no need arises to customize the model to more application-specific concepts.

Frequently, however, and often desirable, a framework captures only the common parts of a domain model and leaves the introduction of application-specific model elements to the respective applications. Thus, each application must (be able to) introduce its own customization of the framework. This is done through framework extensions.

4.4.2 Framework extension (definition)

A framework extension is the specialization of precisely one framework. Its purpose is to customize the extended framework's domain model for a (more) specific purpose, either an application or another framework.

Definition 4-13: Extension class

An extension class of a framework is a subclass of an extension-point class of a framework.

Definition 4-14: Framework extension

A framework extension is a set of classes. Each class is either an extension class of the framework or a class that is transitively connected with at least one extension class through a role model.

A framework extension is said to *extend the framework* of which its classes inherit from.

A framework extension may either be a framework or a domain-specific or application-specific extension that is not a framework. In the first case, the extension can be extended further. In the last cases, the extensions are no frameworks, and form the leafs of the hierarchy of framework extensions.

Definition 4-15: Domain-specific framework extension

A domain-specific framework extension is a framework extension that is not a framework, but that can be used by different applications in the same domain.

Definition 4-16: Application-specific framework extension

An application-specific framework extension is a framework extension that is not a framework and that can be used by one specific application only.

A framework that allows for customization through framework extensions is said to be a *white-box* framework [JF88]. It must have a non-empty extension-point class set; otherwise it is a pure black-box framework.

On this framework level, we can see an analogy to the abstract superclass rule for class hierarchies, which I call the abstract framework rule. In theory, frameworks can only be abstract (white-box frameworks) and be prepared for extension through inheritance. So-called black-box frameworks are actually concrete framework extensions that are either domain or application-specific (and that are no frameworks).

In practice, we find the same shortcut that we find applied to the abstract superclass rule. A gray-box framework is a combination of a (white-box) framework, prepared for extension, and a (black-box) domain-specific framework extension, that cannot be extended further. Packaging the framework and a default extension together and calling it a gray-box framework is a matter of convenience. Done right, it does not violate the abstract framework rule.

Much like a framework, a framework extension defines a free role type set for use-clients that want to make use of the framework extension. It also defines a built-on class set to define the classes it builds upon. The use of these concepts is analog to their use for a framework. A framework extension inherits these sets from the framework it extends. It may only add to them; it may not remove a role type from a class or a free role type from the free role type set, etc. This ensures substitutability on the model level.

In principle, a framework extension could be allowed to draw on several different frameworks and extend them. This is analog to the use of multiple inheritance in class hierarchy design; it is also equally problematic. Extending several frameworks at once, in particular if multiple inheritance is used, typically indicates a sub-optimal understanding of the application domains. A mature framework extension extends only one framework.

Frameworks that are based on two or more frameworks or framework extensions compose them rather than extend them. Such a framework is still an extension of some other framework, but it uses and combines the other frameworks or framework extensions as part of their built-on class set. The situation is analog to using object composition over than multiple inheritance. For practical purposes, one might make a framework the extension of several other frameworks, but over time, the framework is likely to evolve into the extension of one framework that uses several others, even if these other frameworks or framework extensions are specifically made for the framework.

4.4.3 Figure and SimpleFigures framework extensions (examples)

This subsection discusses two examples of framework extensions:

- *Object framework extension.* The Figure framework is an extension of the Object framework. The example introduces the Object framework and discusses the Figure framework as its extension.
- *Figure framework extension.* The SimpleFigures framework extension is a framework extension of the Figure framework. It is not a framework itself, though.

The KidsFigures example of the section on framework use is a third example of a framework extension, in this case an application-specific framework extension. It is not discussed further here.

4.4.3.1 Figure extension of Object framework

Most systems come with a fundamental Object framework that determines what can be done with any kind of object. Java and Smalltalk have an explicit Object framework, while C++ has an implicit one.

Figure 4-5 shows an Object framework similar to the one of Java. It has been simplified significantly. The upper right "…" role model represents a general placeholder for role models that have been omitted from the figure.

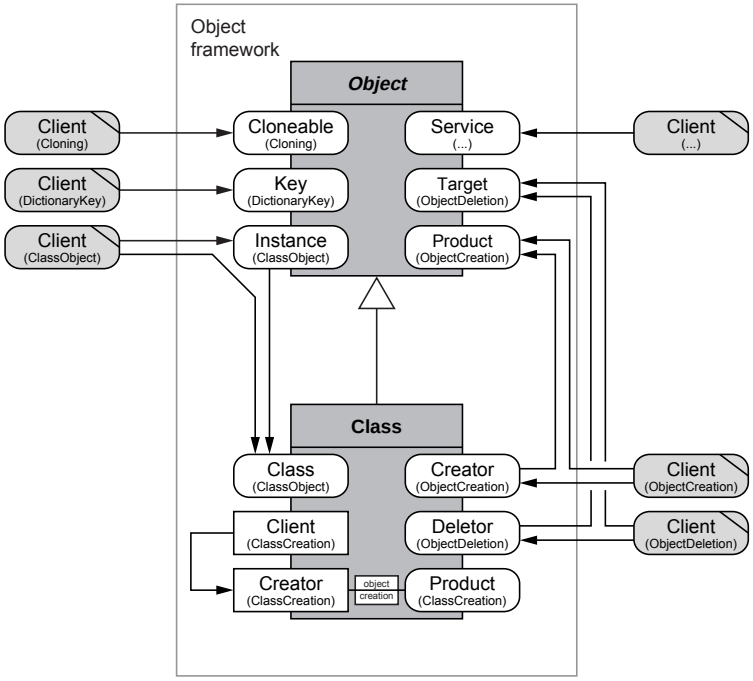

Figure 4-5: Object framework.

Specification 4-3 presents the textual specification of the framework.

```
framework Object {
  public rolemodel Reader { ... }
  public rolemodel Writer { ... }
  public rolemodel ClassObject { ... }
  public rolemodel Cloneable { ... }
  public rolemodel Comparable { ... }
  public rolemodel DictionaryKey { ... }
  public rolemodel ObjectCreation { ... }
  public rolemodel ObjectDeletion { ... }
  ... More role model definitions.

  public abstract class Object {
     roletype Reader.Readable;
     roletype Writer.Writable;
     roletype ClassObject.Instance;
     roletype Cloneable.Cloneable;
     roletype Comparable.Comparable;
```

```
      roletype DictionaryKey.DictKey;
      roletype ObjectCreation.Product;
      roletype ObjectDeletion.Target;
      ... More definition.
   }

   public class Class extends Object {
      roletype ClassObject.Class;
      roletype ObjectCreation.Creator;
      roletype ObjectDeletion.Deletor;
      static roletype ClassCreation.Client;
      static roletype ClassCreation.Creator;
      roletype ClassCreation.Product;
      ... More definition.
   }

   // Free role types of framework.
   freeroletypes {
      Reader.Client;
      Writer.Client;
      ClassObject.Client;
      Cloneable.Client;
      Comparable.Client;
      DictionaryKey.Client;
      ObjectCreation.Client;
      ObjectDeletion.Client;
      ClassCreation.Client;
   }

   // Extension-point class set of framework.
   extensionpoints {
      Object;
   }

   ... More definition.
}
```

Specification 4-3: Specification of Object framework.

Specification 4-4 now defines the Figure framework as an extension of the Object framework.

```
// Figure imports the Common.ObjectProperty
// and Graphics.Graphics role model.
import Common.ObjectProperty;
import Graphics.Graphics;

framework Figure extends Object {
   ... Role model definitions.

   public abstract class Figure extends Object {
      roletype Figure.Figure;
      roletype FigureHierarchy.Child;
      roletype FigureObserver.Subject;
      roletype FigureChain.Predecessor;
      roletype ObjectProperty.Provider;
      roletype Graphics.Client;
      ... More definition.
   }

   freeroletypes {
      Figure.Client;
      FigureHierarchy.Client;
      FigureObserver.Observer;
      ObjectProperty.Client;
      GroupFigure.Client;
      GroupFigureCreation.Client;
   }

   ... More definition.
}
```

Specification 4-4: Revised specification of Figure framework.

The role type set of the Figure class comprises the 8+ role types defined by the Object class, and the 6 role types defined by the Figure class itself. Consequently, the free role type set of the framework is the union of the free role type set of the Object framework and the set of free role types newly introduced by the Figure framework. The specifications show only the addition to the existing inherited role type sets; they do not repeat the full set. This is possible, because once a role type has been made public as part of the class or framework, it cannot be withdrawn (for the sake of substitutability).

4.4.3.2 SimpleFigures extension of Figure framework

Most drawing editors that build on the Figure framework also use a basic set of graphical figures from which they build more complicated application-specific figures. Examples of such basic figure classes are the Polygon, Triangle, Rectangle, and Text classes.

These basic figure classes are captured as the SimpleFigures framework extension of the Figure framework. Figure 4-6 shows parts of its design.

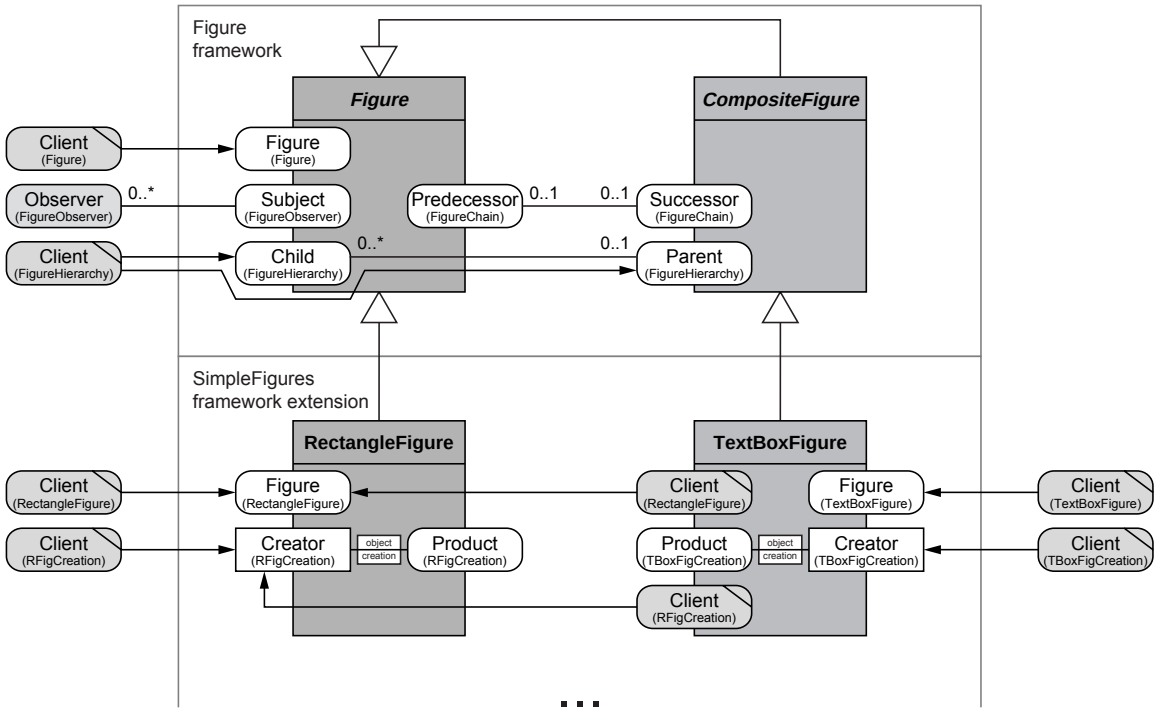

Figure 4-6: Part of the SimpleFigures framework extension.

Specification 4-5 describes the SimpleFigures extension of the Figure framework.

```
// Built-on classes from the Graphics framework.
import Graphics.*;

extension SimpleFigures extends Figure {
  rolemodel RectangleFigure { ... }
  rolemodel RectangleFigureCreation { ... }
  class Rectangle extends Figure { ... }

  rolemodel TextFigure { ... }
  rolemodel TextFigureCreation { ... }
  class Text extends Figure { ... }

  rolemodel TextBoxFigure { ... }
  rolemodel TextBoxFigureCreation { ... }
  class TextBoxFigure extends CompositeFigure { ... }

  ... More basic figures and their models.
```

```
   freeroletypes {
      RectangleFigure.Client;
      RectangleFigureCreation.Client;
      TextFigure.Client;
      TextFigureCreation.Client;
      TextBoxFigure.Client;
      TextBoxFigureCreation.Client;
      ... More free role types.
   }

   builtonclasses {
      Graphics.Polygon;
      Graphics.Font;
      Graphics.Image;
   }
}
```

Specification 4-5: Specification of SimpleFigures framework extension.

The definition of the SimpleFigures framework extension does not provide an extension-point class set, because it should not be extended. New classes might be added to the framework extension itself, however.

As with the Figure framework, the handling of existing and new role types is strictly additive. All role types of Figure and CompositeFigure are inherited, and all free role types from Figure framework are visible on the SimpleFigures extension level.

4.5 Framework layering

In any non-trivial object-oriented system, frameworks build on each other. Free role types, built-on classes, and framework extensions provide the primitives to do so. This subsection examines how these concepts are used to layer frameworks in application systems.

4.5.1 Layers and tiers

Layering in object-oriented systems refers to organizing how classes and class models relate to each other. This is to be distinguished from how (runtime) components as aggregates of objects relate to each other. Runtime components are organized in tiers (and sub-tiers). Layering and tiering are inter-dependent, but not equivalent issues. As used here, layers structure models and their implementation, and tiers structure runtime components.

Reconsider the drawing editor example. The KidsEditor application builds both upon the Editor framework and the Figure framework. It might be put into a KidsEditor application layer. The Editor and Figure framework in turn might be put into a distinct DrawingEditor framework layer. The DrawingEditor framework layer builds upon the Common framework layer that provides the Object and Graphics frameworks. This is a third layer.

In object-oriented systems, this kind of layering is seldom strict. More frequently, frameworks extend or use frameworks from layers below their immediate lower layer. There are no strict rules of how to define layers and what to put into them. However, they should match the application domain and type of system under construction. Domain-specific software architectures are now an active research area. Bäumer et al. present an example of a layering structure for the object-oriented design of interactive software systems [Bäu98, BGK+97].

Tiers, in contrast to layers, refer to how runtime responsibilities are distributed among components. Typical examples are systems based on three tiers: applications, business services, and databases. Each tier consists of one or more components. They are independent of any framework structure. However, frameworks may be used to implement them. An implementation view of these components reveals a layered framework structure, with common frameworks at the bottom, and component specific frameworks at the top. Tiering is typically strict, as components are not allowed to circumvent components of the immediate lower layer.

For our discussion of frameworks, only layering is of interest. Frameworks are not tiered. However, the concept of tier and layer are frequently confused, which is why this subsection explains the understanding of both concepts as used in the following discussion.

4.5.2 Traditional layer coupling

Layers are traditionally coupled using use-relationships and inheritance. Coupling using inheritance can be subdivided into coupling using concept specialization and coupling using callback interfaces. In all cases, the higher layer defines which particular type of coupling it uses. However, a particular coupling becomes only possible if the lower layer allows for it.

In the first type of coupling, control flow is from the higher to the lower layer.

- *Coupling using use-relationships.* A class in a higher layer makes direct use of a class in a lower layer. There need not be any specific relationship between the higher-layer class and the lower-layer class, except that the former needs some of the services of the latter.

In the second and third type of coupling, control flow is from the lower to the higher layer.

- *Coupling using concept specialization.* A class in a higher layer may inherit from a class in the lower layer. In this case, the higher-layer class is a specialization of the concept represented by the lower-layer class. Instances of it may be used wherever instances of the lower-layer class are used.

  If an object is an instance of the higher-layer class, any invocation on an instance of the presumed lower-layer class transfers the control flow from the lower layer to the higher layer. This goes unnoticed from the use-client of the lower-layer class.

- *Coupling using callback interfaces.* A framework in some layer may define a callback interface using an interface or an abstract class. Higher layers implement this interface. The framework explicitly provides it as a hook for extension. It delegates well-defined pieces of work to objects behind that interface.

  A higher layer parameterizes the lower layer with an object of a class that implements the callback interface. While the lower-layer client of the callback interface makes no assumptions about the object behind that interface, it is typically aware of that it is transferring control to a higher layer.

The invocation of an operation that transfers control to a higher layer is frequently called an upcall.

The last two means are similar in that they make use of inheritance and late binding. However, pragmatically they are different. In the first case, inheritance is used to introduce a new concept that extends a lower-layer concept. The higher-layer class inherits the overall set of responsibilities associated with the lower-layer class.

In the second case, the callback interface serves as a hook by which a framework delegates one well-defined piece of work to an unknown client. The implementation of the callback interface does not represent a new concept; it serves simply as a communications hook.

4.5.3 Role–model–based layer coupling

The coupling types just discussed work for class models in general and for frameworks in particular. Using the concepts of this dissertation, however, they can be defined more succinctly. In particular, callback interfaces are better described as callback role types. The use of role models brings the same twist on client responsibilities to framework layering that it has brought to frameworks.

Using role modeling terminology, frameworks are layered using the following coupling mechanisms:

- *Coupling using free role types.* A layer may build on a lower layer via use-relationships as defined by a free role model of a lower-layer framework. A higher-layer class picks up a free role type from a free role model defined by the lower-layer framework.

  This coupling is a special case of the use-relationship-based coupling between clients and a framework, as discussed in Section 4.3 on framework use. A layer must explicitly specify which of its free role types may act as free role types of the layer and therefore which role models may bridge between layers.

- *Coupling using extension.* A layer may build on a lower layer by inheriting from an extension-point class of the lower-layer framework. This coupling mechanism is identical to the coupling using concept specialization mechanism, except that we are now using a more precise terminology (extension-point class).

  A *layer extension-point class* is an extension-point class of a framework that can be inherited from across layer boundaries. A layer must explicitly declare its layer extension-point classes. Also, the layer may only do this for frameworks that are defined by it. It may not redefine frameworks from lower layers.

- *Coupling using callback role types.* A layer may build on a lower layer by assigning callback role types of the lower layer to one of its classes. This coupling mechanism is a specialization of the general coupling using free role types mechanism.

  A *callback role type* is a free role type of a framework that has a non-empty set of operations (see Subsection 4.3.3). It may be picked up by higher-layer classes. Callback role types are the role modeling equivalent of callback interfaces as defined in the previous subsection on traditional coupling mechanisms.

The distinction between classes and role types brings out the different pragmatics of the layer coupling mechanisms.

In an instance of coupling using extension, a higher layer introduces a class model as an extension of a lower-layer framework. This may involve several new classes, each of which may be an extension class of the lower-layer framework. At runtime, instances of these classes are used where framework instances are expected. Often, in particular in case of covariant redefinition, they are used in concert and appear as a team.

In an instance of coupling using free role types, a higher-layer class makes use of lower-layer classes based on traditional use-relationships. The higher-layer class picks up the free role type. Its instances may then collaborate with framework objects based on the free role model the free role type is defined by. The free role model acts as a bridge between layers. It defines precisely what higher-layer clients have to do to make use of lower-layer frameworks. The discussion of black-box framework use applies to this type of layer coupling.

In an instance of coupling using callback role types, a higher-layer class picks up a callback role type from a lower-layer framework. The discussion of coupling using free role types applies, with the addition that the higher-layer use-client class is aware that control may be transferred to its instances at runtime.

This role modeling view on layer coupling provides all the benefits that free role models and extension-point classes have brought to framework design and use.

4.5.4 KidsEditor framework layering (example)

This subsection uses the KidsEditor application system as an example to discuss framework layering.

The KidsEditor system consists of several class models, some of which are frameworks, some of which are domain and application-specific framework extensions. The following discussion distinguishes three main categories, which it organizes into three different layers.

- *Common framework layer*. This layer comprises the Object, Graphics, and Serialization frameworks (among others). Graphics and Serialization are extensions of the Object framework. The Object framework defines the fundamental Object and Class classes, the Graphics framework defines the Graphics and related classes as discussed earlier, and the Serialization framework provides classes for making objects persistent.
- *DrawingEditor framework layer.* This layer comprises the Editor and Figure frameworks, and the SimpleFigures framework extension. Editor and Figure extend the Object framework, SimpleFigures extends the Figure framework. The Editor framework provides the main application classes and their functionality. Figure and SimpleFigures are discussed above.
- *KidsEditor application layer*. This layer comprises the KidsEditor and KidsFigures framework extensions. KidsEditor extends Editor, and KidsFigures extends Figure. Both are application-specific framework extensions that are not frameworks themselves.

Taken together, the two base layers, Object and DrawingEditor, form what is frequently called an application framework. An application framework may be viewed as a composite framework, that is an aggregate of further smaller frameworks.

Figure 4-7 visually depicts the layering structure of the KidsEditor application system. The type of arrows between the class models indicates the type of coupling between the layers being bridged. A white arrowhead indicates a framework extension, and a slim black arrowhead indicates a use-relationship based on free role types.

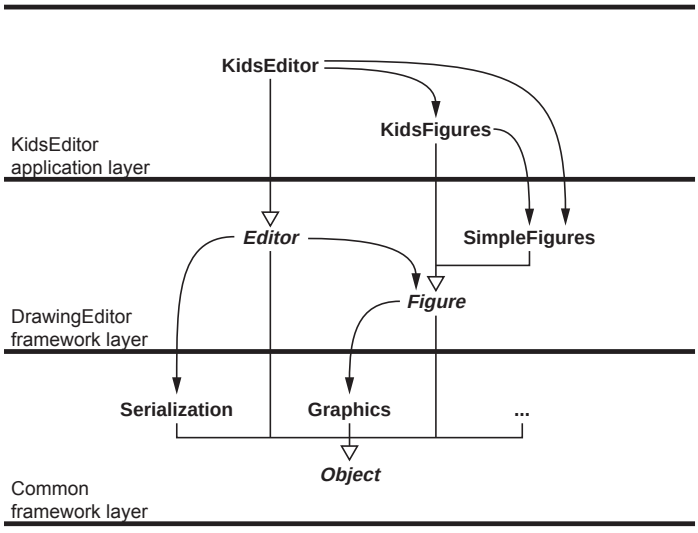

Figure 4-7: Layering of KidsEditor application system.

The layered structure of the application system can also be described textually. Specifications 4-6 to 4-8 show how this looks like, taking into account the specific layer coupling mechanisms. The shortcut

Object.freeroletypes indicates that all free role types from the Object framework are taken and provided as the free role types of the Common framework layer.

```
layer Common {
  framework Object;
  framework Serialization;
  framework Graphics;

  freeroletypes {
     Object.freeroletypes;
     Serialization.freeroletypes;
     Graphics.freeroletypes;
  }

  extensionpoints {
     Object.extensionpoints;
     Serialization.extensionpoints;
     Graphics.extensionpoints;
  }
}
```

Specification 4-6: Common layer.

```
layer DrawingEditor {
  framework Editor;
  framework Figure;
  extension SimpleFigures;

  freeroletypes {
     Editor.freeroletypes;
     Figure.freeroletypes;
     SimpleFigures.freeroletypes;
  }

  extensionpoints {
     Editor.extensionpoints;
     Figure.extensionpoints;
  }
}
```

Specification 4-7: DrawingEditor layer.

```
layer KidsEditor {
  extension KidsEditor;
  extension KidsFigures;

  // No free role types.
  freeroletypes {}

  // No extension of layer.
  extensionpoints {}
}

application KidsEditor {
  // Set of included frameworks is the transitive closure
  // of all class models reached from the root set given here.
  extension KidsEditor;
  extension KidsFigures;
  extension SimpleFigures;
}
```

Specification 4-8: KidsEditor layer and application.

Frequently, all free role types of a framework become free role types of the layer the framework is defined in. Sometimes, free role types are restricted to be visible only within a given layer. This may be used to confine framework extension to one specific layer.

4.6 Framework documentation

This section analyses the consequences of a role modeling approach to framework design on framework documentation. It defines a template for framework documentation that helps to make documentation more precise and more helpful in the face of complex object-oriented frameworks.

4.6.1 What and when to document

Any documentation of a framework serves at least one of several purposes. Two primary purposes of documentation are the explanation of how to use a framework (external client view) and the explanation of how a framework works to change and evolve it (internal view).

Each instance of a documentation type takes on its particular form, depending on where the complexity of using or understanding a framework lies. This form depends on the technique being chosen to illuminate and explain a particularly complex aspect of a framework. For example, formal specifications can be chosen if expected behavior needs to be defined in detail, design patterns can be used if the design rationale of a framework needs to be communicated, or cookbooks can be used if recipe-like learning by example is considered most helpful.

A technique should be chosen judiciously. For example, if the design is simple, a simple approach may suffice. Or, if source code and a mature browsing environment are available, less needs to be documented. Role modeling as a documentation technique should be chosen if the complexity of classes, object collaborations, and requirements put upon use-clients is non-trivial.

Writing good documentation is hard and frequently tiresome. In practice, therefore, developers or vendors try to make a documentation serve many different purposes at once, to reduce their work. Also, using a framework and understanding its inner workings are issues that depend on each other. Therefore, a framework's documentation is often a hybrid, serving different purposes.

The most common type of documentation is the *reference* or *API documentation.* It is also the most basic one. A reference documentation lists the framework classes, possibly with some explanation, and then describes each operation of a class in detail. This minimal type of documentation serves to explain the contents of some library or package, without much consideration for the design or ways of using it.

Frequently, reference documentation is generated from source code with the support of dedicated tools like Javadoc. Examples are the JDK 1.02 and 1.1 documentation, the BeOS reference documentation, and the Unix man pages [Sun96a, Sun96b, Be97].

Of more interest is the *design documentation* of a framework, which is about how a framework works (internal view). It describes its full design, including all relevant details. The documentation includes the classes, their collaborations, dependencies, and constraints on how to use them. Design documentation describes the inner workings of a framework and can therefore be used for many different purposes. Its primary purpose, however, is to help developers understand the framework to change and evolve it.

Finally, documentation is needed that describes how a framework is to be used (external client view). Such usage documentation needs to cover both ways of using a framework (use-clients and extension clients). Again, documenting a framework means choosing techniques to describe particular framework aspects. Therefore, not much can be said about usage documentation in general.

However, role modeling can be used to address several issues that arise in the documentation of a framework.

4.6.2 How role modeling can help

Role modeling can help to make the three types of documentation just discussed more effective.

- *Reference/API documentation.* Reference documentation consists of class descriptions. If the class interfaces are complex and non-trivial, they can be split up into role types. Smalltalk, for example, provides a related feature in its system browsers: methods can be grouped into method categories. Such method categories reduce the complexity of class interfaces without much additional effort.
- *Design documentation.* The design documentation of a framework can be based on role models to describe the internal and external object collaborations. Next to the discussion of the role models, such a design discussion needs to define the classes as compositions of role types. These two dimensions, classes and role models, depend on each other.
- *Usage documentation.* The usage documentation of a framework can use role models to describe the client interaction as a set of free role models. Also, it can use the concept of extension-point classes to determine how to extend a framework. A framework's reference documentation can be viewed as part of such usage documentation.

Wherever a role model is used, it might be identified as an instance of a particular design pattern. Identifying design patterns helps to speed up understanding the role model and the described design aspect.

Also, role modeling can be applied selectively as a technique to focus on specific aspects of a framework, without having to use all of the concepts at once.

Therefore, role modeling is an evolutionary addition to current documentation techniques. It does not invalidate them, but rather adds to them to help make documentation more precise where necessary.

4.6.3 A simple design documentation template

This subsection defines a simple template for documenting the design of frameworks using role modeling. The template is used in the case study Chapters 6 to 8. It is not meant to be complete; rather, it provides the most important parts only.

The description of a framework is broken up into the following pieces, presented in that order:

1. *Framework overview.* This section gives an overview of the whole framework and describes its purpose. It lists its key classes, responsibilities, and collaborations. It lists frameworks built upon.
2. *Class model.* This section walks through the list of framework classes, explains their purposes, responsibilities, and collaborations, as well as the inheritance structure.
3. *Free role models.* This section describes the free role models of the framework. It describes in detail the requirements put upon use-clients that want to make use of the framework.
4. *Internal role models.* This section lists the framework-internal role models that structure the operations of the framework and help it provide its primary services.
5. *Built-on classes.* This section describes how framework classes build upon other classes. It lists the built-on classes and the role models through which framework classes connect to the built-on classes.

Sometimes, the free role model and the internal role model section are merged into one. To better illustrate how the framework is extended, the documentation may be accompanied by an example framework extension.

6. *Example extension.* This section describes an example extension that illustrates how to extend the framework using inheritance. It may be used as a recipe for applying the framework in further domains.

The example extension is described using the same template as this very template, so it comprises an overview, class model, role models, and built-on classes.

The case study chapters present several examples of frameworks documented using this template.

4.7 Summary

This chapter has applied the role modeling concepts from Chapter 3 to the design of object-oriented frameworks. It has covered framework use through use-clients, framework extension through inheritance, framework layering for application design, and framework documentation based on role modeling.

The next chapter focuses on how to describe and implement role–model–based designs using industry standards like UML, Java, and Smalltalk. After this, the case study chapters and their evaluation follow.

5

Extension of Industry Standards

Role modeling for framework design is a new technique, for which no standard exists. Current industry design notations and programming languages provide no direct support. While standard design notations and programming languages provide classes and objects, they do not support role types, role constraints, and role models. However, standards are important, because they provide developers with a shared vocabulary and tool support. This chapter discusses how to extend industry-standard design notations and programming languages with role modeling concepts. Extensions are provided for UML, Java, C++, and Smalltalk.

5.1 Chapter overview and motivation

Role modeling for framework design introduces new concepts and extends established ones. Effectively, it forms a new design method. However, many of the concepts cannot be expressed directly using industry-standard design notations and programming languages.

Some of the new or revised concepts map well on concepts of current design notations and programming language standards. For example, the concept of class maps well on the UML concepts of interface or class. Other concepts, for example, the role model concept, are more difficult to map, because no equivalent concept exists.

Role modeling for framework design is only a part of what could be a full-blown design method. It ignores many issues that are needed for an industrial-strength design method. One possible conclusion might therefore be to develop a new full-blown design notation and programming language that directly supports role modeling.

This approach has the following advantages:

- Developers can directly express framework designs. No mental gap between the concepts behind a design and its expression using a specific notation must be maintained.
- Given adequate tools, framework design and implementation can be checked for conformance based on the new concepts. Such checks may significantly reduce the error rate.

However, this approach also has the following disadvantages:

- Role modeling for framework design is an evolutionary addition to existing approaches. It does not try to replace existing standards and should therefore extend rather than replace them.
- The development of a full-blown industrial-strength modeling approach requires a substantial amount of work. A new approach would distinguish itself only through its new framework design concepts.
- Developers would have to learn yet another new method. This increases the time until they productively join a project and makes it more difficult to find people that are willing to maintain a system.
- No tool support or no tool support comparable to the support for mainstream notations is available.

For practical purposes, it is preferable to extend existing industry standards with the new role modeling concepts rather than to introduce yet another approach that is incompatible with the existing ones. An extended industry standard provides the new concepts and still lets developers get to work quickly. This approach faces less developer resistance, with tool support being available right from the beginning.

A dedicated design notation and programming language for role modeling can be introduced later, should role modeling for framework design reach widespread acceptance. In fact, role programming techniques like [Van97, VN96] and general purpose programming language extensions like aspect-oriented programming [KLM+97] already point into that direction.

This chapter discusses how to extend industrial-strength design notations and programming languages with the new role modeling concepts. First, it discusses requirements and properties common to all these extensions. Then, the chapter introduces an extension of one design notation, UML. Finally, it provides extensions of the programming languages Java, C++, and Smalltalk.

5.2 Common properties

This section discusses how industry standards can be extended with role modeling concepts, and how mappings between standards ease the introduction of a new extension. The section also lists properties that should be common to all role modeling extensions of existing industry standards. It reviews the concepts that must be provided by each such extension and introduces a simplified version of the Figure framework example.

5.2.1 Extending an industry standard

A successful extension of an industry standard is only possible if the extension blends in well with the existing concepts and their idiomatic use. Otherwise, the extension would feel alien to the developers

and be rejected. Fortunately, role modeling for framework design was intended to be an extension of known concepts right from the beginning.

As a consequence, the industry standard will take on the dominant role, and the role modeling concepts that extend it must come natural to users of the standard. They must be expressed in terms of the standard's concepts or extension mechanisms. Chapters 3 and 4 define and discuss the role modeling concepts in standard-neutral terms to ease the extension of different standards and put them onto a common basis.

Figure 5-1 illustrates how the role modeling concepts extend the four standards discussed in this chapter (UML, Java, Smalltalk, and C++).

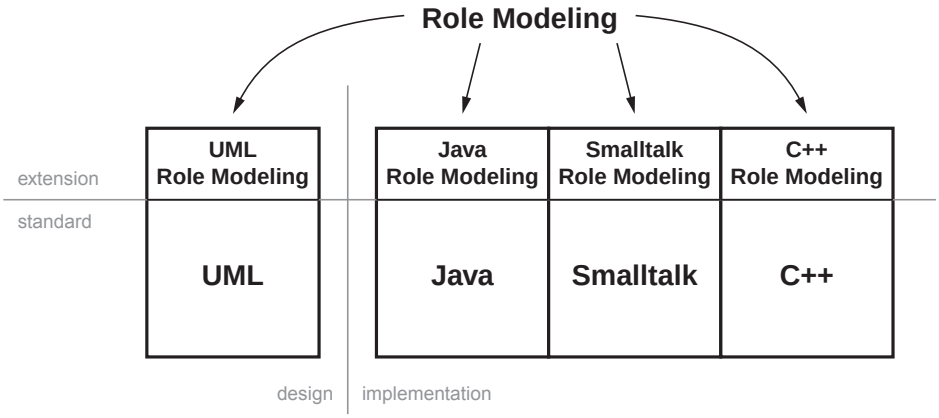

Figure 5-1: Extension of industry standards with role modeling concepts.

Figure 5.1 illustrates two types of relationships: extensions and mappings.

- Extensions are role modeling extensions of the respective industry standard. For example, UML Role Modeling is an extension of UML within the confines of UML. An extension is the definition of role modeling concepts in terms of the existing standard's concepts and extension mechanisms.
- Mappings are mappings between the general role modeling concepts and a role modeling extension of a specific industry standard. A mapping is a function that explains how a role modeling concept is represented in a specific standard.

Being explicitly aware of the mappings is important, because it helps introduce new industry standard extensions quickly and without much overhead. For example, if a new programming language is defined, and UML tool support for that language exists, the extension of the new programming language with role modeling concepts can be described as the composition of three functions $M_{RM/UML}$, $E_{UML/RM}$, and $M_{UML/PL}$:

- $M_{RM/UML}$ is the mapping from the general role modeling concepts to the UML Role Modeling concepts.
- $E_{UML/RM}$ is the extension of the UML concepts with the UML Role Modeling concepts.
- $M_{UML/PL}$ is the mapping between UML and the new programming language.

A more detailed and direct extension of the programming language can be introduced later.

The mappings are typically one-way mappings. Depending on the target of the mapping, it cannot be guaranteed that an inverse mapping exists. In any such mapping, information from the domain is lost. This is one reason, why it is better to provide direct extensions of the programming languages rather than using a mapping detour through UML as just illustrated.

The other reason is that it is more convenient to use a direct extension rather than carrying out the composition of three mappings. Still, if a new programming language is introduced, for which no direct extension exists yet, developers can use this mechanism.

5.2.2 General requirements

The extension of an industry standard with role modeling concepts must not only be as precise as possible, but it must also match the way developers work with the extended standard and the base standard.

Ideally, the following requirements are fulfilled when defining an extension:

- *Robustness.* A design or program based on the extended standard must be robust with respect to interpretation and handling by developers who do not understand the extension. Developers must still be able to work on the design.
- *Inconsistency.* Also, a design that is inconsistent from the extension's point of view must not block the system in any respect (for example, browsing, type checking, or code generation). As a corollary to the robustness requirement, developers must still be allowed to work productively with an inconsistent design.
- *Incompleteness.* The design must allow evolutionary adding of elements based on the extended standard rather than requiring that the design be fully compliant with the extension right from the start. Thus, the extension must allow for its partial application.
- *Tool support.* Non-trivial designs require tool support, such as Rational Rose, Paradigm Plus, Visio, etc. The extended standard must take into account how users handle designs visually. (This is one reason why, for example, the UML notation guide describes a concrete visual syntax rather than an abstract syntax only.)
- *Bridge between design and implementation.* An extended standard should not complicate bridging between design and implementation. A design usually keeps evolving close to its implementation. Developers avoid making the gap between design and implementation too large to reduce intellectual mismatch.

Providing robustness, dealing with inconsistency, and dealing with incompleteness ensures a low entry hurdle to using role modeling. Because designs may be incomplete, the initial investment may scale from zero to the full application of the method. Because a design may be inconsistent from a role modeling perspective, the whole system (tool + design) stays operational at any point in time.

Providing robustness is the hardest requirement. It prevents that the mapping introduces additional complexity and uncommon semantics into a design that is otherwise easily interpretable using general knowledge of the underlying design notation.

5.2.3 Handling role types and role models

UML, Java, C++, and Smalltalk provide support for classes and their packaging, next to a wealth of other concepts. Role modeling for framework design requires support not only for classes, but also for role types, role constraints, role models, class models, and frameworks. None of the industry standards discussed in this chapter provides native support for these concepts.

However, developers always have modeled and implemented class models and frameworks. They typically use the standard's packaging mechanism to define what goes into a class model or framework, and they use the naming mechanism and conventions of the standard to separate different mod-

els. We can therefore use the packaging mechanism to define role models, class models, and frameworks.

Each section of this chapter on extending a specific standard shows how the role modeling concepts of class, role model, class model, and framework are represented using native concepts. The representation is typically based on the standard's concept of type, class, and packaging.

The primary challenge is to adequately extend a standard with the role type, role constraint, and role model concept, and to define how they relate to classes and class models. For each standard, this is done differently. Role constraints do not require much discussion, but role types do, due to the many different ways of using them.

Let us assume that a standard provides a simple means to represent a type, for example a UML or Java interface. A role type could then be expressed using an interface. However, different kinds of role types have different pragmatics, and it is questionable whether one concept of interface fits all uses of the role type concept. Moreover, given the high number of role types in a design, the naive approach faces an explosion in the number of interfaces.

For expressing role types, we face the following options:

- Leave role types implicit in a class interface.
- Make role types explicit in a class interface by annotating them (for example, through method categories).
- Make role types explicit as interfaces of their own.

For expressing role models, we face the following options:

- Leave role models implicit by leaving all of their role types implicit.
- Make a role model explicit as part of a class model by packaging role models with their class model.
- Make a role model explicit and reusable for different class models by packaging it on its own.

Design and implementation use these options differently. In design, typically every important design decision should be recorded and expressed. In implementation, only all implementation decisions should be explicit, but there is no need to directly reflect the full design.

The design notation and the programming language sections introduce their respective rules below.

5.2.4 Figure framework (example)

This chapter uses a simplified version of the Figure framework as its example. Figure 5-2 shows its design.

The Figure framework comprises two classes (Figure and CompositeFigure), builds on the Graphics class, and extends the Object framework. The dynamics of the framework are defined by the ObjectProperty, Figure, FigureChain, FigureObserver, FigureHierarchy, and Graphics role models.

The framework is discussed in depth in Chapters 3 and 4. It offers all constellations that we need to discuss.

- ObjectProperty is a reusable role model. All other role models are non-reusable.
- Figure.Figure, FigureObserver.Subject, etc. are regular role types without further qualification.
- FigureObserver.Observer is a free role type that has operations.
- Figure.Client, FigureHierarchy.Client, and Graphics.Client are free no-op role types.

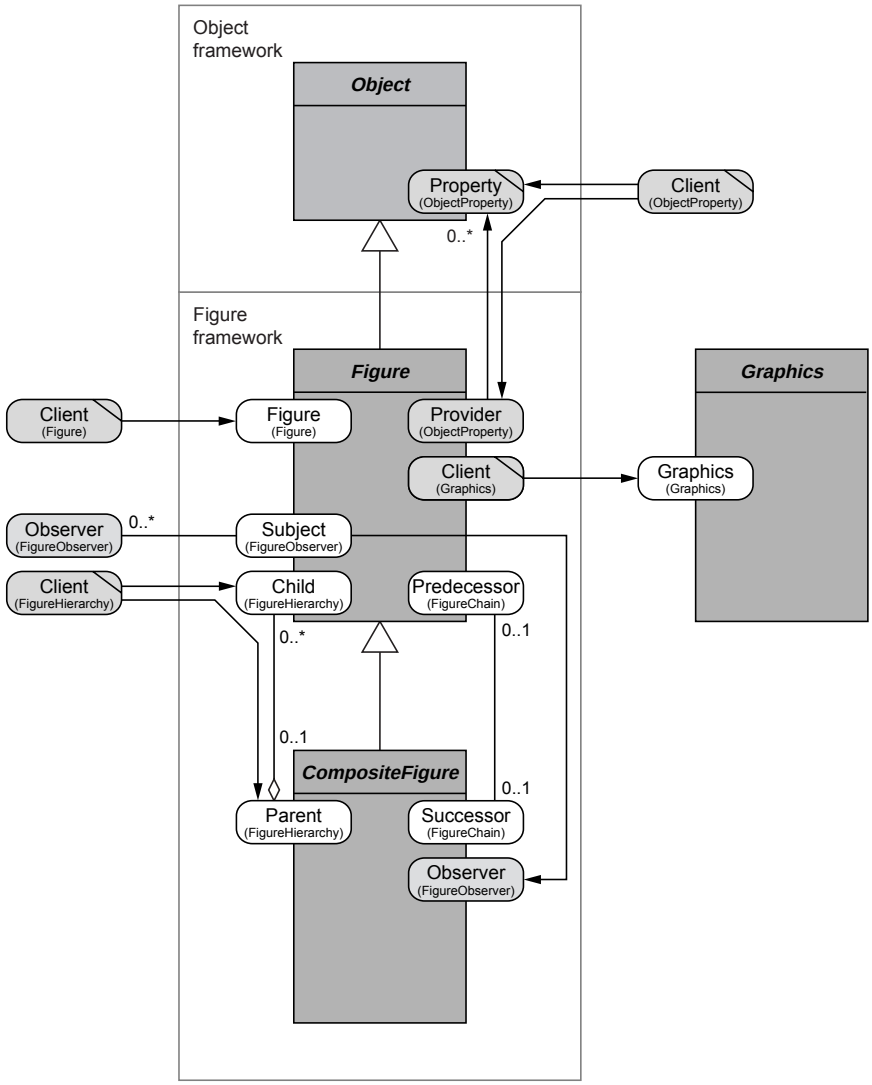

Figure 5-2: The (simplified) Figure framework.

Sections 5.3 and 5.4 show how the framework is expressed in UML and implemented in Java, C++, and Smalltalk.

5.3 Design notations

This section discusses how to extend a design notation. It provides one example: the extension of UML with role modeling concepts for framework design. The discussion omits the trivial parts and focuses on the higher-level concepts of role type, role constraint, class, role model, class model, and framework.

5.3.1 Extending design notations

A design notation must provide means to record and document *all* relevant design decisions. In contrast to programming languages, which usually let developers make only parts of a design explicit in the implementation, should a design notation let developers record all relevant information (whether they do so is another question).

Therefore, the extension of a design notation must cater for all new or revised modeling concepts, including role type, class, role constraint, role model, class model, and framework. Also, it should be possible to express variations of specific design concepts, like free role types and no-operation role types.

This section does not discuss how to express the dynamic behavior of objects acting according to a class type or role type. The definition of such a notation is left to the available type specification mechanisms. UML, for example, offers object and sequence diagrams to illustrate runtime object behavior. Also, more elaborate approaches like Extended-Event-Traces [BHKS97] or others can be used.

The rules for expressing role models, class models, and frameworks on a design level are simple:

- *Class model.* A class model should be represented explicitly. It gets at least one package. For structuring complex models, more packages can be used.
- *Reusable role model.* A reusable role model should be represented explicitly. It should be packaged independently of a specific class model in which it is used, for example, in a package of reusable role models.
- *Non-reusable role model.* A role model that is used once in a class model need not be packaged on its own. It should go into the class model's package. Within that package, it might get its own diagram or not, depending on the significance of the role model.

Role types of a reusable role model should always be made explicit, both in design and implementation.

For non-reusable role models, the situation is more complicated. As discussed, we can qualify role types as being free and/or no-op. These properties lead to the following kinds of role types and their rules for being left implicit or made explicit:

- *Role types.* Regular non-free role types with operations should be made explicit, because they represent an important design aspect that needs to documented explicitly.
- *Free role types (both no-op and with operations).* Free role types should be made explicit, because they may be picked up by more than one client class, and because they represent an important design aspect.
- *Non-free no-op role type.* A non-free no-op role type may or may not be made explicit. It is only used within the context of the current class model; hence the number of uses is fixed.

Thus, role types of a non-reusable role model should be made explicit if they are regular or free role types. If they are non-free no-op role types, a developer may decide to not explicitly represent a role type but to reduce it to an annotation of the class providing the role type.

5.3.2 Extending UML with role modeling

UML offers a rich metamodel for modeling object systems, which makes it easy to extend it with role modeling concepts. The extension of UML with role modeling concepts relies only on three basic UML concepts: Class, Interface, and Stereotype.

- A class maps either on a UML class or a UML interface. Whether a class or an interface is chosen depends on the complexity of implementing the class. A lightweight class is typically represented as a UML class, while a heavyweight class, for which several implementations exist, or which has many role types, is represented as a UML interface.
- A role type maps on a UML interface tagged with the «RoleType» stereotype. Such an interface is called a role-type interface. A free role type is represented as a «FreeRoleType» interface, a no-op role type as a «NoopRoleType» interface, and a free no-op role type as a «FreeNoopRoleType» interface.
- A role constraint between two role types is mapped on a textual annotation of the relationship between the two role types. If no annotation is provided, the default case (role-dontcare) is assumed.
- A role model or a class model maps on a class diagram. There is no need to make a class diagram for every role model, but if a role model is reusable, it should get its own class diagram.
- A framework also maps on a class diagram. In addition, it might be packaged as its own component, drawing visible boundaries to the outside.

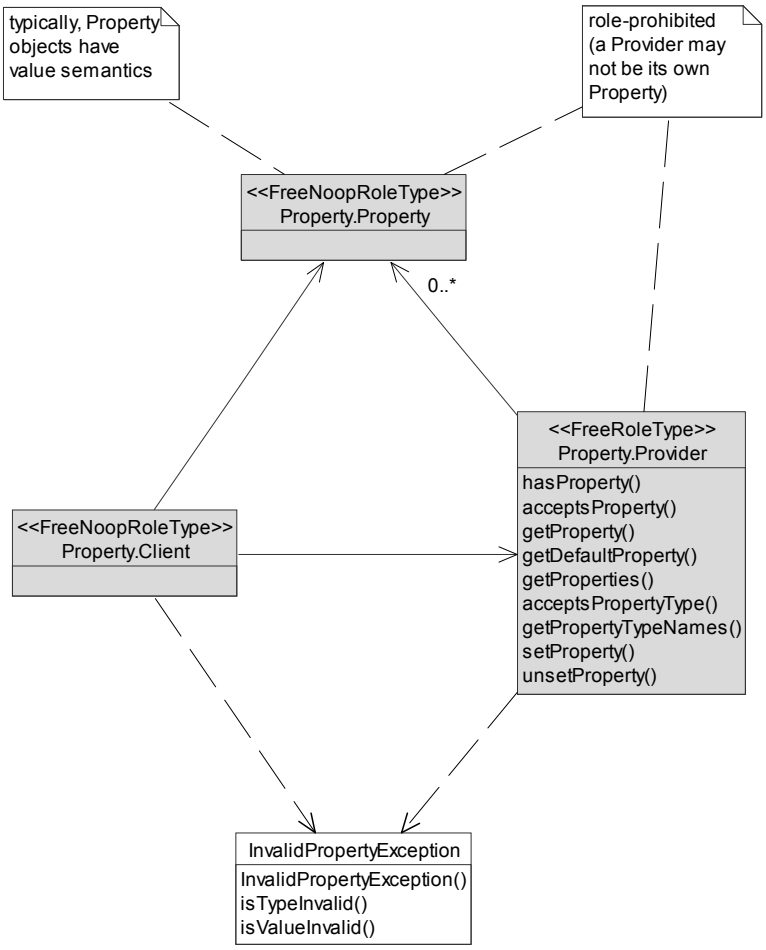

Figure 5-3: The ObjectProperty role model as a UML class diagram.

When working with UML Role Modeling, developers use UML classes and interfaces as they are used to. If needed, they add additional role type and role model information. They tag interfaces and classes as specific kinds of role types, and make the role models explicit by connecting role-type interfaces and classes.

Figure 5-3 shows the ObjectProperty role model using a UML class diagram. It uses the color coding introduced earlier: free role types have a light-gray background.

The ObjectProperty role model is a reusable role model that should be packaged on its own, or, at least, should get its own diagram. Property.Client, Property.Provider, and Property.Property are role-type interfaces. Because the ObjectProperty role model is a reusable role model, all of the role types are free. In addition, the Property.Client and the Property.Property role types are no-op role types.

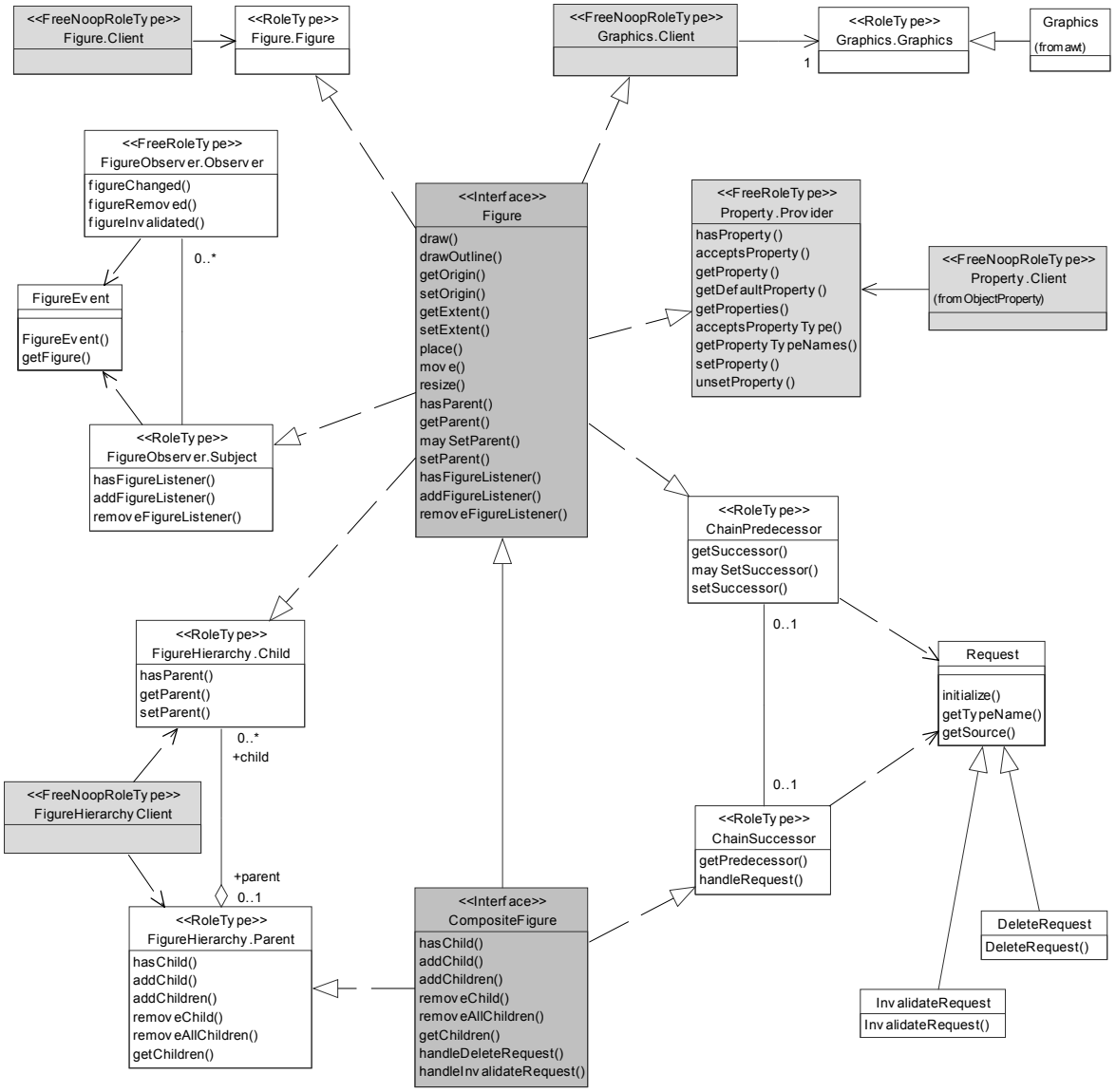

Figure 5-4: The Figure framework as a UML class diagram.

Please also observe the use of the class InvalidPropertyException without any further qualification. This reflects the pragmatic approach of robustly embedding the role modeling concepts into an existing standard. From a UML perspective, the role-type interfaces are just interfaces that relate to other interfaces and classes without further qualification.

Figure 5-4 shows the Figure framework using a UML class diagram. In addition to coloring free role types in light-gray, it colors role modeling classes in dark-gray. Please note that Figure and CompositeFigure are conceptually classes from a role modeling perspective, but are represented as UML interfaces. As explained, this decision is a matter of expected convenience of implementation.

Figure 5-4 shows how different kinds of role types are represented:

- Examples of role-type interfaces are FigureHierarchy.Child and FigureChain.Predecessor.
- Examples of free role-type interfaces are FigureObserver.Observer and ObjectProperty.Provider.
- There are no examples of non-free no-op role type interfaces.
- Examples of free no-op role type interfaces Figure.Client and FigureHierarchy.Client.

Most of the role models bound to this class model are not made explicit. They exist only in this diagram, and are implicitly represented by the naming convention of their role types.

The role modeling classes Figure and CompositeFigure are represented as UML interfaces, while the (simplified) helper class Request is represented as a UML class.

5.3.3 Extension properties

The following discussion shows how the extension properties asked for in Section 5.2 are realized by the UML role modeling extension. It thereby describes the rationale behind the extension.

- *Robustness.* The interpretation of a UML Role Modeling class diagram can be done by any developer who knows interfaces and classes, and who is told that a roletype interface represents a type that is part of the class type it is extended by. This is the minimal effort required.
- *Inconsistency.* Standard UML allows classes and interfaces to directly refer to each other. Thus, role-type interfaces may refer to other UML classes and interfaces, without an intermediate role-type interface. From a role modeling perspective, such direct reference indicates that the design may not yet be done and that further elaboration will provide the missing role types.
- *Incompleteness.* Designs may be inconsistent, and therefore incomplete.
- *Tool support.* The mapping uses only basic UML concepts. These concepts are the first to be provided by a UML based modeling tool, so that tool support for UML Role Modeling is readily available, even if the tool supports only a subset of standard UML. If code generation needs to be supported in more detail, additional stereotyping of interfaces can provide missing information and control back-end code generation.
- *Bridge to implementation.* The mapping supports modeling close to the implementation level. In an implementation, a UML interface typically maps onto the interface concept of a programming language, while a UML class directly maps on an implementation class. Mapping a role modeling class on a UML interface or class lets developers make this distinction on the design level and supports code generation.

The definition of the mapping uses the most common abstractions only, because developers best know them, and because the UML definition still evolves, as it is imprecise and unclear in many instances [BHH+97].

5.4 Programming languages

Every framework is eventually implemented using a programming language, independently of whether it was designed using role modeling or not. Typically, programming languages provide only meager support for representing design-level concepts that go beyond individual classes. This section analyses which of the role modeling concepts to map to a programming language and how to do so. The section covers Java, C++, and Smalltalk.

5.4.1 Extending programming languages

Programming languages provide mechanisms for source code definition and structuring. Designers of programming languages typically do not include higher-level design concepts like role model, class model, or framework in the language definition. These concepts are left to the design level of software system development, as opposed to the implementation level, the support of which is the primary concern of programming language designers. All programming languages covered in this section exhibit this property.

The distinction between design and implementation relaxes the situation. While a design must capture all relevant design details, an implementation must only provide a proper implementation of the design, but not the design itself. Hence, there is no need to extend a programming language with all role modeling concepts. Rather, it is sufficient to provide those concepts that let us implement a design in such a way that we find our way back to the design documentation and do not have to bridge too far a mental gap when doing so.

For identifying role models, class models, and frameworks, the packaging mechanism of a given programming language can be used. Implementation-level packaging follows the same rules as design-level packaging:

- A class model gets its own package (at least one).
- A reusable role model gets its own package.
- A non-reusable role model becomes part of the package of a class model.

Implementation-level role types can be expressed using interfaces or equivalent concepts. All role types of reusable role models should be explicit. For role types from non-reusable role models, the following rules apply:

- *Role type.* A regular role type need not be made explicit as an interface. It can be merged with the class interface. It is helpful, though, to annotate the operations of a class interface with the role type names.
- *Free role type.* A free role type with operations must be represented explicitly, so that clients can pick up the role type by inheriting from it or copying it.
- *No-op role type.* A no-op role type that is not free need not be made explicit. However, it is helpful to annotate a class as providing a particular no-op role type.
- *Free no-op role type.* A free no-op role type may or may not be made explicit. Because the role type is free, it is important to see it explicitly. Because it is no-op, simply tagging a client class might be sufficient.

A class should list all of its role types, including no-op role types, and it should annotate an operation with the role type name from which it is derived. If a role type is explicit, this happens automatically. If a role type is not made explicit, developers need to do this by hand.

Role constraints are not made explicit. This feature of the role modeling technique is most useful on the design level to communicate the idea behind a role model. It is left to the implementation of a class to ensure that a role constraint is maintained. The role types or class interfaces involved may document the constraint, though.

To support the annotation of class interface with role type and role model information, specialized documentation tools can be used. For example, Java's javadoc and its variants in other programming languages provide lightweight annotation features that support documentation, code generation, and round-trip engineering between design and implementation.

5.4.2 Problems of programming language extension

The pragmatic extension of programming languages described above is too weak for properly checking the conformance of a design with its implementation. Such conformance checking is desirable, because it lets us catch implementation errors as early as at compile-time. However, conformance checking is typically done only for very specialized areas of software system design. The critique therefore applies to the whole of object-oriented modeling.

More immediate problems occur due to the composition of role types to form a class type. Most programming languages do not support type specification and composition. Eiffel is an exception of a close-to-mainstream programming language that provides several useful features of this kind, in particular a lightweight type specification mechanism (design by contract [Mey91, Mey92b]), and a renaming feature [Mey92].

The following problems arise from the need to compose role types:

- *Name clashes.* Different role types might define two identical operations. However, their semantics is different, and they are used for different purposes. The operations must be distinguished, and the naming conflict must be resolved.
- *Redundant operations.* Different role types might define operations that have different signatures, but the same implementation. In the Figure framework example, operations getParent and getSuccessor always return the same object reference, making one implementation redundant.
- *Mapping of abstract state on implementation state.* Each role type comes with its own state space definition. When composing role types, the state spaces of the role types must be mapped on the single state space of the class type, for which a proper implementation state space must be defined.

The first problem, name clashes, is a common problem that is independent of the approach presented here. Some programming languages offer explicit renaming features. If no such feature is available, the names must be changed to avoid the conflict. Usually, the operation names are pre- or postfixed to indicate their origin.

The second problem, redundant operations, is not a real problem, but rather an issue of style. Should a redundant operation be removed? Names are important, even if two operations do the same. Each client expects operation names that are best suited for the task at hand. Therefore, redundant operations should not be removed. Rather, they should be named well and implemented using common primitive operations.

The last problem, the definition of the implementation state space of a class, results from the larger problem of type composition. The programming languages considered here describe the semantics of some operation using prose. The only aspects described formally are the parameters and return values of the operations of a role type. They define the abstract state space that clients see of an object playing a role defined by the role type.

A developer of a class must define how the abstract state spaces of the role types are composed to form the abstract state space of the class. For the abstract state space of a class, the developer must

then find a suitable and efficient implementation. In practice, the intermediate step of defining the abstract state space of a class is not explicit, and a class is implemented by deriving its implementation state space directly from the role types.

In the end, this is not a problem that can be treated on a general level, and it is up to every developer to find a suitable implementation state that lets him or her efficiently implement the different role types. Frequently, a class internal set of primitive operations is used by the role type implementations to manipulate the common implementation state. Also, the implementations of role type operations call each other to communicate a state change. Implementing this is an issue of proper programming practice, and not considered further here.

5.4.3 Extending Java with role modeling

Java provides interfaces, classes, and packages as concepts that can be used to implement role–model–based designs. Java interfaces and classes can be used to represent role types and classes, and Java packages can be used to provide a namespace for role models, class models, and frameworks.

A role type is represented as a Java interface. A class is represented as a Java interface or class, depending on its expected implementation complexity. Reusable role models and class models are packaged as individual units.

Specification 5-1 shows the source code of the Java Figure interface.

```
package org.riehle.diss.Figure;

import java.util.Enumeration;
import java.awt.Graphics;
import java.awt.Point;
import org.riehle.diss.ObjectProperty.*;

/**
A Figure object is a graphical figure object that
can be used in drawing editors. It provides the
following role types:

- Figure.Figure
- FigureHierarchy.Child
- FigureObserver.Subject
- FigureChain.Predecessor
- Graphics.Client
- ObjectProperty.Provider
**/

public interface Figure extends PropertyProvider {
  /**
  @roletype Figure.Figure
  **/
  public void draw(Graphics gc);
  public void drawOutline(Graphics gc);
  public Point getOrigin();
  public void setOrigin(Point origin);
  public Point getExtent();
  public void setExtent(Point extent);
  public void place(Point location);
  public void move(int dx, int dy);
  public void resize(int handle, int dx, int dy);

  /**
  @roletype FigureHierarchy.Child
  **/
  public boolean hasParent();
  public CompositeFigure getParent();
  public boolean maySetParent();
  public void setParent(CompositeFigure parent);
```

```
  /**
  @roletype FigureObserver.Subject
  **/
  public boolean hasFigureListener(FigureListener listener);
  public void addFigureListener(FigureListener listener);
  public void removeFigureListener(FigureListener listener);

  /**
  @roletype FigureChain.Predecessor
  **/
  public CompositeFigure getSuccessor();
  public boolean maySetSuccessor();
  public void setSuccessor(CompositeFigure successor);

  /**
  @roletype Graphics.Client
  @properties free noop
  **/
}
```

Specification 5-1: Definition of the Java Figure interface to represent the Figure class.

A role type that is to be explicit is represented as a Java interface. The use of abstract classes as a representation mechanism of a free role type does not make sense, because Java classes are restricted to single inheritance. The interface should be named after the role type.

Using customary Java naming, Specification 5-2 shows the free role type FigureObserver.Observer.

```
package org.riehle.diss.Figure;

import java.util.EventListener;

/**
This interface represents the FigureObserver.Observer role type.

A FigureListener is an object that is notified by a Figure object
about state changes of that Figure object.

@roletype FigureObserver.Observer
@properties free
**/

public interface FigureListener extends EventListener {
  public void figureChanged(FigureEvent e);
  public void figureRemoved(FigureEvent e);
  public void figureInvalidated(FigureEvent e);
}
```

Specification 5-2: Definition of the Java FigureListener interface to represent the FigureObserver.Observer role type.

A free no-op role type of a class model may be represented as an empty Java interface (or not at all). For example, the Figure class provides the Graphics.Client role type. This is a free no-op role type of the Graphics framework. The provision of the role type can be annotated in the class interface as illustrated in Specification 5-3:

```
public interface Figure extends PropertyProvider {
  ...

  /**
  @roletype Graphics.Client
  @properties free noop
  **/
}
```

Specification 5-3: Role type Graphics.Client in the definition of the Java Figure interface.

Examples of where it makes sense to explicitly represent a free role type without operations are client role types for object creation as presented in Specification 5-4. The comments in the client interface describe the ordering criteria imposed on the client of operation invocations on the newly created object.

```
package org.riehle.diss.Figure;

/**
This interface represents the GroupFigureCreation.Client role type.

@roletype GroupFigureCreation.Client
@properties noop
**/

public interface GroupFigureCreationClient {
  // After calling a GroupFigure constructor, you may
  // add and remove as many Child objects as you wish.
  // You must finalize the initialization of a GroupFigure
  // with a call to initDone(). After this, you cannot
  // add or remove any Child object, unless you call
  // reinitialize() to reopen the GroupFigure object.
  // Calling reinitialize() will cause the GroupFigure
  // to drop all contained Figure objects.
}
```

Specification 5-4: Definition of the Java GroupFigureCreationClient interface to represent the GroupFigureCreation.Client role type.

A role type of a reusable role model is represented as a Java interface. The same argument that applies to a free role type of a class model applies here as well. Specification 5-5 shows the ObjectProperty.Provider role type.

```
package org.riehle.diss.ObjectProperty;

import java.util.Enumeration;

/**
This interface represents the ObjectProperty.Provider role type.

A Provider provides Object instances as Properties. A Property
has a name. A Client may ask for a property by name. It may set
a Property to the Provider using the Property's name. It may
request all names of accepted Property types and may test whether
a certain Property type is known and acceptable to the Provider.

@roletype ObjectProperty.Provider
@properties free
**/

public interface PropertyProvider {
  public boolean hasProperty(String name);
  public boolean acceptsProperty(String name, Object prop);
  public Object getProperty(String name);
  public Object getDefaultProperty(String name);
  public Enumeration getProperties(); // collection of Object

  public boolean acceptsPropertyType(String typeName);
  public Enumeration getPropertyTypeNames(); // collection of String

  public void setProperty(String name, Object prop)
     throws InvalidPropertyException;
  public void unsetProperty(String name);
}
```

Specification 5-5: Definition of the Java PropertyProvider interface to represent the ObjectProperty.Provider role type.

Classes are simpler to represent. They are expressed either as a Java interface or a class. Which variant to choose depends on the number of implementations of the design-level class [Rie97d, Rie97e,

RD99a, RD99b]. In the example, the class Figure is represented as a Java interface, for which different Java classes exist as different implementations.

In both cases, it is possible to tie in the different role types that a class must provide. If the class is represented as a Java interface, it inherits from the interfaces that represent (some of) the role types it is to provide. If the class is represented as a Java class, it simply implements them. In addition, non-reusable role types are textually directly embedded into the Java interface or the Java class.

Role models that are considered reusable should get their own Java package. The package introduces a convenient namespace for the role types of the role model. A framework should also get its own package. It ties in reusable role models by means of import statements. In the example, ObjectProperty is considered a reusable role model and thus gets a package of its own. The Figure framework also gets its own package, from which it imports these two other packages.

5.4.4 Extending C++ with role modeling

In contrast to Java, C++ does not offer an explicit interface concept. However, it provides the concept of pure virtual functions (polymorphic operations without implementation) and multiple inheritance. Taken together, these concepts can be used to emulate the Java interface concept. This also meets the intent of the designers of the C++ programming language [Str94].

The Java ObjectProperty interface from above is expressed in C++ as shown in Specification 5-6.

```
#include "Object.h"
#include "String.h"
#include <vector.h>

/**
This interface represents the ObjectProperty.Provider role type.

A Provider provides Object instances as Properties. A Property
has a name. A Client may ask for a property by name. It may set
a Property to the Provider using the Property's name. It may
request all names of accepted Property types and may test whether
a certain Property type is known and acceptable to the Provider.

@roletype ObjectProperty.Provider
@properties free
**/

class PropertyProvider : public Object {
  public bool hasProperty(string name) =0;
  public bool acceptsProperty(string name, Object* prop) =0;
  public Object* getProperty(string name) =0;
  public Object* getDefaultProperty(string name) =0;
  public vector<Object*>::iterator getProperties() =0;

  public bool acceptsPropertyType(string typeName) =0;
  public vector<string>::iterator getPropertyTypeNames() =0;

  public void setProperty(string name, Object* prop)
     throw InvalidPropertyException =0;
  public void unsetProperty(string name) =0;
};
```

Specification 5-6: Definition of the C++ PropertyProvider interface to represent the ObjectProperty.Provider role type.

Using this substitute for Java interfaces, the whole Java discussion also applies to C++. The Java implements relationship becomes a regular inheritance relationship. C++'s inheritance mechanisms allows multiple inheritance, so that there are no constraints on the number of interfaces that might represent a role type and that might be provided by a class.

Packaging in C++ is done using files and name spaces. The namespace keyword lets developers scope classes with a name space so that references to these classes must be qualified using the name of the name space.

5.4.5 Extending Smalltalk with role modeling

The representation of the role modeling concepts in Smalltalk differs from the one used for Java and C++. Smalltalk does not provide interfaces, multiple inheritance, or name spaces. The following discussion uses a generic Smalltalk dialect and ignores vendor-specific extensions.

In Smalltalk, it is a convention to express abstract methods by implementing them with the statement *self subclassResponsibility*. When executed, this statement invokes the subclassResponsibility method of the object, which will usually bring up the debugger to indicate that a method was executed for which a subclass failed to provide an implementation.

A class with only abstract methods is an abstract class. It can be used to represent a role type. Specification 5-7 describes the role type FigureObserver.Observer in Smalltalk.

```
"A FigureObserver is an object that is notified by a
Figure object about state changes of that Figure object.

@roletype FigureObserver.Observer
@properties free"

Object subclass: # FigureObserver
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

"FigureObserver publicMethods"

figureChanged: figureEvent
  self subclassResponsibility

figureRemoved: figureEvent
  self subclassResponsibility

figureInvalidated: figureEvent
  self subclassResponsibility
```

Specification 5-7: Definition of the abstract FigureObserver class in Smalltalk to represent the FigureObserver.Observer role type.

A class composes role types. Java and C++ do this using multiple inheritance. Smalltalk does not provide multiple inheritance, at least not the mainstream Smalltalk implementations VisualWorks and VisualAge. Smalltalk's elaborate metalevel architecture makes it possible to introduce multiple inheritance without breaking the language. Schäffer presents an in-depth discussion of how to do so [Sch98]. However, the introduction of multiple inheritance is proprietary and frequently problematic.

As an alternative, developers can use tools to copy method definitions from an abstract class representing a role type to a regular class [RS95]. Such tools can emulate multiple inheritance, but require additional maintenance efforts. However, tool support is better than copying methods by hand, which requires an even higher maintenance effort to keep definitions synchronized.

For structuring a class interface, Smalltalk uses *method categories.* A method category is a (sub-)set of methods from the overall set of methods of a class. Each class has a set of method categories. A programmer assigns a method to a method category. It is customary that a method is assigned to exactly one method category, but most tools do not enforce this, and it is possible to have a method be an element of more than one method category.

A method category can be used to group all methods of a particular role type. On a design-level, the Figure class provides the role types Figure.Figure, FigureHierarchy.Child, FigureObserver.Subject, Graphics.Client, FigureChain.Predecessor, and ObjectProperty.Provider. On a Smalltalk level, the Figure class provides the corresponding method categories.

The method category GraphicsClient to represent the free no-op role type Graphics.Client is empty, because no methods are assigned to it. It is sensible to make a role type without methods explicit, even if it leads to an empty method category. This is equivalent to annotating the class interface with no-op role types.

It is a Smalltalk convention to have a method category called “Accessing”. This method category provides all methods used for simple get and set access to instance variables of an object. From a role modeling perspective, the Accessing method category is problematic. It does not represent a role type but rather cuts across the whole set of role types provided by a class. The Accessing method category provides a good overview of the abstract state of the class, even though it is not guaranteed to give a complete overview.

It does not make sense to break this long-established convention. However, it would be equally annoying, if a method category that represents a role type would be incomplete, because its get and set methods are assigned to the Accessing method category rather than the role type’s method category. It is therefore best to keep the methods in both categories, serving both conventions equally well.

Smalltalk does not provide name spaces. Rather, developers use prefixes to tag classes as belonging to a particular name space. This can be used to emulate name spaces. However, it is fairly impractical to have a prefix of several words in front of the actual class name, so that name spaces are only used on a coarse-grained basis.

5.4.6 Extension properties

The following discussion shows how the extension properties asked for in Section 5.2 are realized by the programming language extensions. It thereby describes the rationale behind the extensions.

- *Robustness.* All three programming language extensions are robust with respect to being used by developers who do not know about the role modeling extension. The extensions use only such features of the programming language that are widely understood.

  In the case of Smalltalk, a metalevel extension could introduce new role modeling concepts on top of the existing language. Such an extension is not robust, because it requires developers to understand the concepts and handle them appropriately. Section 5.1 discusses the pros and cons of this decision.

- *Inconsistency.* A program written in any of the three programming languages may be inconsistent from the point of view of the role modeling extension. However, if it is a proper program, it still runs without problems. Therefore, developers can mix and match role modeling with traditional concepts.

- *Incompleteness.* Source code may be inconsistent from a role modeling perspective and therefore incomplete as well.

- *Tool support.* There is no dedicated tool support for role modeling in current mainstream programming environments. These environments only support the immediate language features. Therefore, the programmer must maintain a role modeling perspective when implementing designs.

  However, maturing programming environments tend to introduce new tools and concepts that reflect the pragmatics of using programming language constructs better than could be derived from the language definition. we can expect role modeling to provide significant input to these tools.

- *Bridge to design.* Annotations provide sufficient means for bridging back from implementation to design, both for developers to look up a particular documentation aspect, and for tools for code generation and round-trip software engineering.

Except for Smalltalk, none of the programming languages provides an extension mechanism that lets us gracefully introduce new language features. As a consequence, we take a conservative approach and use basic language concepts only. Except for the area of tool support, all desirable properties are achieved.

5.5 Summary

This chapter presents the extension of UML, Java, C++, and Smalltalk with the role modeling concepts for framework design. Using these extensions, developers can work in any of these design notations or programming languages and apply the new role modeling concepts where appropriate. The extensions are robust and can cope with inconsistency and incompleteness.

By choosing to provide extensions of industry standards rather than introducing a new isolated approach, Chapter 5 demonstrates that role modeling for framework design is an evolutionary step ahead in the definition of design methods. It adds to existing methods, but does not replace them. It can be used without invalidating existing investments.

This chapter is the last theory chapter. The following chapters are case studies that show how role modeling works in practice.

6

Case Study: The Geo Object Framework

This chapter presents a first case study: the Geo Object framework. This framework is the root framework of the Geo system, a distributed object system based on an explicit metalevel architecture. The Object framework comprises the most fundamental classes of a Geo system: Object, MetaObject, Class, MetaClass, and others. The chapter presents the framework and discusses the experiences of its development team with designing, learning, and using the framework. While doing so, the team made explicit use of role modeling and the role–model–based catalog of design patterns. This framework is of particular interest, because many systems today have such a root framework. The Geo system is only one example; other examples are Smalltalk and Java in general, and various C++ application frameworks.

6.1 Case study overview

The Geo system is a distributed object system that has an explicit metalevel architecture. It is organized as a service architecture and implemented using frameworks. All of the frameworks build on the Object framework, which is discussed in this chapter.

6.1.1 Project history

The Geo system was developed at Ubilab, UBS AG, during 1997. Its primary goal was to evaluate and demonstrate novel design concepts for a metalevel architecture that allowed programmers to customize object behavior on a per-instance basis. However, the metalevel architecture is only a small part of the system. The largest part are the service frameworks that made Geo a distributed system.

The Geo system was part of a project that was jointly run by Kai-Uwe Mätzel and me. The Geo frameworks were designed and implemented by a team that over time consisted of the following members: Roger Brudermann, Bruno Essmann, Frank Froese, Patrizia Marsura, Kai-Uwe Mätzel, and me. In the following, "the team" refers to this team, and "team members" refers to one or several of its members, depending on the context.

The team cooperated with the GBO project, another UBS IT project. GBO ("Global Business Objects") was a much larger project [BGR96a, BGR96b] that had been started in 1996. It aimed at providing a new distributed object infrastructure for UBS' worldwide IT operations. GBO was to integrate the many existing heterogeneous systems through means of a metalevel architecture. The goal of developing the Geo system was to feed back the results from the metalevel architecture into GBO. However, in course of the UBS/SBC merger development was stopped, and the project eventually was terminated.

As a consequence, different parts of the Geo system have different maturity. The framework presented in this chapter is very mature. It has formed the foundation for most other frameworks of the Geo system and proved its reliability.

6.1.2 The case study

The Geo Object framework is the root framework from which all other frameworks in the Geo system inherit. It defines key classes like AnyObject, from which every other class in the system either directly or indirectly inherits. The Object framework defines the overall structure of the metalevel architecture, but also allows for customization so that extensions can introduce their own custom-tailored versions.

In Geo, every object has a metaobject that can be customized with RequestHandler objects. Each such request handler manages one particular functional aspect of the execution of a method of the baseobject. The metaobject itself coordinates the overall method execution process. This setup is similar to the one of CodA [McA95a, McA95b], with the difference that the project team was not interested in the primitive aspects of an object (a minimal object model), but rather in what an object needs in a distributed system. So the functional aspects team members were thinking about were logging, security, error handling, etc.

The Object framework is based on experiences with similar frameworks and on-going refinement during the development of the Geo system. Every other framework in the Geo system directly or indirectly has to use it, so everyone who works with the Geo system has to learn it.

In the design and implementation process of Geo, the team members made frequent use of role modeling. While doing so, they also used the design patterns catalog that presents patterns using role models [Rie97a]. An abbreviated version of this catalog is available as Appendix D.

6.1.3 Chapter structure

The next section describes the Object framework using the framework documentation template from Chapter 4. The section also shows how the Object framework has been used by one higher-level framework, the Object Transport Service framework. The final section then describes the team's experiences with using role modeling in the development of the Geo Object framework and two of its extensions.

6.2 The Geo Object framework

The Geo Object framework provides basic object and class management functionality for a Geo process. It defines the classes AnyObject, AnyClass, and MetaClass for objects and classes, MetaObject and RequestHandler for handling and executing object requests, and ObjectRegistry and ClassManager for managing objects and classes. These are the most important classes that represent the interface architecture of the framework; the implementation of the framework provides many more classes.

6.2.1 Framework overview

The Object framework is the root framework of all Geo frameworks. It is both a white-box and a black-box framework. It is a white-box framework, because new classes necessarily inherit from AnyObject or AnyClass, and it is a black-box framework, because it provides readily usable implementations of MetaClass, ObjectRegistry, and ClassManager.

The framework provides different categories of functionality.

- *Standard AnyObject functionality.* AnyObject and AnyClass define what it means to be an object in a Geo system. This comprises a wide set of functionality, directly offered to clients.
- *Request handling functionality.* AnyObject, MetaObject, and RequestHandler define how a request is executed by any instance of (a subclass of) AnyObject.
- *Object management functionality.* AnyObject, AnyClass, and ObjectRegistry define the minimal lifecycle management applied to any object in the system.
- *Class management functionality.* AnyClass, MetaClass, and ClassManager define additional lifecycle management that is applied to any class object in the system (class objects are instances of AnyClass).

Subsection 6.2.2 presents the class model of the framework. It comprises a discussion of the classes and the structural relationships they define for their instances. Subsection 6.2.3 then presents the role models that define the collaboration of instances of these classes.

6.2.2 Class model

Figure 6-1 shows the seven key classes of the Geo Object framework, together with their structural relationships.

These seven classes serve the following purposes:

- *AnyObject* defines functionality provided by every object in a Geo system. The functionality covered ranges from simple operations that provide an object identifier up to a metaobject protocol that lets clients request metadata about the object at hand. AnyObject relates to all other classes of the Object framework. It is the direct or indirect superclass of every class in the system.
- *MetaObject* is the (super-)class of all metaobject classes in a system. Every instance of AnyObject has a metaobject. An instance of the MetaObject class defines the configuration and execution of incoming requests of the baseobject the instance is responsible for. A metaobject handles accepting, queueing, processing, and dispatching of a request. It allows custom-tailoring the runtime behavior of its baseobject.
- *RequestHandler* is the (super-)class of all request handlers. Every metaobject may have several request handlers associated with it. A metaobject runs any incoming request by its request handlers. A request handler may change or veto a request. A request handler is devoted to a particular

request execution concern, for example to logging the request, decrypting it, or authenticating the originator.

- *AnyClass* is the (super-)class of all class objects. A class object represents a specific class. Every instance of AnyObject has a class object. Each class object may have several instances. In contrast, to class objects, there is one dedicated metaobject for each baseobject (at least conceptually). AnyClass defines all the class-level functionality needed by an object. It provides metadata, identifier, stubs, etc.
- *ObjectRegistry* is the class of the singleton object registry. AnyClass is responsible for creating and deleting its instances. It manages these objects through the use of the object registry. ObjectRegistry focuses solely on providing convenient access to objects using either a unique identifier or a name that an object has been given.
- *MetaClass* is the (super-)class of the class object of all class objects. It is a subclass of AnyClass. The instance creation process of AnyClass is too simple for classes, so MetaClass provides more elaborate functionality and ensures that all pieces fit together. MetaClass implementations vary with each system type (front-end browser, application server, and distributed service provider).
- *ClassManager* is the class of the singleton class manager. The class manager provides a convenient access interface to all classes of a system. A class can be retrieved using its fully qualified name. If needed, the class manager lazily creates a requested class using the metaclass object.

The following role models detail the role types from the role type sets of these classes.

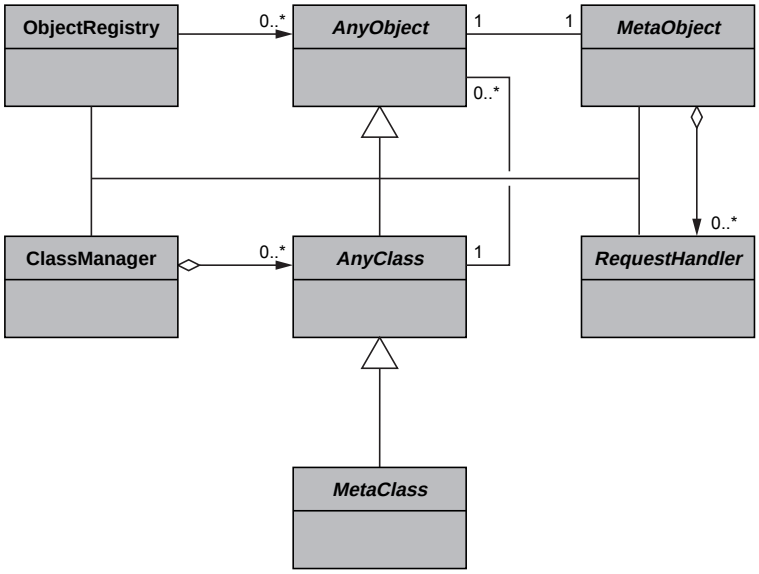

Figure 6-1: Class model of the Geo Object framework without role models.

6.2.3 Role models

The overall set of role models defined in the Geo Object framework can be partitioned into several categories:

- *AnyObject role models.* These role models are binary: they relate a client role type with a role type provided by AnyObject that provides some basic functionality.
- *AnyObject and AnyClass role models.* These role models relate AnyObject and AnyClass instances with each other, with the focus on AnyClass as the primary service provider.

- *Request Handling.* These role models deal with executing an incoming request. Represented as an object, the request is processed by RequestHandler objects before it is dispatched to its target.
- *Instance management.* These role models deal with maintaining objects as instances of classes in an object registry. AnyClass and ObjectRegistry work together to maintain the objects.
- *Class management.* These role models deal with the management of class objects. Management includes creating and maintaining class objects in a registry. MetaClass and ClassManager do it.

6.2.3.1 Role models of basic object functionality

The first part describes the standard AnyObject functionality, and the second part describes the AnyObject/AnyClass collaboration. Figure 6-2 shows the role models that are relevant in this context.

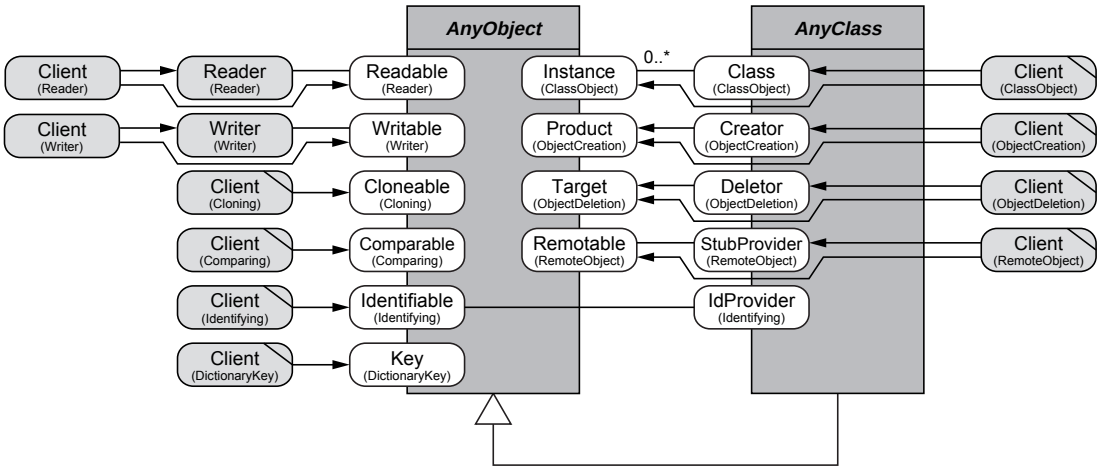

Figure 6-2: AnyObject and AnyObject/AnyClass role models.

The following first part describes the AnyObject role models.

- The *Reader* role model serves to initialize an object from a data-oriented backend. A Client tells an object, the Readable, to read and subsequently initialize its state from another object, the Reader. It is an instance of the Serializer pattern. AnyObject provides the Readable role type, and Client and Reader are free role types.

  The Reader role type provides operations to start the reading process for a Readable, as well as operations for the Readable to request its state. The Readable role type in turn provides operations to receive the Reader from which to read its state, as well as post-processing operations after the reading took place. Graphs of objects are read by a recursive descent into the graph, realized by the Reader and Readable alternatingly calling each other.

- The *Writer* role model serves to write the state of an object to a data-oriented backend. A Client tells an object, the Writable, to write its state to another object, the Writer. It is an instance of the Serializer pattern. AnyObject provides the Writable role type, and Client and Writer are free role types.

  The Writer role type provides operations to start the writing process for a Writable, as well as operations for the Writable to write out its state. The Writable role type provides operations to receive the Writer it is to write its state to. Graphs of objects are written by a recursive descent into the graph, realized by the Writer and Writable objects, which alternatingly call each other.

- The *Identifying* role model makes an object provide a unique identifier (of it). A Client may request an identifier from an object, the Identifiable. If no identifier is present, the Identifiable re-

quests it from its class, the IdProvider. AnyObject provides the Identifiable role type, AnyClass provides the IdProvider role type, and Client is a free role type.

The Identifiable role type provides operations to get and set the identifier, as well as to compare two objects for identity with respect to their identifier. The IdProvider role type provides operations for the Identifiable to request an identifier. Typically, an object does not receive an identifier right from the start, but only when it is needed, for example, when it is referenced remotely for the first time.

- The *Cloning* role model serves to make shallow or deep copies of an object. A Client may ask an object, the Clonable, for a copy of it. AnyObject provides the Clonable role type, and Client is a free role type.
- The *Comparing* role model serves to compare two objects for equality. A Client may ask an object, the Comparable, to compare itself with another object. AnyObject provides the Comparable role type, and Client is a free role type.
- The *DictionaryKey* role model allows an object to act as a key. A Client can use the object as a Key. AnyObject provides the Key role type, and Client is a free role type.

The following second part describes combined AnyObject/AnyClass functionality.

- The *ObjectCreation* role model serves to create a new instance of a class. A Client object asks a Creator object to create the new Product object. AnyObject provides the Product role type, AnyClass provides the Creator role type, and Client is a free role type.

  The Creator role type defines operations for generically requesting a new instance. The Product role type defines generic initialization operations. For example, it receives a unique identifier and a reference to its class object. The ObjectCreation role model does not cover class-specific initialization, which must be done by a different dedicated role model.

  Upon successful creation, the Creator registers the new instance at the ObjectRegistry using the ObjectRegistry role model.
- The *ObjectDeletion* role model serves to delete an existing instance of a class. A Client object asks the Deletor object to delete the Target object. AnyObject provides the Target role type, AnyClass provides the Deletor role type, and Client is a free role type.

  The Deletor role type defines operations by which an existing object can be deleted. The Target role type defines a protocol for finalizing the object.

  Before deletion, the class object unregisters the object from the object registry.
- The *ClassObject* role model serves to provide metadata about an object. It is an instance of the Class Object pattern. A Client object can ask the Instance object about its Class object. AnyObject provides the Instance role type, AnyClass provides the Class role type, and Client is a free role type.

  The Instance object lets Clients request the Class object. The Client object may request metadata both from the Instance or the Class object. Only the Class object, however, provides the full set of metadata. For example, the Class object provides metalevel access to the attributes, behavior definition, and implementation of a given instance.
- The *RemoteObject* role model serves to make objects remotely referencable. A Client may ask a class, the StubProvider, to make an object, the Remotable, remotely accessible. AnyObject provides the Remotable role type, AnyClass provides the StubProvider role type, and Client is a free role type.

  The StubProvider role type provides operations to receive a stub for an object, decouple an object from the stub, etc. The Remotable role type provides operations to get and set the stub.

6.2.3.2 Role models of request handling

The following third part describes how incoming requests are handled and executed. The relevant role models are shown in Figure 6-3.

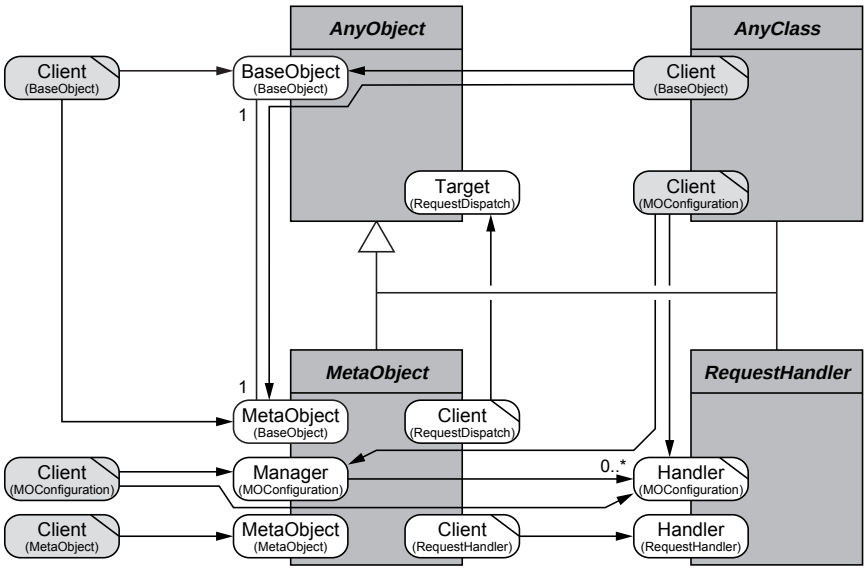

Figure 6-3: Classes and role models involved in the request handling process.

- The *BaseObject* role model defines how a Client may ask an object, the BaseObject, about its metaobject, the MetaObject. It is an instance of the Metaobject pattern. AnyObject provides the BaseObject role type, MetaObject provides the MetaObject role type, and AnyClass provides the Client role type. The Client role type is also a free role type.

  The BaseObject role type provides operations to get the metaobject. The MetaObject role type provides those operations that the baseobject wants to delegate to the metaobject, for example, error logging.

  Conceptually, there is one metaobject for each baseobject, but in practice, metaobjects can be shared among baseobjects. Therefore, all operations of the metaobject require a reference back to the baseobject they are to work on or provide services for.

- The *MOConfiguration* (MetaObjectConfiguration) role model defines how a Client object may configure a metaobject, the Manager, with a Handler for processing requests. MetaObject provides the Manager role type, RequestHandler provides the Handler role type, and Client is a free role type.

  The Manager role type provides operations for the Client to insert a Handler at a certain position in the chain of request handlers along which a request is passed.

- The *MetaObject* role model defines how a Client may ask a MetaObject to execute a request. MetaObject provides the MetaObject role type, and Client is a free role type.

  The MetaObject role type provides operations to receive an incoming request and to inquire about the current request processing state.

- The *RequestHandler* role model defines how a metaobject, the Client, passes a request along a set of Handler objects. MetaObject provides the Client role type, and RequestHandler provides the Handler role type.

  The Handler role type provides the operations to process a request. A handler might do nothing to a request, change it, or veto it.

- The *RequestDispatch* role model defines how a metaobject, the Client, hands over a Request object to its baseobject, the Target, for dispatch (also known as dynamic invocation interface). The Target dispatches the request to one of its operations. MetaObject provides the Client role type, and AnyObject provides the Target role type.

  The Target role type provides operations to receive the request object, which it then dispatches to the correct operation.

6.2.3.3 Role models of object and class management

The fourth part describes the role models for managing objects in an object registry. The fifth part describes role models for creating and maintaining class objects. Figure 6-4 shows the role models and the involved classes.

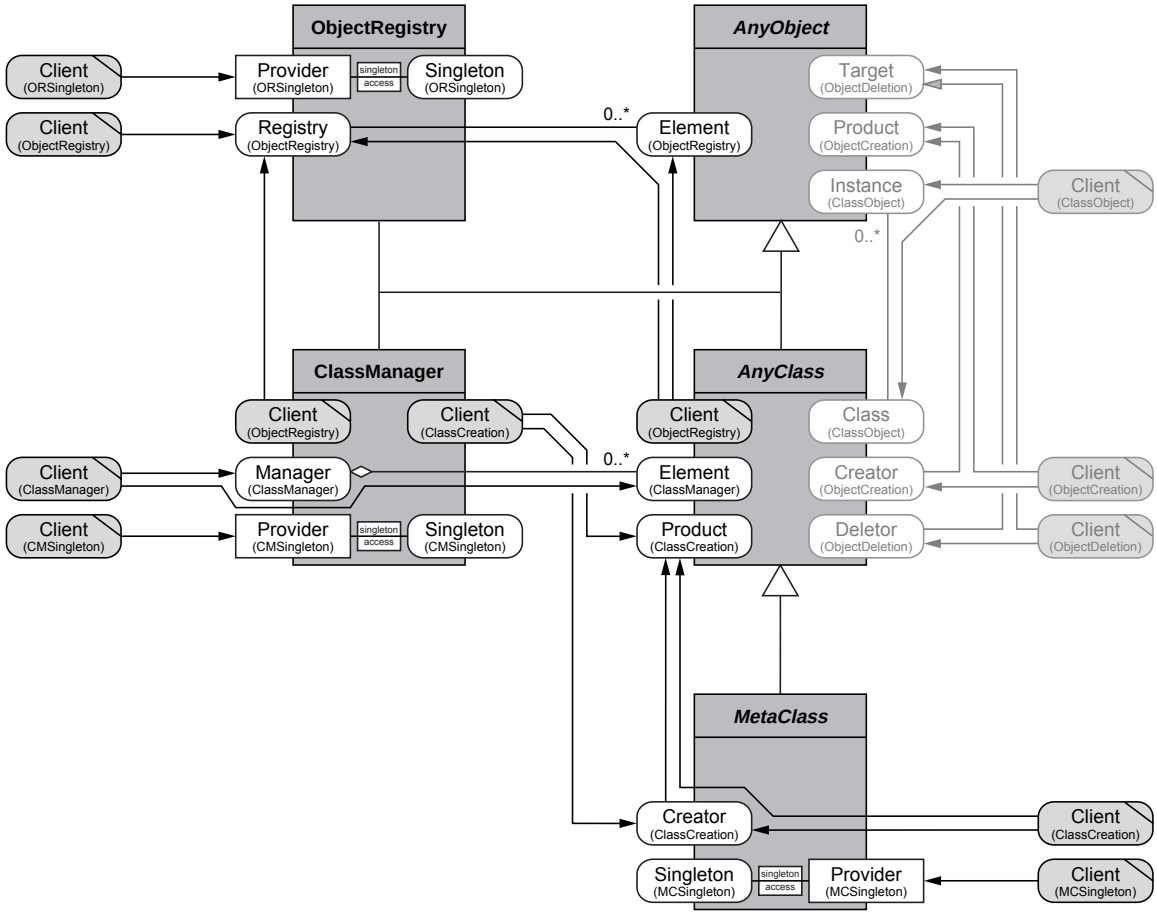

Figure 6-4: Classes and role models for managing objects and classes
(grayed-out role models are not discussed here).

The following fourth part describes the role models associated with the ObjectRegistry.

- The *ObjectRegistry* role model serves as a central access point to all objects in the process. It is an instance of the Object Registry pattern. A Client object may ask the Registry object about any particular Element object. ObjectRegistry provides the Registry role type, AnyObject provides the Element role type, and AnyClass provides the Client role type. Client is a free role type.

  The Registry role type provides operations to register and unregister Elements using object identifiers and names.

Within the framework, AnyClass provides the Client role type. AnyClass objects use it to register or unregister their instances at the ObjectRegistry.

- The *ORSingleton* role model (ObjectRegistrySingleton) serves to provide convenient access to the single ObjectRegistry object. It is an instance of the Singleton pattern. It is shown in Figure 6-4 using the singleton shorthand.

The following fifth part describes the role models for creating and managing classes.

- The *ClassCreation* role model serves to instantiate new class objects. A Client object may ask the Creator object to create a new Product object. AnyClass provides the Product, MetaClass provides the Creator, and ClassManager provides the Client role type. Client is a free role type.

  The Creator role type provides convenience operation to instantiate new class objects using the most common combinations of parameters. Class objects in the Geo metalevel architecture are configured with a large number of parameters. The Product role type provides initialization operations to receive these parameters.

  Creation of a class object is a special case of creation of a regular object, so the ObjectCreation role model is used to setup the Object related parts of the new class object. Also, the ObjectRegistry role model is used to register the new class object at the ObjectRegistry, using its identifier and class name.

- The *MCSingleton* role model (MetaClassSingleton) serves to provide convenient access to the single MetaClass object. It is an instance of the Singleton pattern and shown in the figure using the singleton shorthand.

- The *ClassManager* role model serves to provide a central access point to all classes of a process. A Client may ask the Manager for an Element. Class provides the Element role type, ClassManager provides the Manager role type, and Client is a free role type.

  The Manager role type provides operations to register and unregister class objects using their class name. If a class is not available, the class manager loads it on the fly.

  The ClassManager uses the ClassCreation role model and underlying system facilities for on-demand creating and loading a class that is not currently available in the system.

- The *CMSingleton* role model (ClassManagerSingleton) serves to provide convenient access to the single ClassManager object. It is an instance of the Singleton pattern and shown in the figure using the singleton shorthand.

Most of the role models are free role models. Virtually every framework makes use of the ClassManager.Client role type to retrieve classes by name. Other role types are taken up by more specialized frameworks. For example, the system configuration frameworks make use of the ClassCreation.Client role type, and others.

6.2.4 Built-on classes

The Object framework is a fundamental framework that does not build on any other framework. It uses several underlying Java APIs, though.

6.2.5 Example extension

If a class wants its instances to make use of system services and to be referenced remotely, it must directly or indirectly inherit from AnyObject. Therefore, the Object framework forms the foundation of nearly every other non-trivial class in a system.

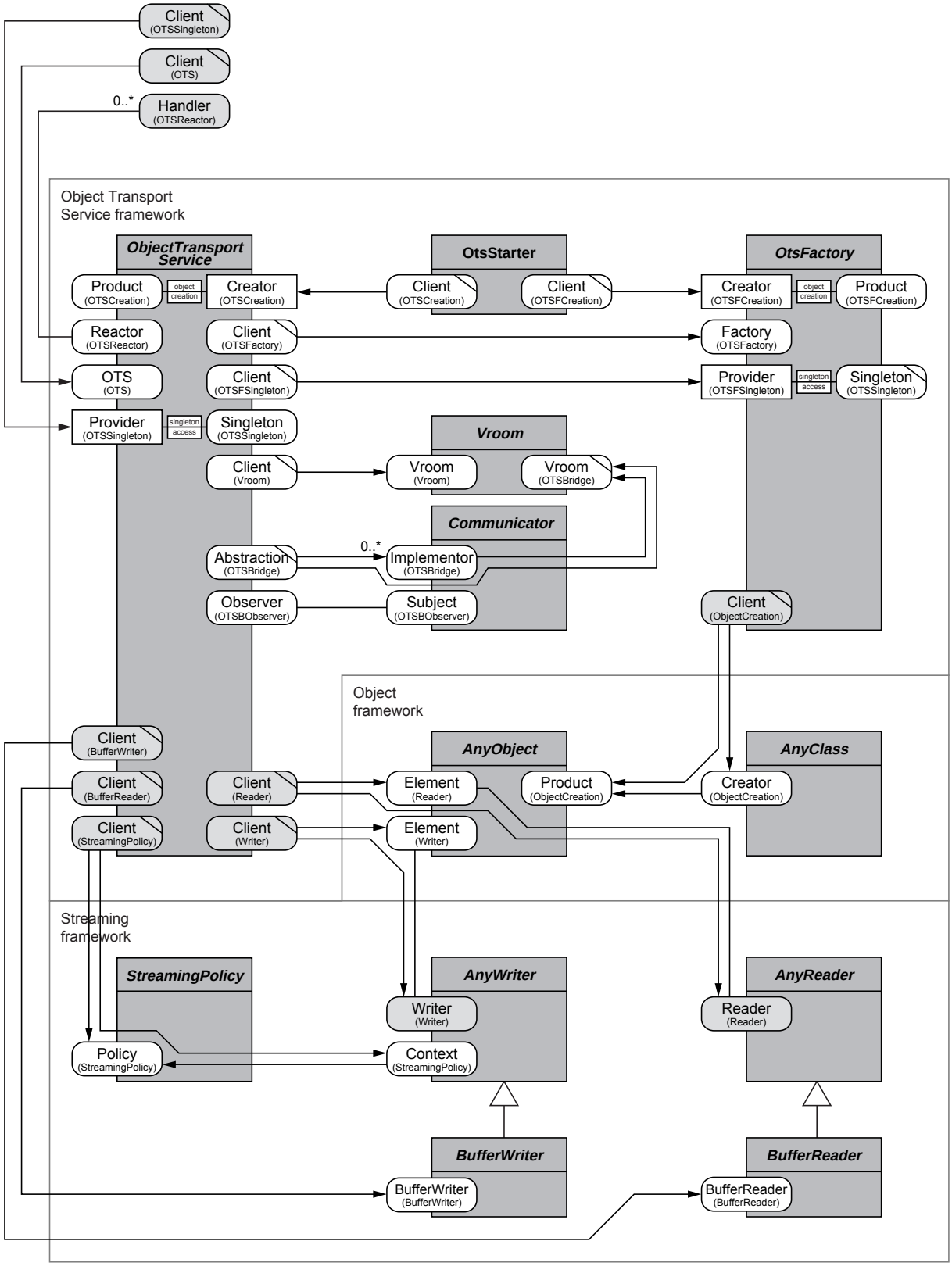

Figure 6-5: The OTS framework making use of the Object framework.

Figure 6-5 shows two example frameworks that both extend and build on the Object framework. (This is a simplified example). Another account of this example has been given in [RBGM99].

The two example frameworks that build on the Object framework are the Streaming and the Object Transport Service framework. An intermediate Service framework that also extends and builds on the Object framework is ignored.

The Streaming and the Object Transport Service framework are both extensions and use-clients of the Object framework. They are extensions, because their classes either directly or indirectly inherit from AnyObject, which is the primary extension-point class of the Object framework. They are use-clients, because they tie in with the Object framework using several different free role models.

The Streaming framework introduces classes that define and implement the Reader and Writer objects implied by the Reader and Writer role models. These classes tie in to the Object framework through the Reader and Writer role models. The Object Transport Service uses such Reader and Writer objects to marshal and unmarshal an object's state.

The Object Transport Service (OTS) framework ties in with the Object and Streaming frameworks using the Reader and Writer role models. It also uses the ObjectCreation role model of the Object framework. The OTS does not need more role models, because it copies objects across process boundaries, but does not migrate them (which is the task of the Object Migration Service acting on top of the OTS).

The Streaming and the OTS framework are straightforward examples of how frameworks build on each other. The use of free role models lets us precisely specify what is expected of clients. As the example shows, this helps clarify framework relationships.

6.3 Experiences and evaluation

During the development of the Geo system, the team used role modeling and the role–model–based catalog of design patterns to discuss the design of frameworks. The following subsections present and discuss the team's experiences with the use of role modeling for framework design.

6.3.1 Statistics of case study

The Geo Object framework provides the data shown in Table 6-1.

Number of classes	7
Number of role models	21
Number of pattern instances	11
Number of role types assigned to classes	41
Ratio of role types per class	5.86
Standard deviation of role types per class	3.80

Ratio of role models per class	3.0
Ratio of pattern instances per role model	0.52

Table 6-1: Raw data and computed figures from the Object framework.

The Geo Streaming framework provides the data shown in Table 6-2.

Number of classes	5
Number of role models	3
Number of pattern instances	1
Number of role types assigned to classes	6
Ratio of role types per class	1.2
Standard deviation of role types per class	0.4
Ratio of role models per class	0.6
Ratio of pattern instances per role model	0.33

Table 6-2: Raw data and computed figures from the Streaming framework.

The Geo Object Transport Service framework provides the data shown in Table 6-3.

Number of classes	5
Number of role models	10
Number of pattern instances	8
Number of role types assigned to classes	28
Ratio of role types per class	5.6
Standard deviation of role types per class	5.43
Ratio of role models per class	2.0
Ratio of pattern instances per role model	0.8

Table 6-3: Raw data and computed figures from the Object Transporter framework.

The discussion of the Object framework, the Streaming framework, and the Object Transport Service framework shows the key interface architecture only and omits classes of lesser importance and helper classes.

6.3.2 Complexity of classes

The Geo system has many complex classes. On the one hand, these are classes like AnyObject and AnyClass, which are complex due to the non-trivial number of object collaboration tasks their instances get involved in. On the other hand, these are the classes that serve as facades to a subsystem. Examples of facades are service interfaces like the one of the Object Transport Sservice and those of its dependent services like the Object Migration Service or the Remote Request Execution Service.

In both cases, the team found that being able to describe these classes using role types eased its work considerably. During the development of the Geo system, the team made the following observations:

- During design, team members found that role types and role models made defining the new classes easier than they remembered from earlier non-role–modeling–based experiences.
- Team members who used complex classes like AnyObject that they had not develop themselves found these classes easier to understand given a role modeling explanation of their behavior.
- Moreover, most of the team members felt that the complex classes were not only easier to understand, but also easier to use.

It is the team's experience that designing object systems is much like playing with scenarios of collaborating objects. The use of role types and role models lets a developer focus on the position of one object in such collaboration, without having to care about other roles. Later, a developer can care about the other roles, and carry out the composition of the role types. Of course, in the design process there is never such a clear-cut step-wise proceeding, but as a mental tool, describing classes using role types eases framework design significantly.

AnyObject is a prime example of a class where role modeling made it easier for team members to learn and use it. Some of AnyObject's functionality is of general interest, while some is not. Describing each piece of functionality as a role type (in the context of a role model) helped separate the different concerns, and communicate only those concerns of interest to a specific team member.

6.3.3 Complexity of object collaboration

The Geo system has many frameworks with complex object collaborations. For example, the Object framework is based on many complex collaboration tasks between AnyObject, AnyClass, and others. Other frameworks have equally complex internal collaborations. Extending and using the Object framework both requires understanding these object collaboration tasks.

During the development of the Geo system, the team made the following observations:

- In design discussions, the use of role models and design patterns supports the focus on individual object collaboration tasks. A developer can delay dealing with other tasks without critically loosing touch with the overall design. In fact, a developer can even switch between the collaboration tasks and work on the larger whole while staying focused on one particular issue.
- Learning and using a framework is facilitated by the clear definition of free role models. Developers who want to use a service learn about its interface and behavior in terms of role models, with each of the role models addressing a different issue. Separating these different issues reduces the complexity of learning the framework and putting it to use.

A general observation of the team was that role modeling achieves a higher degree of separation of concerns in the design of object collaborations than a traditional approach does, and that this kind of separation of concerns significantly reduces intellectual load, thereby easing the work task at hand.

The team also found that breaking up an overall collaboration into different role models supports being precise about the individual object collaboration tasks. In design discussions, thinking in terms of

role models and design patterns frequently helped team members derive conclusions about dependencies between classes, the framework, and clients that had initially been missed. Members attribute this to the separation of concerns achieved by role models and to the reduced intellectual load.

6.3.4 Clarity of requirements put upon use-clients

The Geo system comprises many frameworks. Black-box use of these frameworks is based on the use of free role types and free role models. The Geo system is implemented in Java, so free role types with operations are represented as Java interfaces. The other free role types are described in the documentation and as part of the annotations of the class interfaces.

During designing, learning, and using the frameworks, the team made the following observations:

- While designing a framework, explicitly thinking about what a client may see and what not helps focus on the relevant design issue at hand. This in turn helps reduce the complexity of designing the client interaction, because one can focus on one pertinent issue at a time. Role modeling increases the awareness of developers that they are defining the client interaction, so they work twice as thoroughly, because client interactions are more important than internal collaborations.
- While learning a framework, free role models communicate clearly how to use a framework from the outside. Not only can a developer use role modeling as a thinking tool in design, but also can he use it to review his design for clarity and preciseness. If a class has operations that do not belong to a role type, or if the client interaction is just one big role model, this usually is a good indication that the design needs further improvement.
- While using a framework, designing and programming in terms of free role types helps developers focus on what they are doing with the framework. Using a role type that is not a free role type indicates either a problem with the framework or with the developer's understanding of it. Using a free role type is helpful in its explicit description of what a developer can do and what he is supposed to do.

The use of free role types and free role models proved to be very helpful in defining the use-client interaction. This is something that would otherwise have stayed implicit. The team's experiences with using frameworks whose client interaction were defined in terms of free role models are favorably with respect to their effectiveness. For the first time team members had an explicit part of the design to turn to and to learn from how to use the framework.

6.3.5 Reuse of experience

The following two forms of reuse occurred in the development of the Geo system.

- *Reuse of experience through design adaptation.* In this form of reuse, a team member recalls some prior experience and adapts it to the current problem at hand.
- *Reuse of experience through design patterns.* In this form of reuse, a team member recognizes a common problem and instantiates a design pattern to solve it.

The following two subsections examine the team's experiences with these two forms of reuse in the development of the Geo system.

6.3.5.1 Reuse of experience through design adaptation

Reuse of experience through design adaptation occurs when designs are similar to designs from prior experience. Two examples are the design of the Object framework, and the design of the service interfaces.

- During the design of the AnyObject and AnyClass classes team members drew heavily on similar classes they had seen in Java, Smalltalk, and several C++ application frameworks. They all have root classes Object and Class, and they all serve similar purposes.
- During the design of some of the service interfaces, team members reused their experiences with earlier service interfaces that have a similar structure. The Object Transport service, the Object Migration service, and the Remote Request Execution service interface all provide similar kinds of role types (primary domain functionality, a callback role type, and a Singleton access role type).

Role modeling made this form of reuse easier than team members could imagine possible with a traditional approach. In all cases, the relevant functionality was represented and understood in the form of role models. The team's experience is that a role model represents a precise and convenient design fragment of possible reuse that is precisely at the right level of granularity.

Team members either decided to adapt a role model from a design fragment of prior experience, or they decided to drop it. Once a team member had decided to adapt and thereby reuse it, he worked within the boundaries set up by the role model. Here, role models proved to be excellent design elements of possible reuse. They are much better suited than classes, which one has to break up into pieces to arrive at something similar to role models before one can start reusing them.

The use of role models also made team members not forget the client side and therefore helped them even further to reuse prior experience.

6.3.5.2 Reuse of experience through design patterns

The Geo frameworks exhibit a high design pattern density [RBGM98]. The team made use of design patterns in many instances, both by using the original class-based design pattern catalog [GHJV95] and by using the role–model–based version of a design pattern catalog [Rie97a]. This use of design patterns made the team's design efforts more effective by allowing team members to communicate more effectively [BCC+96, Vli98]. This increased productivity and the overall quality of work.

The Object framework as illustrated here has 7 classes and 21 role models. Conservatively counted, of these 21 role models, 11 are pattern instances. If one counts role models like Cloning, Comparing, or DictionaryKey as instances of yet undocumented patterns, the pattern/role model ratio increases further. The Object Transport Service framework features similar numbers, and so do most other frameworks.

The clear structure of the frameworks and the preciseness of their role models are a result of the team's understanding and use of design patterns. Because such a large percentage of the functionality of a framework could be described using design patterns, a large part of the framework had a clear structure and well-defined meaning early on. The remaining functionality became much easier to define in terms of role models, because there were less holes left and the boundaries of these holes were much better defined.

Team members worked most of the time using the role–model–based version of the design pattern catalog. The role model form made composing design patterns much easier than possible with classes. When using the class-based form, a team member usually had to carry out an intermediate step in terms of mapping from classes to functionality and responsibilities and then to the concrete situation where the pattern was to be applied.

The use of role–model–based design patterns allowed team members to drop this intermediate step. The pattern's role model structure is directly applicable to any concrete design situation, because the role types from the pattern definition are directly instantiated in the context of the specific situation.

Also, recognizing an applied pattern became easier using the role–model–based version. Mapping back from the framework is less complicated, because the role model as an instance of a pattern is right there in the design, while a class-based structure still requires an intermediate interpretation step of which class is what participant.

6.3.5.3 Conclusion on reuse of experience

During the development of the Geo frameworks, it was the team's experience that role modeling makes it easier to reuse prior experience, be it in the form of adapting old designs or applying design patterns, than it could imagine possible with a traditional class-based approach.

7

Case Study: The KMU Desktop Tools Framework

This chapter presents a second case study of role–model–based framework design. It describes the Tools framework of the KMU Desktop project (KMU = “Kleinere und Mittlere Unternehmen”, a Swiss-German abbreviation for small and medium-size enterprises). The KMU Desktop project develops a system to support the corporate customer credit process of UBS AG, a large international bank. A first version of the Tools framework was designed and implemented using a traditional class-based approach. After about a year of use, a redesign was carried out using role modeling and the role–model–based patterns catalog as an aid. The chapter presents both designs as well as the redesign team’s experiences during the redesign process. Also, the chapter compares the two designs and analyzes how role modeling helped to reach the much cleaner redesign.

7.1 Case study overview

UBS AG is a large bank, currently (mid 1999) the largest bank in the world with respect to assets under management. The KMU desktop system is an interactive software system that supports the credit management of UBS for small and medium-size corporate customers. Credit officers use it to determine whether a credit is to be granted, to assess and control its risk, and as a support of the whole granting and reviewing process.

7.1.1 Project history

The corporate customer division of UBS has undergone major changes in recent years, and new software was to reflect these changes and to help account managers and credit officers work more effec-

tively. The KMU Desktop project's mission was to develop this new software support. The project's primary focus was to support credit management of small and medium-size corporate customers of UBS.

The project's approach to analysis, design and implementation is based on the Tools and Materials metaphor [RZ95, RZ96]. Applications are implemented using Smalltalk on Windows-based PC clients, C++ on Solaris servers, and CORBA as middleware. The overall architecture is a three-tier client/server architecture. The design approach is based on frameworks.

7.1.2 The case study

A particularly important framework is the Tools framework. It is used to develop client-side software tools, which account managers and credit officers use in their daily work. A first version of this framework was developed in-house and finished in April 1997. The framework contributed significantly to an increase in productivity, but it was also difficult to use. Framework users had problems understanding and properly using it. After about one year of, project management decided to redesign the framework to overcome the existing problems with using it.

In March 1998, a team of three developers carried out the redesign. This redesign team consisted of two of the original developers, Gregory Hutchinson and Birgit Rieder, and me. The redesign team first analyzed the existing framework, determined its functionality and the problems developers had using it, described this functionality using role models, added new functionality and changed existing one, and recomposed the pieces to arrive at a new framework design.

In the following, the word "team" refers to the redesign team mentioned above. "Framework developer" refers to either Gregory Hutchinson or Birgit Rieder, or both. "Framework user", or user for short, refers to other developers from the KMU Desktop project that are using the Tools framework to build software tools.

7.1.3 Chapter structure

First, this chapter presents the original framework. The framework is described using the original class-based documentation available to framework users. We add to this description what I learned from my colleagues from the redesign team. This discussion includes the problems framework users had using the framework.

Then, the chapter describes the new revised framework using role modeling. The framework structure changed in many important aspects, but still, one can recognize the original framework and its intent. The role–model–based description makes use of design patterns terminology, based on the catalog of role model patterns [Rie97a]. An abbreviated form of this catalog is available as Appendix D.

Finally, the experiences of the redesign team with the redesign process and the experiences of users of the new framework are presented and analyzed.

7.2 The original Tools framework

This section presents the original Tools framework, as described by its developers. The original design had been carried out using traditional class-based modeling. In addition, Smalltalk method categories

had been used to structure the class interfaces. Also, the developers had used design patterns occasionally. However, they had not used role modeling.

7.2.1 Framework overview

The Tools framework is a framework used to build software tools for interactive software systems based on the Tools and Materials Metaphor [RZ95, RZ96]. It is a white-box framework.

Software system users use tools to change an underlying domain model called the materials (of work). Examples of tools are form editors. Form editors work on forms, their material. Other examples of tools are customer browsers, which serve to browse a list of customers, and risk assessment tools, which serve to determine the risk of a loan (current or applied for by a customer).

7.2.1.1 Software tools

A software tool is built from a *hierarchy of tool components.* Every tool component represents a specific part of the overall tool's functionality. Every tool component provides both a user interface and the functionality behind it to access and manipulate parts of the underlying domain model. At any one time, users interact with one tool component from the hierarchy. The tool component carries out user requests. Such a user request might cause complex control flow within the tool component hierarchy, possibly involving all components up to the root component of the hierarchy.

A tool component consists of at least two objects: *one functional part object (FP object)* that represents the functionality of the tool component, and *one or more interaction part objects (IP objects)* that provide the user interface to use the tool component's functionality. Thus, a tool component does not manifest itself as a single object, but rather as one or more IP objects with one FP object. When speaking of a tool component, typically the component's FP object is meant that represents the component.

A software tool is represented to its environment by the FP object of the root tool component, the so-called root-FP object. The tool component hierarchy may be arbitrarily deep. In practice, it seldom goes beyond three levels. The root-FP object manages the overall tool. Every tool component is responsible for managing its subordinate components (sub-components). In the component hierarchy, every FP object receives two types of information from sub-components. First, an FP object receives user requests from sub-components that could not be handled. Second, an FP object receives notifications about state changes from sub-components based on successfully executed user requests. The overall structure follows the Bureaucracy pattern [Rie98].

As an example, consider a simple NoteBrowser tool. It consists of one root tool component, the NoteBrowser tool component, and two subordinate NoteLister and NoteEditor tool components. The NoteLister component presents a list of notes to choose from, and the NoteEditor component lets users edit a note selected in the NoteLister. The object structure of the tool is depicted in Figure 7-1.

The object pair (noteBrowserIP, noteBrowserFP) from Figure 7-1 forms the root tool component, and the object pairs (noteListerIP, noteListerFP) and (noteEditorIP, noteEditorFP) form the two sub tool components.

Figure 7-2 depicts the original Tools framework, as taken from the documentation. A number of minor adjustments were made, mainly to make the design more readable and to correct omissions. Also, the class names have been translated to English where necessary.

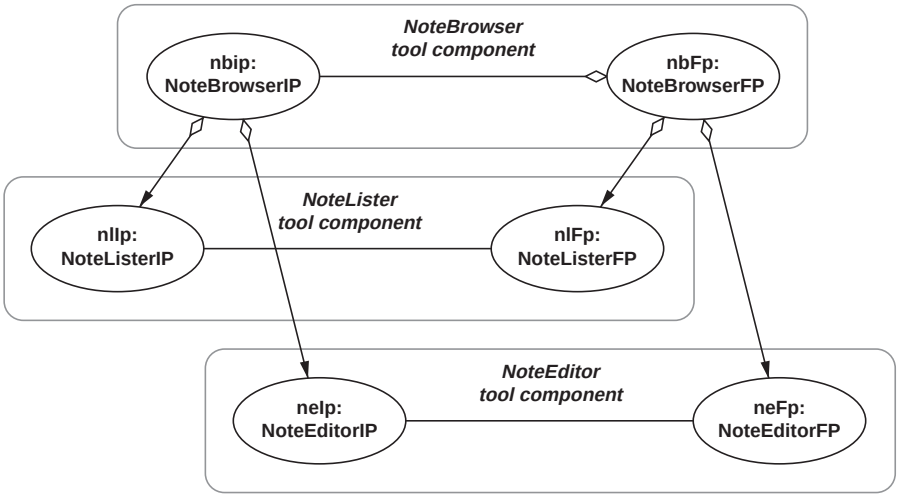

Figure 7-1: Example object structure of a simple NoteBrowser tool
(the gray boxes are a visual aid to mark the extent of a tool component).

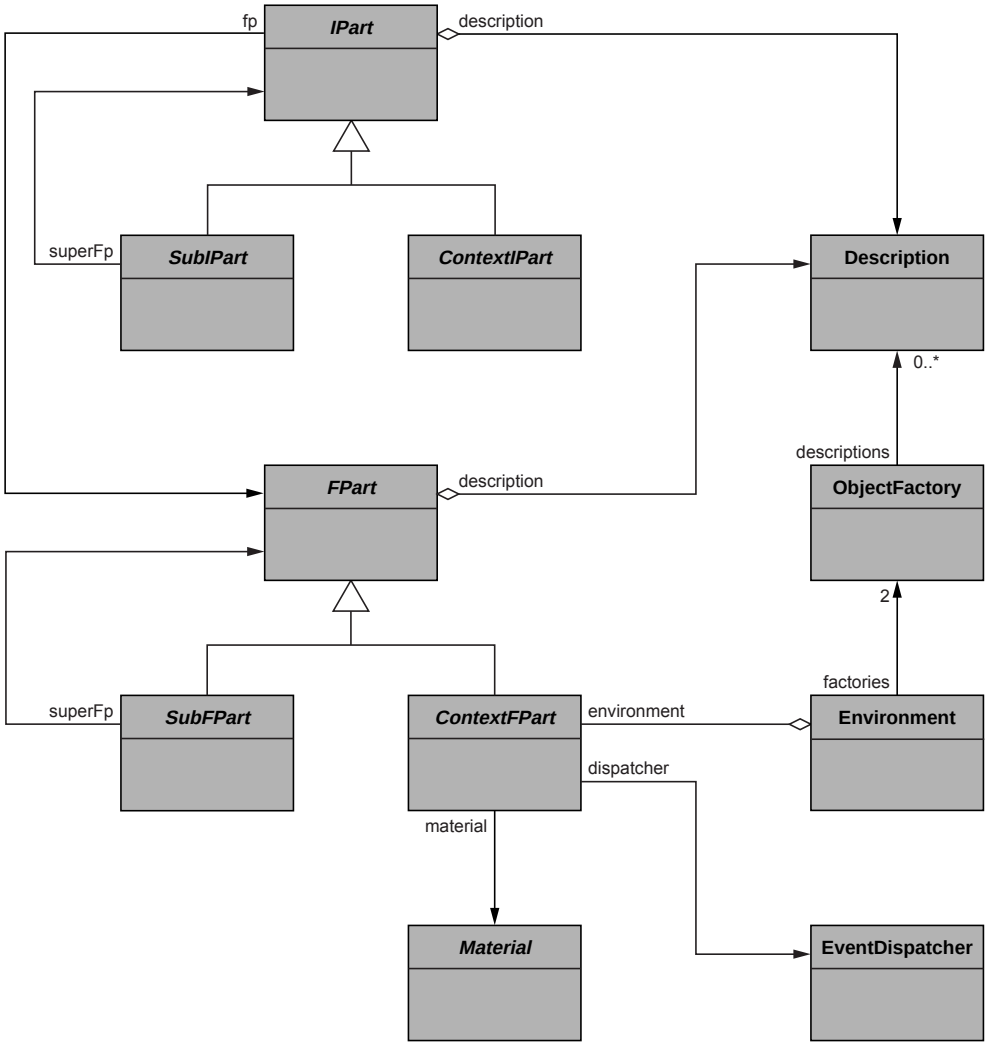

Figure 7-2: The KMU Desktop Tools framework.

The framework classes for tool construction comprise the interaction part classes IPart, SubIPart, and ContextIPart, as well as the functional part classes FPart, SubFPart, and ContextFPart. The framework context comprises the classes Material, Environment, EventDispatcher, ObjectFactory, and Description.

As discussed, a tool component comprises one FP object (functional part object, FPart instance), and one or more IP objects (interaction part objects, IPart instances). An FP object must always be an instance of a concrete subclass of FPart, and an IP object must always be an instance of a concrete subclass of IPart.

ContextFPart is the superclass of all root-FPart classes, so if an FP object is to be the root object of the FP object hierarchy, its must be an instance of (a subclass of) ContextFPart. Similarly, a root-IP object must be an instance of (a subclass of) ContextIPart. Also, an FP object that is an object in the FP hierarchy, but that is not the root-FP object, must be an instance of (a subclass of) SubFPart. Similarly, an IP object that is an object in the IP hierarchy, but that is not the root IP object, must be an instance of (a subclass of) SubIPart.

In the NoteBrowser tool example, NoteBrowserFP and NoteBrowserIP are subclasses of ContextFPart and ContextIPart, respectively, and NoteListerFP and NoteListerIP, and NoteEditorFP and NoteEditorIP are subclasses of SubFPart and SubIPart respectively.

7.2.1.2 Environment integration

An environment object, the sole instance of the Environment class, manages software tools. The root-FP object represents a tool; it is an instance of a subclass of ContextFPart. Thus, ContextFPart defines the interface through which the environment communicates with a tool.

The environment object receives requests for tool creation from a desktop (not shown in a figure). The request provides a name for the tool (a string), but not the tool nor its class. The environment uses tool specifications and the object factory to map the name onto a tool class and to instantiate the tool class. The tool class must be a concrete subclass of ContextFPart. Tool specifications are instances of class Description, and the object factory is the sole instance of class ObjectFactory.

The process of determining a tool class from a specification is described by the Product Trader pattern [BR98]. A description object can calculate a key from the parameters it originally received from a client (an example of a key is the tool name string). The key identifies the tool class. It is typically unique. For each description, the object factory determines the corresponding class, either by looking it up in pre-configured tables, or by walking over the class hierarchy matching each class with the key.

Once the root-FP object has been instantiated, the environment object properly initializes the tool. First, it repeats the object creation process for the root-IP object. This time, the key is the root-FP class itself. Using the object factory guarantees that the correct root-IP for the given root-FP is created, while it need not to hard-coded which IP class matches which FP class. Also, the environment object provides a new tool with parameters from the desktop, most notably the material the tool is to be used on (if specified by the user).

More handling is going on behind the scenes, in particular on the desktop and for loading and storing materials, before tools can handle them. However, this part of the overall application framework does not add much to the discussion of the Tools framework and is therefore omitted.

7.2.2 Classes and their functionality

The classes of the framework can be categorized as follows:

- *FPart hierarchy.* These are the FPart, SubFPart, and ContextFPart classes.

- *IPart hierarchy.* These are the IPart, SubIPart, and ContextIPart classes.
- *Tools environment.* These are the Environment, EventDispatcher, Description, and ObjectFactory classes.

The following first part describes the FPart class hierarchy.

- *FPart* is the abstract superclass for all FP objects. It defines the functionality common to all of them. Specific FP classes must inherit from it. FPart defines the following protocols:
  - *Event notification.* Clients of an FP object can register to be notified about events. A client registers for specific event types and provides the FP with operation names to call if the event occurs. By providing different operations for each event type rather than a dedicated callback operation, no common protocol is needed among observers that are interested in different event types.
  - *Request handling.* An FP provides operations to receive requests from its sub-FP objects. Requests are instances of a dedicated Request class.
  - *Sub-FP instantiation.* An FP declares operations to instantiate its sub-FP object. These are internal operations to be implemented by subclasses (inheritance protocol).
  - *FPart description.* An FP class provides a description of its properties. In Smalltalk, this is a class-level operation. The description provided is an instance of class Description. The description is used in the instantiation of objects through the object factory (see below).
  - *IPart instantiation.* An FP provides operations to instantiate its IP (there is exactly one IP object for an FP object in this Tools framework).
- *ContextFPart* is the abstract superclass of all root-FP classes. Every tool must define a subclass of ContextFPart whose instances represent the tool to the environment. ContextFPart inherits from FPart and adds the following protocols:
  - *Event dispatcher connection.* A root-FP provides operations to connect to the event dispatcher. It knows its dispatcher, forwards specific events to it, and receives events from it.
  - *Environment connection.* A root-FP provides operations to connect to its environment. A client may ask about the current tool status, for example whether it has been launched successfully.
  - *Material handling.* A root-FP provides operations to provide sub-FP objects with the current material.
- *SubFPart* is the abstract superclass of any FP class of objects from the FP hierarchy, except for the root-FP object, which must be an instance of a subclass of ContextFPart. SubFPart is a subclass of FPart and adds the following protocols:
  - *Super-FP handling.* It provides operations to get and set the super-FP.

The following second part describes the IPart class hierarchy.

- *IPart* is the abstract superclass of all IP objects. It defines the functionality common to all IP objects. Every IP class must inherit from it to extend the framework. IPart defines the following protocols:
  - *FPart handling.* An IP object provides operations to attach itself to an FP. It receives the FP object and registers with it for the event types it is interested in.
  - *Sub-IP instantiation.* An IP declares operations to instantiate its sub-IPs. These are internal operations to be implemented by subclasses (inheritance protocol).

  - *IPart description.* An IP class provides a description of its properties. In Smalltalk, this is a class-level operation. The description provided is an instance of class Description. It is used in the instantiation of IP objects through the object factory (see below).

- *ContextIPart* is the abstract superclass of all root-IP classes. Every new tool must define a subclass of ContextIPart. ContextIPart inherits from IPart and adds the following protocol:
  - *GUI handling.* A root-IP provides operations to retrieve an icon, which represents the tool on the desktop. It also provides operations to open and close the main window.
- SubIPart is the abstract superclass of any IP class of objects from the IP hierarchy, except for the root-IP object, which must be an instance of a subclass of ContextIPart. SubIPart is a subclass of IPart, to which it adds the following protocols:
  - *Super-IP handling*. It provides operations to get and set the super-IP.

Both SubFPart and SubIPart leave open the handling of further embedded sub-parts. This functionality must be defined and implemented by every sub-part anew, as described by the section on how to use the framework.

Finally, the following third part describes the environment classes.

- *Environment* is a concrete class, whose sole instance is the environment object. This object manages all available tools. The Environment class defines the following functionality:
  - It starts up new tools based on user input, initializes them, manages, them, and finally shuts them down.
  - It provides access to the event dispatcher and the IP and FP factories (as asked for by root-FP objects, see below).
- *EventDispatcher* is a concrete class, whose sole instance serves to inform dependent tools about state changes. It collects and distributes events it receives from root-FP objects, so that the tools can update themselves, if a material of relevance to them has changed. The EventDispatcher class provides the following functionality:
  - It provides operations for a root-FP object to register and unregister interest in specific event types.
  - It dispatches events provided by a root-FP to all FP objects that have registered interest into the event type.
- *ObjectFactory* is a concrete class whose instances serve to create objects without naming the classes of the objects. Thus, a class is not named directly, but identified by an instance of class Description. Such a description might be a simple string, for example a tool name. The ObjectFactory class provides the following functionality:
  - It provides an initialization protocol that lets clients define the root class of the hierarchy from which objects can instantiated.
  - It provides a protocol that lets clients create instances of classes defined by a description object, and lets them retrieve the full set of classes that match the description.

  There may be any number of object factories at runtime. Two dedicated object factories, both of which are provided by the environment object, are the IP and FP factories. They are used to instantiate the root objects of both the IP and FP object hierarchies of a tool.
- *Description* is the superclass of object descriptions that can be used by object factories. A description object identifies a set of equivalent classes (typically, there is only one element in the set, which means that the description unambiguously identifies a specific class). The Description class provides the following protocols:

- It provides an operation to check two description objects for equality and an operation to provide a key object for use in a dictionary.
- It provides operations to match description objects with each other.

Each subclass provides initialization protocols specific to the class hierarchy the description objects are to be used for.

The discussion omits the classes Material and MaterialManager, because they do not add much to the discussion.

7.2.3 How to use the framework

The Tools framework is a white-box framework. Defining concrete subclasses of SubIPart, ContextIPart, SubFPart, and ContextFPart creates new types of tools. IPart and FPart are usually not subclassed directly.

- *ContextFPart* is the superclass of all root-FP classes. Whenever a new tool is developed, a new subclass of it must be created. The following inheritance protocols have to be implemented by every subclass:
  - A protocol to instantiate new sub-FP objects.
  - A protocol to conveniently access specific sub-FP objects.
  - A class-level protocol that provides metadata like the tool name or the default material class.

  In addition, for each new root-FP that reuses sub-FP objects (and every non-trivial root-FP does so), management operations for handling these sub-FP objects must be written. Typically, this includes operations to add and remove sub-FP objects from a sub-FP collection.

A new tool might solely be built by reusing existing IP and FP classes. More typically, though, new sub-FP classes need to be introduced. Such a new sub-FP must be a subclass of SubFPart.

- *SubFPart* is the superclass of all FP classes, which are not root-FP classes. This includes all classes whose instances play middle-tier and leaf node roles in the FP object hierarchy. When defining a new sub-FP class, no inheritance protocol needs to be implemented.

  However, if the new sub-FP class is not a leaf class, but rather a middle-tier node class, it must provide functionality to manage its sub-FP objects. Typically, this includes operations to add and remove sub-FP objects from a sub-FP collection. This functionality is redundant with the one provided by a new ContextFPart subclass (see discussion above on how to extend ContextFPart).

On the interaction side of a tool, a new subclass of ContextIPart needs to be created for each new subclass of ContextFPart.

- *ContextIPart* is the superclass of all root-IP classes. The following inheritance protocol needs to be implemented by every subclass:
  - A protocol of how to react to user interface events like closing the window.

  In addition, each new root-IP needs to manage its sub-IP objects, so it defines operations to handle its sub-IP objects. Typically, this includes operations to add and remove sub-IP objects from a sub-IP collection.

For each sub-FP class, there needs to be at least on sub-IP class. Such a class must be a subclass of SubIPart. When defining a new sub-IP class, no inheritance protocols need to be implemented.

For sub-IP objects, which may contain further sub-IP objects, a management protocol of these sub-IP objects needs to be defined and implemented. This mirrors the situation of the IP/sub-IP relationship.

This protocol and its implementation are also redundant with the one of the subclasses of ContextIPart.

7.3 Problems with the original framework

Discussions with users and the developers of the framework lead to the recognition of the following problems with understanding and using the framework.

On a general level, the problems that form the motivation for this dissertation were present:

- *Class interfaces are complex and hard to understand.* Users wished they could get into the framework faster and with less overhead and pain. The developers wished they could reduce their coaching efforts.
- *Object collaboration was not well understood.* Different purposes of object collaborations had been recognized, but only sparsely separated, and tools for making object collaboration tasks explicit to help communicate them were missing.
- *Too tight coupling between tools and environment.* The fixed coupling between tools and environment classes hid how to use the framework. While less relevant for users, developers wished they could have a better separation of concerns.
- *Too many simple repetitive tasks to be carried out by hand.* Developers had to implement lots of simple and frequently redundant functionality. Most of it could be automated or delegated to the GUI builder (that had only been put to limited use).

In general, the developers wished they could communicate faster and better how the framework worked and how users were to use it.

These general problems were complemented by several minor observations on using the framework.

- Creating new subclasses required implementing too many abstract operations. The inheritance protocols were too broad and put too much of a burden on the users of the framework.
- There was too much code redundancy between new subclasses of Sub- and ContextFPart as well as Sub- and ContextIPart. Much of the management of sub-FP and sub-IP objects was redundant.
- A general feeling was that the class hierarchy was not as good as it should be. A prime indicator of this is the aforementioned implementation redundancy.

To overcome these problems, clean up the design, and make the framework more effective, the KMU Desktop project management decided to redesign the Tools framework. The next sections describe the redesigned framework. The final section describes the redesign team's observations from the process.

7.4 The redesigned Tools framework

This subsection describes the new Tools framework after the redesign took place. It uses the framework documentation template from Chapter 4 to document the framework using role modeling.

7.4.1 Framework overview

The redesigned Tools framework is a white-box framework that is used to construct tools, just like the original framework. It extends the KMU Desktop Object framework (not discussed here). In contrast to the original framework, it has a different class hierarchy, and some functionality, most notably functionality provided by or close to the Environment class, is moved out into a new framework, the Environment framework. The Environment framework is described in the next subsection.

Tools still serve the same purposes as in the original Tools framework: users use them to work on their materials, which are the objects from the underlying domain model. Also, the overall software tool is to be understood as a hierarchy of logical tool components, each of which is represented by one FP object. For each FP object, there may be one or more IP objects. Taken together, the FP object and its IP objects form a tool component.

Generally speaking, the overall runtime architecture of software tools remains the same, but the underlying object-oriented framework changed to make it more easily usable.

Figure 7-3 shows the class model structure of the redesigned Tools framework.

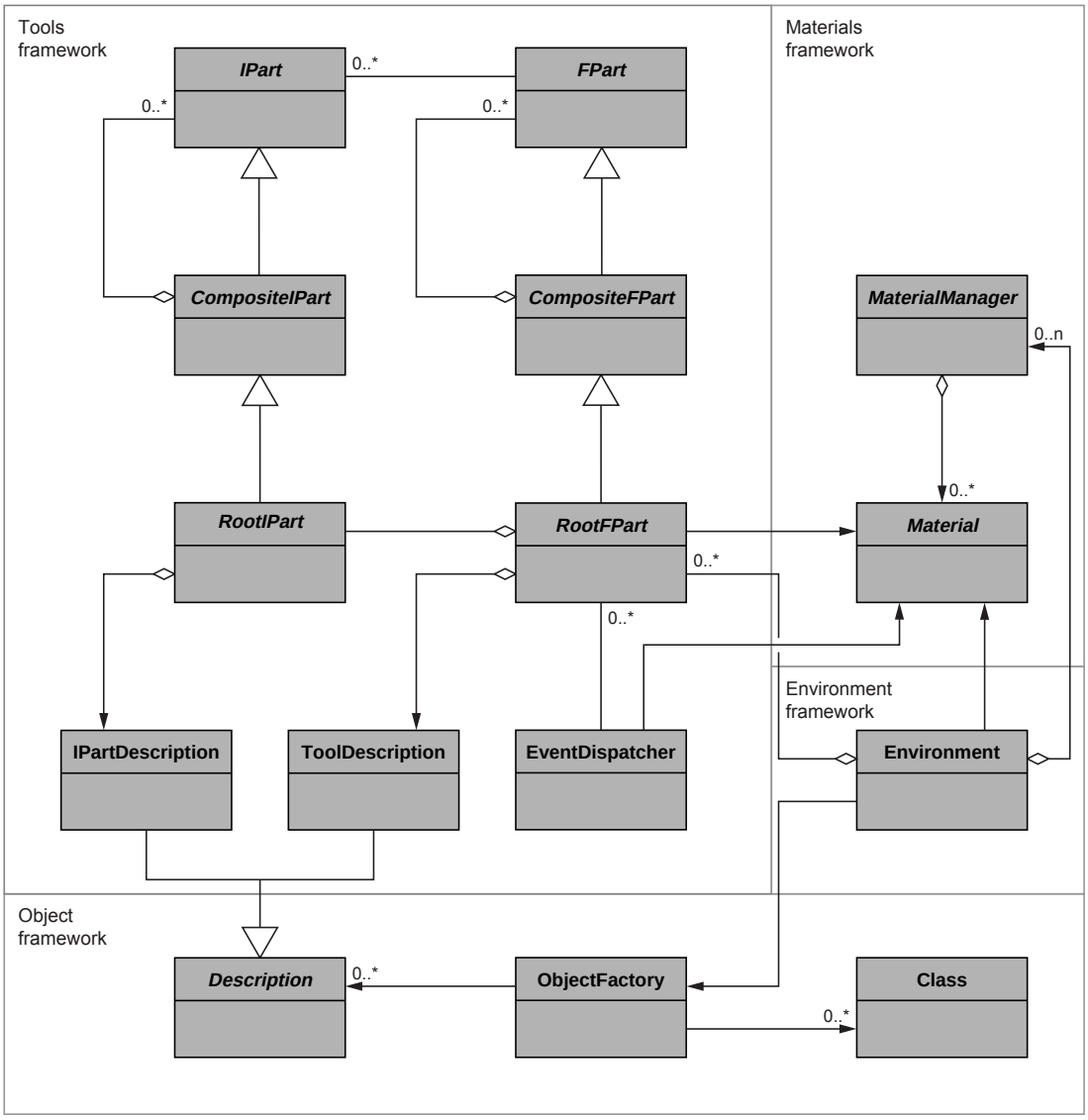

Figure 7-3: Class model structure of the redesigned Tools framework.

The framework consists of two primary class hierarchies, the IPart and the FPart class hierarchy, as well as additional Description classes and the EventDispatcher class.

An IP object is an instance of (a subclass of) IPart, and an FP object is an instance of (a subclass of) FPart.

The redesigned framework now employs the class-based version of the Composite design pattern [GHJV95], so there are subclasses CompositeIPart and CompositeFPart of classes IPart and FPart, respectively. This is the only major structural change to the framework. Effectively, it is a refactoring of functionality among the classes rather than the introduction of new functionality.

Any leaf IP or FP object must be an instance of a subclass of IPart or FPart, respectively, but not of CompositeIPart or CompositeFPart. Any IP or FP object that may contain sub-IPs or sub-FPs must be an instance of a subclass of CompositeIPart or CompositeFPart. If, in addition, an IP or FP object serves as the root of the IP or FP object hierarchy, it must be an instance of a subclass of RootIPart or RootFPart.

The RootIPart and RootFPart classes provide clients with instances of IPartDescription and ToolDescription, respectively. These description classes identify a specific subclass of RootIPart or RootFPart. They are subclasses of a more generalized Description class, which stems from the Object framework. A ToolDescription identifies the tool by its root-FP class for a given tool name, and an IPartDescription identifies the root-IP class for a given root-FP class.

7.4.2 Class model

The Tools framework comprises the IPart, CompositeIPart, and RootIPart classes, the FPart, CompositeFPart, and RootFPart classes, the IPartDescription and the ToolDescription classes, and the EventDispatcher class.

The FPart class hierarchy defines the abstract classes for building the FP objects of a software tool instance. It is structured according to the class-based version of the Composite pattern. Leaf-FP classes must inherit from FPart (but not from CompositeFPart), Composite-FP classes that are not root-FP classes must inherit from CompositeFPart, and Root-FP classes must inherit from RootFPart.

- *FPart* is the abstract superclass of all FP objects in a tool. It provides role types that define what clients can do with *any* kind of FP object. For example, FP objects collaborate with their super-FP in the FP object hierarchy and they collaborate with their IP objects.
- *CompositeFPart* is the abstract superclass of all FP objects in a tool that may have sub-FP objects. It is a subclass of FPart. In addition to the role types inherited from FPart, it provides role types of role models that create, manage and collaborate with sub-FP objects.
- *RootFPart* is the abstract superclass of all root-FP objects. In addition to the role types inherited from CompositeFPart, it provides role types that define how a root-FP object collaborates with its environment.

The IPart hierarchy provides the abstract classes for IP objects of a tool instance. It is structured ismorphically to the FPart hierarchy, employing the Composite pattern again. Leaf-IP classes must inherit from IPart, composite-IP classes must inherit from CompositeIPart, and root-IP classes must inherit from RootIPart.

- *IPart* is the abstract superclass of all IP classes. It provides role types that define what clients can do with *any* kind IP object. This includes the collaboration with its super-IP, as well as the collaboration with its FP object.

- *CompositeIPart* is the abstract superclass of all IP objects that may have sub-IP objects. It is a subclass of IPart, to the role type set of which it adds role types for creating, managing, and collaborating with sub-IP objects in the hierarchy.
- *RootIPart* is the abstract superclass of all root-IP classes. It is a subclass of CompositeIPart, to the role type set of which it adds role types for collaborating with its FP object.

Software tools are instantiated by creating the root-FP object, which then builds the rest of the tool. The root-FP object may either be instantiated directly by naming its class, or it may be instantiated lazily by using a name reserved for the tool (for example, “Customer Browser” or “Risk Assessment Tool”). The lazy instantiation process uses the Product Trader pattern. This pattern is used twice, for the root-FP and for the root-IP. The specifications for these classes are subclasses of Description, which is a class inherited from the Object framework.

- *ToolDescription* is a concrete subclass of Description. Every concrete root-FP class provides an instance of ToolDescription that offers a tool-specific key object for use in a dictionary. The key is calculated from a string that represents the tool name. In this context, the key used to identify an FP class in the Object Factory.
- *IPartDescription* is a concrete subclass of Description. Every concrete root-IP class provides an instance of IPartDescription that offers a root-IP specific key object for use in a dictionary. The key is an identifier for the FP class the root-IP class can work with (the IP class must match the FP class). IP root classes are registered under this key in the object factory.

Finally, software tools coordinate each other using a central event dispatcher.

- *EventDispatcher* is a concrete class that is instantiated as the event dispatcher singleton. Tools use it to communicate state changes to other tools. The communication is based on a fixed set of generally known event types, for which tools may register interest, and about whose concrete occurrences they are notified.

The root-FP object of each tool can access the event dispatcher at its central location. Each root-FP registers its interest in particular other tools or their materials, and provides the dispatcher with event notifications about changes to its own state or its materials.

7.4.3 Free role models

The role models of the framework can be classified into free externally visible role models, and hidden internal role models. This first part focuses on the free role models, as they are available to black-box use clients.

The free role models fall into two categories.

- *Managing a tool through its FP objects.* These are role models that describe how root-FP objects communicate with their environment.
- *Creating a tool using the Product Trader pattern.* These role models describe how description objects are created to instantiate the root-IP and FP object of a tool without naming their classes.

Figure 7-4 shows the class model including all free and all internal role models.

The following role models deal with managing a tool through its FP objects.

- The *FPart* role model serves to provide functionality that is available from any FP object for any Client. FPart provides the FPart role type, and Client is a free role type. The FPart role type provides all needed information about the FP object, for example its name (for resource management).

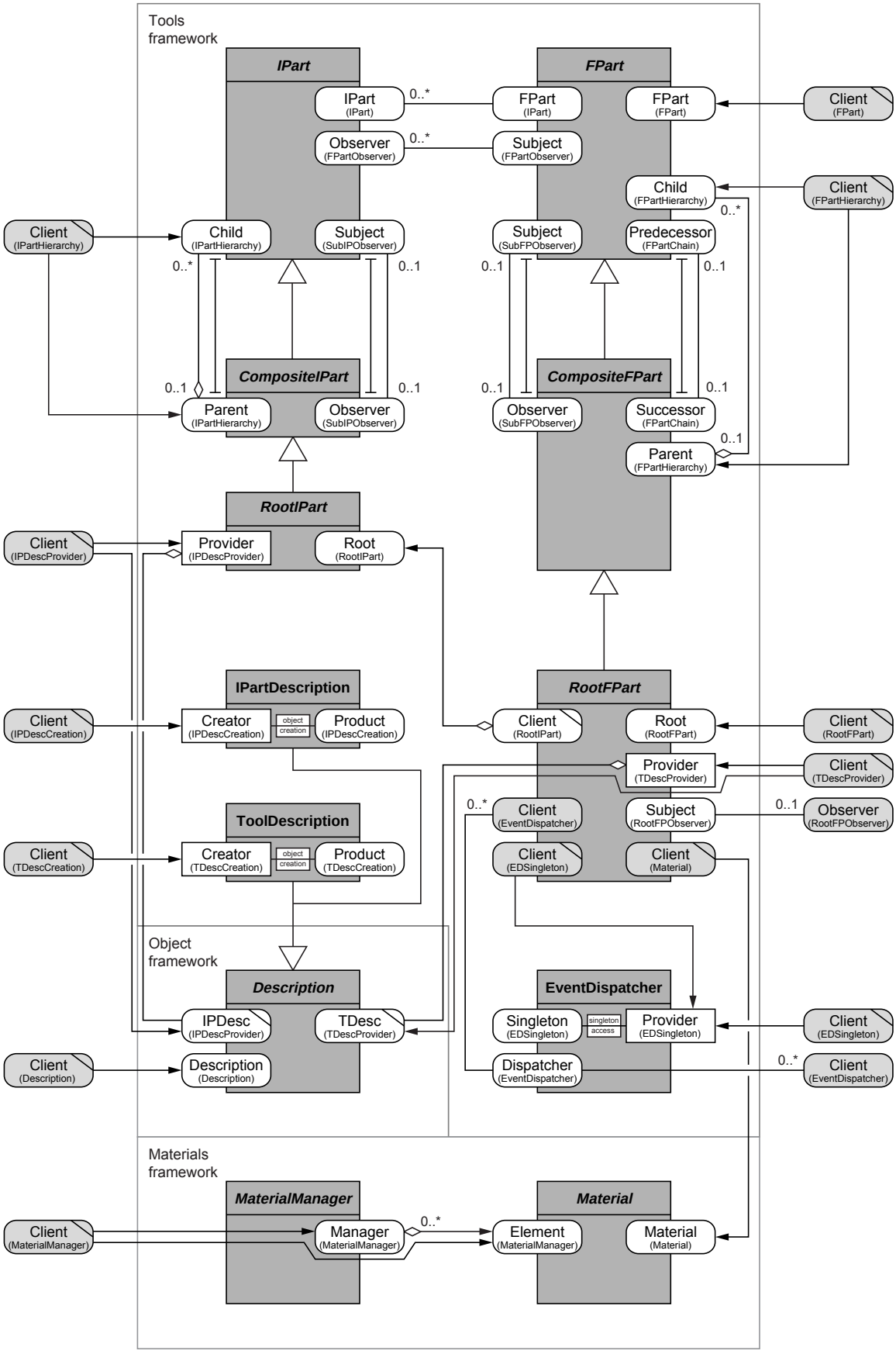

Figure 7-4: Class model of the redesigned Tools framework, including all role models.

- The *RootFPart* role model serves to provide functionality to clients, which is available from any root-FP object. A Client thereby handles the RootFPart. RootFPart provides the RootFPart role type, and Client is a free role type.

  The RootFPart role type lets clients get and set the main material of the tool, request status information about the tool, and start it up or shut it down.

- The *RootFPObserver* role model serves to let exactly one framework-external object monitor a root-FP object for state changes. It is an instance of the Observer pattern. RootFPart provides the Subject role, and Observer is a free role type.

  The Subject role type provides operations to register and unregister the Observer object. The Observer object provides operations to receive the state change notifications.

The following role models deal with instantiating a tool. The Description role model is defined in the Object framework. It is described here to help understand the application of the Product Trader pattern.

- The *Description* role model serves to match Description objects with each other and to maintain them in a dictionary. Each Description object identifies a specific object (typically a class). The role model is an instantiation of the Specification patterns from [Rie96c, EF97]. Description provides the Description role type, and Client is a free role type.

  The Description role type provides operations that return a key that identifies the Description object. Two keys received from two different Description objects are equal, if they represent the same class. In addition, the Description role type provides operations to match two different Description objects and check for substitutability.

  Subclasses of Description implement how to compute the key, check for equality, and match two Description objects. Example subclasses are ToolDescription and IPartDescription.

- The *TDescProvider* role model (ToolDescriptionProvider) makes a root-FP class object provide a Description object that unambiguously identifies the class. RootFPart provides the Provider role type, Description provides the Description role type, and Client is a free role type.

  Any concrete subclass of RootFPart instantiates exactly one ToolDescription object. The root-FP class object returns this Description object when asked for it through the Provider role type. For the creation of the ToolDescription object, the concrete RootFPart subclass uses the TDescCreation role model.

- The *TDescCreation* role model (ToolDescriptionCreation) serves to instantiate a ToolDescription object with sufficient parameters to identify the object the Description object represents. It is shown in the figure using the object creation shorthand. ToolDescription provides the class-level Creator role type and the instance-level Product role type, and Client is a free role type.

  The Creator role type offers instantiation operations that take a string. The string represents the tool name.

  This role model is used by the environment of the Tools framework to instantiate a tool using its name only (rather than a specific class name).

- The *IPDescProvider* role model (IPartDescriptionProvider) makes a root-IP class object provide a Description object that unambiguously identifies the class. RootIPart provides the Provider role type, Description provides the Description role type, and Client is a free role type.

  Any concrete subclass of RootIPart instantiates exactly one IPartDescription object. The root-IP class object returns this Description object when asked for it through the Provider role type. For the creation of the IPartDescription object, the concrete RootIPart subclass uses the IPDescCreation role model.

- The *IPDescCreation* role model (IPartDescriptionCreation) serves to instantiate an IPartDescription object with sufficient parameters to identify the object the Description object represents. It is shown in the figure using the object creation shorthand. IPartDescription provides the class-level Creator role type and the instance-level Product role type, and Client is a free role type.

  The Creator role type offers instantiation operations that take the root-FP class the root-IP class has been designed to work with.

  This role model is used by the environment of the Tools framework to instantiate the root-IP object for a new tool.

These role models (Description, ToolDescription, TDescProvider, IPartDescription, IPartProvider) and these classes (Description, ToolDescription, IPartDescription) taken together with the ObjectFactory introduced as part of the Environment framework, form the instantiation of the Product Trader pattern.

7.4.4 Internal role models

The internal role models structure the communication of framework objects among each other. They fall into four main categories:

- *IPart with FPart communication.* These role models describe how the IP objects of a tool collaborate with their respective FP object.
- *Managing the FP object hierarchy.* These role models describe how the FP object hierarchy is built and how the FP objects collaborate with each other.
- *Managing the IP object hierarchy.* These role models describe how the IP object hierarchy is built and how the IP objects collaborate with each other and with FP objects.
- *Maintaining state dependencies between tools.* These role models define how root-FP objects (each representing a specific tool instance) communicate to maintain their state dependencies.

The following first category of role models describes how IP objects collaborate with FP objects.

- The *FPart* role model serves to provide functionality that is available from any FP object for any Client. FPart provides the FPart role type, and Client is a free role type. The FPart role type provides all needed information about the FP object, for example its name (for resource management).
- The *IPart* role model serves to let an FP object manage its IP objects. IP objects can add or remove themselves from an FP, and the FP can use the standard functionality of an IP, like showing or hiding it. IPart provides the IPart role type, and FPart provides the FPart role type.
- The *FPartObserver* role model serves to make an FP object notify its IP object about state changes relevant for the handling and user-interface of the tool component. It is an instance of the Observer pattern. FPart provides the Subject role type, and IPart provides the Observer role type.
- The *RootIPart* role model serves to let a root-FP object handle the IP object hierarchy as a whole. It provides operations to startup, show, hide, and shutdown the overall user interface. RootIPart provides the Root role type, and RootFPart provides the Client role type.

The following second category of role models describes how FP objects collaborate with each other to build and maintain the FP object hierarchy of a tool.

- The *FPartHierarchy* role model is used to build and change the FP object hierarchy. It is an instance of the Composite pattern. FPart provides the Child role type, CompositeFPart provides the

Parent role type, and Client is a free role type. Typically, the Client role type is picked up by subclasses of CompositeFPart from a framework extension.

- The *SubFPObserver* role model serves to make a sub-FP object notify its parent-FP object about state changes relevant for the tool functionality and material state. It is an instance of the Observer pattern. FPart provides the Subject role type, and CompositeFPart provides the Observer role type.
- The *FPartChain* role model serves send request up the FP object hierarchy until an FP object knows how to handle it. It is an instance of the Chain of Responsibility pattern. FPart provides the Predecessor role type, and CompositeFPart provides the Successor role type.

  The Successor role type provides a generic operation for receiving Request objects, and a few operations common to all tools that directly reflect a specific request type, for example QuitRequest.

The following third category of role models describes how IP objects collaborate with each other to build and maintain the IP object hierarchy of a tool, and how they communicate with FP objects.

- The *IPartHierarchy* role model is used to build and change the IP object hierarchy. It is an instance of the Composite pattern. IPart provides the Child role type, CompositeIPart provides the Parent role type, and Client is a free role type. Typically, the Client role type is picked up by subclasses of CompositeIPart from a framework extension.
- The *SubIPObserver* role model serves to make a sub-IP object notify its parent-IP object about state changes relevant for the display of the user interface. It is an instance of the Observer pattern. IPart provides the Subject role type, and CompositeIPart provides the Observer role type.

Finally, the following fourth category of role models describes how the root-FP objects communicate to maintain their state dependencies.

- The *EventDispatcher* role model serves to inform tools about state changes of other tools or materials they depend on. It is a variant of the Observer pattern. EventDispatcher provides the Dispatcher role type, and RootFPart provides the Client role type.

  A Client may act both as a source and as a target of event notifications. First, a Client registers itself at the Dispatcher, providing its type (tool type) and name (tool instance). Then, it registers those materials at the Dispatcher, for which it holds ownership. It thereby promises to inform the Dispatcher about state changes of these materials. Finally, it registers his interest into state changes of other tools or materials using names or object references to identify them.

  At runtime, the Client informs the Dispatcher about relevant state changes. The Dispatcher dispatches these event notifications to other Clients that had registered interest in these state changes.

  Therefore, the Dispatcher role type provides operations for Clients to register their interest in various types of events, as well as operations to receive and dispatch event notifications. The Client role type in turn provides operations for the Dispatcher to call back on it, that is to receive the event notifications.
- The *EDSingleton* role model serves to provide a convenient access point to the system-wide EventDispatcher singleton. It is shown in the figure using the Singleton shorthand. EventDispatcher provides the class-level Provider role type and the instance-level Singleton role type, and RootFPart provides the Client role type.

7.4.4.1 Built-on classes

The Tools framework builds on several other frameworks. The two most important ones are the Widget framework (CommonWidgets and VisualAge Parts, in this case), and the Materials framework.

- For building the user interface, an IP object arranges a set of widgets in a hierarchical fashion. How this is done, depends on the concrete IP class. However, the IPart class itself holds a reference to a distinguished root Widget for the part of the user interface the IP is responsible for.
- For handling material objects, a root-FP object maintains a reference to the main material object the tool is working on. A material manager object maintains such material objects. Figure 7-4 illustrates these relationships (but does not detail them).

The use of these built-on classes takes place with the appropriate role models. It would be tedious and tiresome to add them to the discussion, so they are omitted.

7.5 The new Environment framework

The old Tools framework has been split up into the redesigned Tools framework and the new Environment framework to better separate design and implementation artifacts and ease application packaging. This section describes how parts of the new Environment framework use the Tools framework.

The Environment framework integrates the Tools framework into the larger context of a (desktop) application. The Environment framework provides the root objects of a system. These root objects control the startup and shutdown of the application process, instantiate tools, and connect tools with the material management and individual materials. The focus of this discussion is on the collaboration of the Environment framework with the Tools framework only. All issues not relating to this collaboration are omitted.

The focus of the Environment framework is the Environment class, which is the class that provides the root object the system is started up by. The Environment object creates, manages, and deletes all tools. Tools are represented by their root objects, which are instances of concrete subclasses of RootFPart. Currently, tools are single-threaded, which allows for the simplified use of singleton objects.

The environment object uses a system-wide object factory to instantiate a tool. The object factory is the sole instance of ObjectFactory. From a client, the factory receives a specification for an object, determines a class that matches the specification, and creates an instance of the class, which it returns. A class specification is always an instance of a concrete subclass of Description like IPartDescription or ToolDescription.

Description and ObjectFactory are classes from the Object framework. Their use allows application developers to configure a process with new tools through configuration data and dynamic class loading, without having to change and recompile the system.

Figure 7-5 shows the design of Environment framework and its use of the Tools framework.

Of the 7 role types provided by the Environment class, 6 are from the Tools framework, where they are discussed (see Section 7.3). These are the RootFPart, RootFPartObserver, IPDescCreation, TDescCreation, EventDispatcher, and EDSingleton role models.

In addition, the Environment class provides the free client role type of the Object Factory role model.

- The *Object Factory* role model serves to let a client abstractly create an object that conforms to a given specification. It is an instance of the Product Trader pattern [BR98]. ObjectFactory provides the Factory role type, and Environment provides the Client role type.

  The Factory role type provides operations to request a new object that conforms to given specification. The specification must be an instance of a concrete subclass of Description.

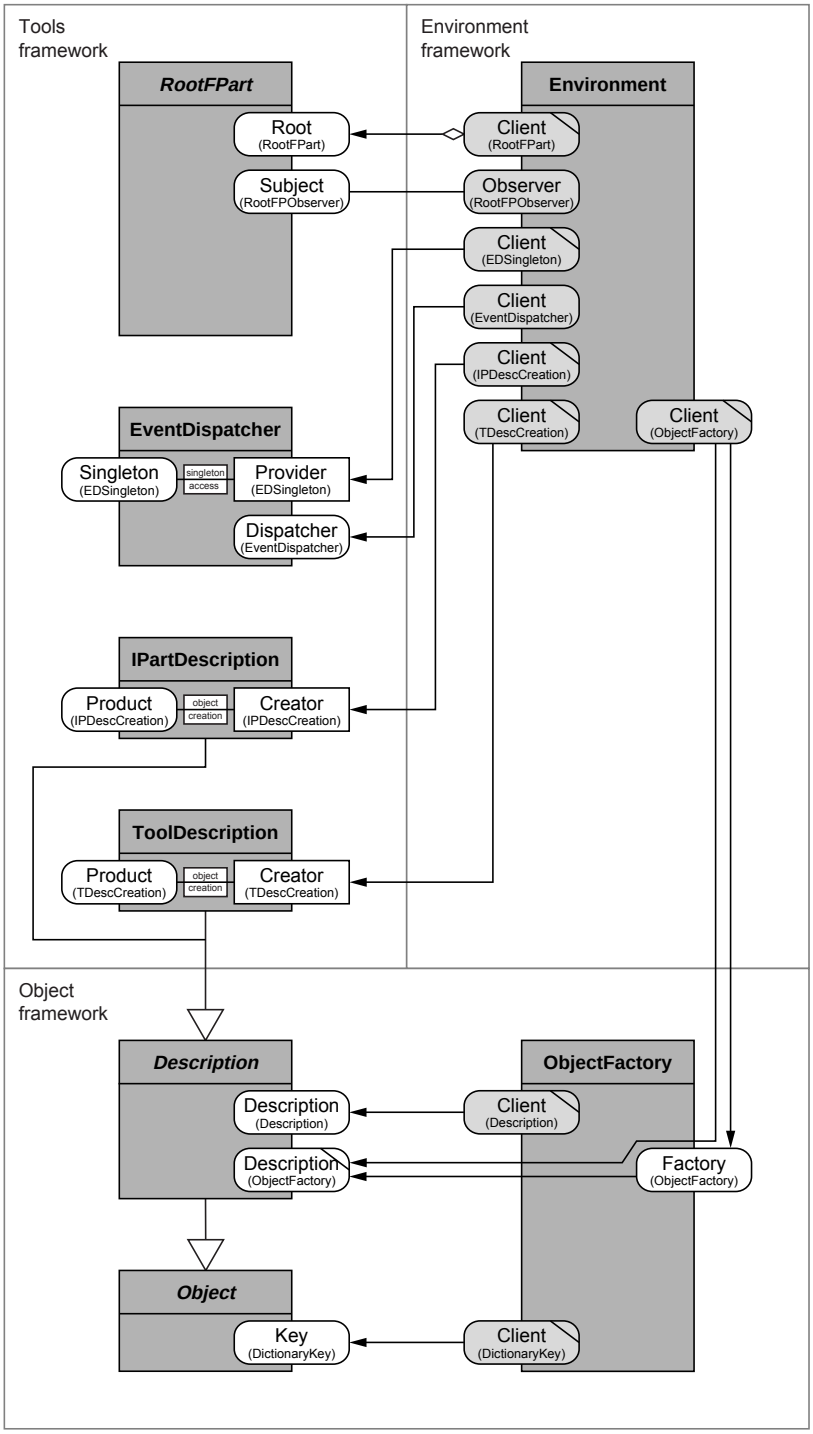

Figure 7-5: The integration of the Tools framework into an environment.

For its implementation, the Object Factory uses further role models from the Object framework.

- The *Description* role model (from the Object framework) serves to match specifications with each other. It is an instance of the Specification pattern [Rie96c, EF97] and part of an instance of the Product Trader pattern. Description provides the Description role type, and ObjectFactory provides the Client role type.
- The *DictionaryKey* role model (from the Object framework) serves to make an object provide keys (hashcodes) of itself for use in a dictionary (hashtable). It is an instance of a common (un-

named) pattern. Object provides the Key role type, and ObjectFactory provides the Client role type.

There are many more role models, but they do not add much to showing how the Environment framework is using the Tools framework. Thus, they are omitted from this discussion.

7.6 Experiences and evaluation

During the redesign process the redesign team made use of role modeling and the role–model–based design pattern catalog. The following subsections present the experiences of team members in a structured way based on the problems motivating this dissertation. I received these experiences and observations shortly after the redesign process had been finished and after the new framework had been released to users.

Of primary interest is the quality of the redesigned framework and whether it has overcome the problems stated earlier in this chapter. This is the case. However, while the redesign team used role modeling for its work, the framework users did not. They only received traditional documentation. The primary reason for this is that role modeling as presented in this dissertation requires framework users and developers to have substantial experience in object-orientation (see thesis statement in Chapter 2).

In large projects like the KMU Desktop project, the developer population is diverse, and not everyone can be expected to be an expert of object-orientation. There are many technical and organizational solutions to the problems strong variations in developer expertise cause for framework-based development. Some of those applied in the KMU Desktop project are listed in Section 7.6.6, which also shortly discusses how the framework kept evolving after the initial redesign.

7.6.1 Statistics of case study

The KMU Desktop Tools framework provides us with the data shown in Table 7-1.

Number of classes	9
Number of role models	17
Number of pattern instances	10
Number of role types assigned to classes	31
Ratio of role types per class	3.44
Standard deviation of role types per class	1.77
Ratio of role models per class	1.89
Ratio of pattern instances per role model	0.59

Table 7-1: Raw data and computed figures from the Tools framework.

The KMU Desktop Object framework provides us with the data shown in Table 7-2.

Number of classes	3
Number of role models	3
Number of pattern instances	2
Number of role types assigned to classes	6
Ratio of role types per class	2.0
Standard deviation of role types per class	0.82
Ratio of role models per class	1.0
Ratio of pattern instances per role model	0.67

Table 7-2: Raw data and computed figures from the Object framework.

The KMU Desktop Environment framework provides us with the data shown in Table 7-3.

Number of classes	1
Number of role models	0
Number of pattern instances	0
Number of role types assigned to classes	7
Ratio of role types per class	7.0
Standard deviation of role types per class	0
Ratio of role models per class	0
Ratio of pattern instances per role model	N/A.

Table 7-3: Raw data and computed figures from the Environment framework.

The discussion of the frameworks only shows their key interface architecture and omits classes of lesser importance and helper classes. Also, it does not show any extension. Finally, the Environment framework is shown only in so far as it relates to the Tools framework.

7.6.2 Complexity of classes

Regarding the complexity of classes, the redesign team made the following observations:

- *Designing classes.* Team members found that the focus on role types and role models made designing the new classes easier than the original process (as it was remembered).
- *Learning classes.* Framework users and original developers said that they could more easily understand the new classes than they were able to understand the original ones.

- *Using classes.* Also, the users of the new framework said that the redesigned framework was easier to use than the original framework.

The use of Smalltalk method categories helped learning and using classes. The method categories map directly on role types.

These observations apply to all complex classes. For example, the RootFPart or Environment classes became easier to define and work with once they were viewed from the point of view of role types.

7.6.3 Complexity of object collaboration

Regarding the complexity of object collaboration, the redesign team made the following observations:

- *Designing object collaborations.* Team members continuously switched between a class model structure and an object collaboration view, the later of which was based on role models.
- *Learning object collaboration.* Users and original developers said that they could more easily understand how objects collaborated to achieve the overall purpose of the framework.
- *Using the framework.* Users of the new framework said that the redesigned framework was easier to use than the original framework, in particular with respect to how the objects interacted.

Smalltalk method categories also supported thinking in terms of collaborations, because the roles of objects could be distinguished from each other, and method categories of one class had counterparts at those classes whose instances were part of the different object collaboration tasks.

Again, these observations apply to all complex collaborations between objects. For example, the collaboration between an IP and an FP object, and the collaboration between a root-FP object and the environment object became much easier to define and handle once the different collaboration tasks involved were understood and modeled using role models.

7.6.4 Clarity of requirements put upon use-clients

Regarding the problems of clear expectations on use-clients of the framework, no specific observations were made. This is primarily due to the fixed embedding of the Tools framework into the Environment framework. The lack of further clients prevents the repeated use of free role types, and therefore does not suggest anything with respect to their usefulness in defining requirements on use clients.

7.6.5 Reuse of experience through design patterns

Finally, the redesign team observed that both the original and the new framework exhibit a high density of pattern applications. Hence, it achieved a high degree of reuse of experience.

The Tools framework uses the following patterns: Composite (both role–model–based and class-based version), Observer (repeatedly), Chain of Responsibility, Product Trader, and Specification. It uses further undocumented patterns (keys for a dictionary, Class Object). Also, the documentation does not show several pattern instances, in particular those close to the code.

In the new framework, the team was able to view role models as pattern instances that we had not recognized as such before.

For example, the team's analysis of the collaboration between a super-FP and a sub-FP lead to the introduction of the FPartObserver and FPartChain role models, while only one (implicit) role model, an instance of the Observer pattern, existed in the original framework. The team made this design deci-

sion, because the focus on object collaboration tasks made it recognize the two different purposes the original Observer pattern application was being used for.

Other examples are the same distinction on the IPart class hierarchy side, and the separation of the different collaboration tasks in the RootFPart interface.

Perhaps the most important example of how role modeling eased reusing design experience is the application of the Composite pattern. The original design repeatedly introduced code in subclasses that reflected the Composite pattern. Once the team recognized this, it decided to change the class model structure accordingly to better reflect the class-based version of the Composite pattern. Disassembling the existing class model into its role model pieces, refactoring the class model to match the class-based version of the pattern, and recomposing it was significantly facilitated through the use of role models.

The use of role modeling made the recognition and application of design experience in the form of design patterns easier than would have been possible with a traditional class-based approach.

7.6.6 Further evolution of framework

Since the redesign and its subsequent implementation in March 1998 (and the feedback I got at that time) the framework has kept evolving. As of April 1999, the following changes have been applied to the framework.

- *Simplification of FPart class hierarchy.* The classes FPart and CompositeFPart have been merged to form one FPart class. However, the functionality of the classes has remained the same. In particular, this one FPart class still represents an instance of the Composite pattern. (See the Composite pattern example in Section 3.3.11 on design patterns in role modeling).
- *Simplification of IPart class hierarchy.* Similarly, IPart and CompositeIPart have been merged to form one IPart class. In addition, a generic RootIPart class has been introduced that need not be subclassed anymore.
- *Simplification of ObjectFactory.* The Description classes are gone and specifications for classes like FPart or IPart are handled generically as objects. Strings and class objects now directly serve as specifications.
- *Introduction of support tool.* A software tool now supports the creation of new tools, including both IP and FP classes. The tool frees users from implementing behavior that cannot be captured as part of the framework but that is repetitive and redundant otherwise.

For the case study, it is important to note that the framework functionality did not change much over the course of its evolution. The original framework was already close to the redesigned framework in terms of functionality, and the current version is even closer to the redesigned framework.

What did change is the class structure and the distribution of responsibilities among classes. The framework evolved towards less but more complex classes whose interfaces are highly structured through method categories. Most of the method categories represent a specific role type.

The evolution process (disassembling class functionality into pieces and recomposing them to form a new class structure) represents further evidence that role types and role models are better at capturing functionality than single classes.

8

Case Study: The JHotDraw Framework

This chapter presents the third and last case study of this dissertation: the JHotDraw framework for building graphical drawing editor applications. JHotDraw has several framework predecessors, most notably the Smalltalk framework of same name (HotDraw) and ET++. Also, an expert developer (Erich Gamma), who uses it for teaching purposes, developed JHotDraw. As a consequence, JHotDraw is a very mature framework. This chapter presents the framework starting out with a class-based design as gathered from the documentation. It then adds a role modeling interpretation to the existing design and thereby shows that role modeling adds crucial information that would otherwise be missed. The catalog of role model patterns was used in this work. Finally, this chapter consolidates its observations and relates them to the problems driving this dissertation.

8.1 Case study overview

JHotDraw is an application framework for building graphical drawing editor applications. It is a mature Java framework that is publicly available. The case study presented in this chapter discusses the core classes of the framework and shows how role modeling helps in its documentation.

8.1.1 JHotDraw history

Users use drawing editors to visually arrange graphical figure objects on a drawing area. Drawing editors are a common type of application, found on nearly every computer desktop. However, the type of figures may significantly vary. Some drawing editors are more like painting applications, allowing users to draw paintings. Other drawing editors cover a specific domain, so that the figures users ma-

nipulate reflect the semantics of the domain. Examples of the later are drawing editors for technical drawings in specific application domains like architecture or manufacturing.

JHotDraw is an application framework that can be used for developing custom-made drawing editor applications. Each application is targeted at a specific domain and reflects the domain's semantics by providing specific figure types and by observing their relationships and constraints.

JHotDraw was developed by Erich Gamma. The current version, on which this case study is based, is 5.1. It is a mature framework, and it is publicly available (see Appendix E for a pointer). This makes it an ideal candidate for a case study in a dissertation.

JHotDraw itself is based on a long history of drawing editor frameworks. In particular, JHotDraw is the Java version of an earlier Smalltalk framework, called HotDraw [Joh92]. Hotdraw is also publicly available (again, see Appendix E for a pointer). In addition, JHotDraw draws on its developer's background with ET++, an early C++ application framework [WGM89, WG95].

Erich Gamma uses JHotDraw for teaching purposes (which is one reason, why it is a well-designed and implemented framework). The code is annotated (using JavaDoc-style comments). Also, a tutorial exists, which discusses the major design issues of the framework [BG98].

In the following, we refer to this set of information (source code, code annotations and comments, and the tutorial) as the JHotDraw documentation. It is not a complete documentation (it is not meant to be complete). This does not represent a problem, because the case study does not attempt to provide a complete documentation of JHotDraw either.

8.1.2 The case study

The case study walks through the major design aspects of the JHotDraw framework. It categorizes them into three parts: a first part on the Figure class hierarchy, a second part on the Drawing and DrawingView classes, and a third part on the DrawingEditor classes.

Each part is presented in two forms. First, the design is described using the information available from the existing documentation. Each part starts with the class documentation from JavaDoc and relates it to the design documentation from the tutorial. This gives a fairly complete picture of the design under discussion. To make things complete, each part adds information from the source code that is missing from the documentation.

A second part then uses role modeling to describe the role–model–based class model of the design. This revised class model is the result of reading the JHotDraw documentation and implementation and deriving the role models from it. While the first part provides us with a traditional documentation, enhanced with pattern annotations, the second part provides us with a role modeling view of the design.

This partitioning lets us compare the role–model–based version of the class model with the original class model. The evaluation section at the end of the chapter uses this comparison to derive arguments about the suitability of using role modeling for framework design and documentation.

In the final section, the case study first reports about the observations made during determining the role–model–based documentation of the class model. These observations are then related to the problems in framework design that are driving this dissertation.

8.1.3 Chapter structure

The next section presents the JHotDraw framework. It starts with an overview, and then walks through the three major design parts. The final section presents the observations made during this design documentation and relates them to the framework design problems stated in Chapter 1 and 2.

8.2 The JHotDraw framework

This section describes the JHotDraw application framework for graphical drawing editors. It first gives an overview of the design, and then splits it up into three parts: one on the Figure class hierarchy, one on the Drawing and DrawingView classes, and one on the DrawingEditor and associated functionality classes.

8.2.1 Design discussion overview

Figure 8-1 shows the class model of the JHotDraw drawing editor framework (as far as discussed in this case study; this is not a complete model). It lists all classes relevant for the discussion and shows their structural relationships. This class model stems from the JHotDraw tutorial, so we are following Kent Beck and Erich Gamma in picking these classes as the most interesting ones for communicating the JHotDraw (interface) architecture.

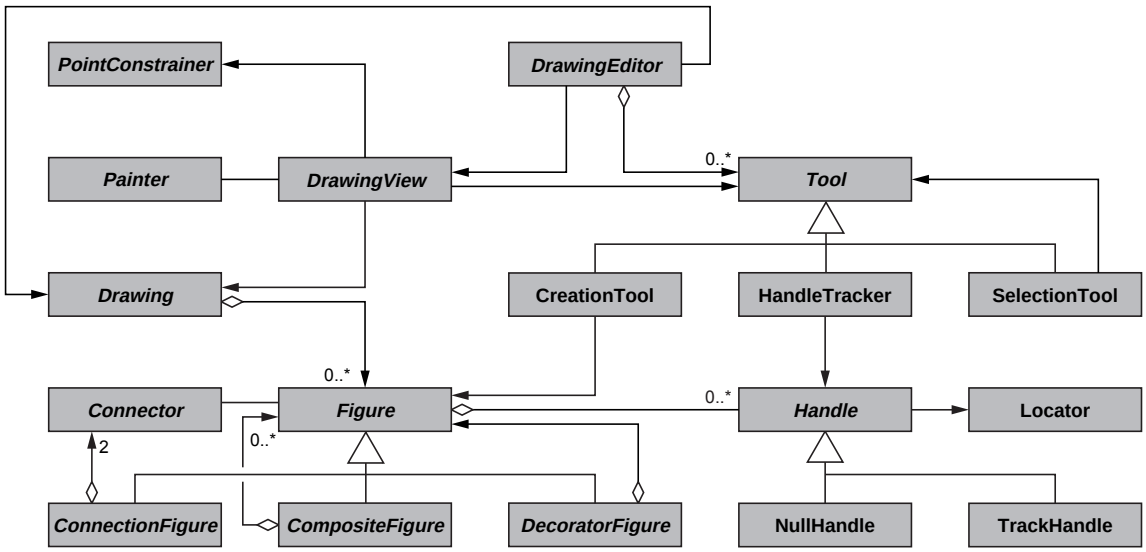

Figure 8-1: The class model of core JHotDraw classes (as chosen for the case study).

The overall design discussion is divided into three parts. The following subsections discuss each part.

- *The Figure classes.* This part describes the Figure class hierarchy and associated classes. It discusses the Figure, CompositeFigure, DecoratorFigure, ConnectionFigure, Connector, and Handle classes.
- *The Drawing and DrawingView classes.* This part describes the use of Figure objects in the context of a drawing and its display in a drawing view. It discusses the Drawing, DrawingView, DrawingEditor, and Painter classes.
- *The DrawingEditor classes.* This part describes the overall drawing editor functionality. It discusses the DrawingEditor, the Tool, CreationTool, HandleTracker, and SelectionTool classes, and the Handle, TrackHandle, NullHandle, and Locator classes.

The design discussion uses the following simplification (rules) to shorten the presentation.

- *Merging of interface and abstract implementation into one class.* Experienced developers factor the code of an important abstraction into at least two parts: an interface that represents the domain concept and an abstract implementation that captures most of the common implementation aspects of the abstraction. The abstract implementation is reused by concrete subclasses [Rie97d, Rie97e].

For our design discussions, these code factoring patterns are unimportant. We therefore merge interface and abstract implementation into a single class. A prime example is the JHotDraw interface Figure and its abstract implementation class AbstractFigure; both become a single class Figure in our design discussion. Other examples are Tool and AbstractTool, and Handle and AbstractHandle.

- *Subsuming a large set of similar classes under a much smaller set of representative classes.* Frequently, an abstraction has a large set of similar subclasses that vary only in minor aspects. Putting all these minor variations as classes into a class model clutters up its visual presentation, but does not add to the discussion. Therefore, we use a small set of placeholder classes to represent the larger set of classes.

  The first example is the use of three classes, DecoratorFigure, CompositeFigure, and ConnectionFigure, to represent the set of all Figure subclasses. A similar example is the use of CreationTool, SelectionTool, and HandleTracker to represent the set of all Tool subclasses. A related but different example is the use of a (fake) placeholder class TrackHandle to represent the different Handle classes figures use to represent their handles.

These are the same rules that Kent Beck and Erich Gamma applied in their tutorial on JHotDraw (at least implicitly, as can be derived by comparing the source code with the tutorial figures).

8.2.2 The Figure classes

This first part of the JHotDraw framework design discussion describes the Figure class hierarchy. Figure is a central abstraction of the drawing editor framework. It represents a graphical figure that users can work with (arrange them to form the drawing). The discussion comprises the classes Figure, CompositeFigure, DecoratorFigure, ConnectionFigure, Connector, Handle, and Drawing. It does not cover all aspects of these classes, but only those relevant from the perspective of the Figure class hierarchy.

8.2.2.1 Original documentation

Figure 8-2 shows the part of the Figure class hierarchy, as it can be reengineered from the code. The figure uses plain UML, and therefore presents the structure of the class model only. Unnamed associations between classes are associations that are not bound to a field (=Java instance variable) of one of the involved classes. They are derived from the abstract state defined in interfaces.

The classes in Figure 8-2 have the following definition (as taken from the source code, and edited and adapted for the case study):

- *Figure*. "A figure knows its display box and can draw itself. A figure can be composed of several figures. A figure has a set of handles to manipulate its shape or attributes. A figure has one or more connectors that define how to locate a connection point. A figure can have an open ended set of attributes. A string identifies an attribute. [...]"
- *CompositeFigure.* "A composite figure is a figure that is composed of several figures. It does not define any layout behavior. It is up to subclasses to arrange the contained figures. [...] A composite figure lets you treat a composition of figures like a single figure."
- *DecoratorFigure.* "A decorator figure is used to decorate other figures with decorations like borders. DecoratorFigure forwards all the method invocations to its contained figure. Subclasses can selectively override these methods to extend and filter behavior. [...] DecoratorFigure is based on the Decorator pattern."
- *ConnectionFigure.* "A connection figure connects connector objects provided by figures. A connection figure knows its start and end connector. It uses the connectors to locate its connection

points. A connection figure can have multiple segments. It provides operations to split and join segments. [...] The Strategy pattern is used to encapsulate the algorithm to locate a connection point. ConnectionFigure is the Context and Connector is the Strategy participant. [...] The Observer pattern is used to track changes of connected figures. A connection figure registers itself as an observer of the source and target figure."

- *Connector.* "A connector knows how to locate a connection point on a figure. A connector knows its owning figure and can determine either the start or the endpoint of a given connection figure. A connector has a display box that describes the area of a figure it is responsible for. A connector can be visible but it does not have to be. [...] A connector is a Strategy used to determine the connections points. [...] Connectors are created by Figure's factory method connectorAt()."
- *Handle.* "A handle is used to change a figure by direct manipulation. A figure may have one or more handles. A handle knows its owning figure, provides its location on the figure, and helps track changes. [...] A handle adapts the operations to manipulate a figure to a common interface."
- *FigureChangeListener.* "A FigureChangeListener object is a listener interested in figure changes." (Listener interfaces are a common Java pattern. A Listener interface represents the callback interface for Observer (= Listener) objects of a given type of subject, here Figure objects).
- *Drawing.* "A drawing is a container for figures. A drawing sends out DrawingChanged events to DrawingChangeListener objects whenever a part of the drawing's area was invalidated. [...] The Observer pattern is used to decouple a drawing from its views and to enable multiple views."

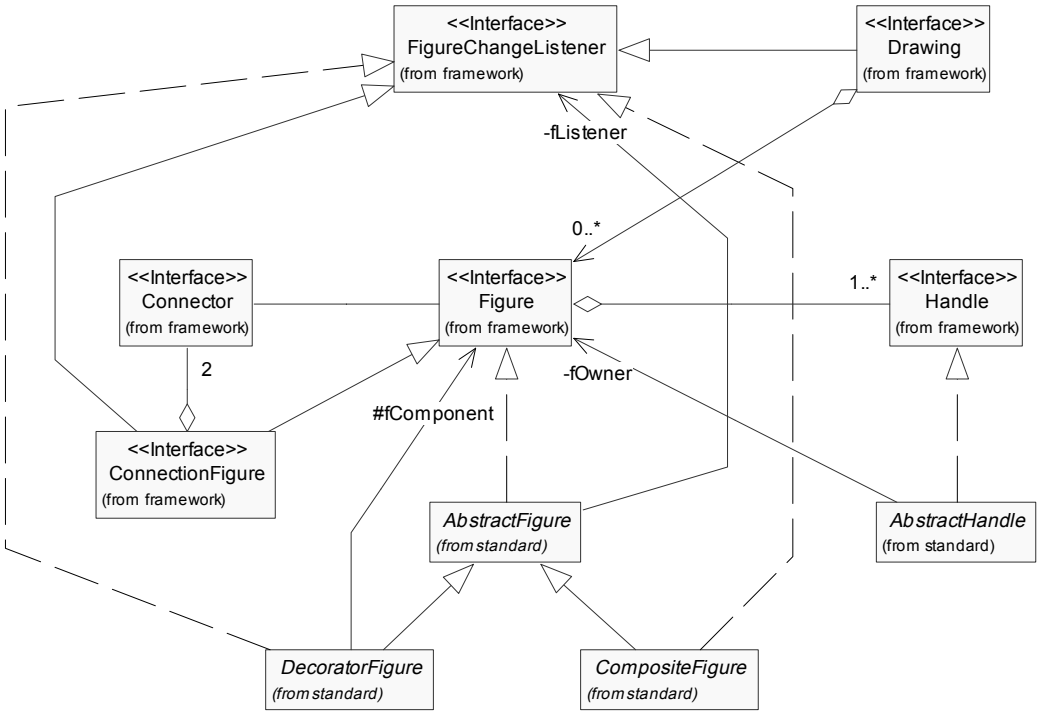

Figure 8-2: Structure of the class model of part of the JHotDraw Figure class hierarchy.

The JHotDraw tutorial provides further information about the collaborative behavior of instances of these classes. It uses design patterns to illustrate it. Figure 8-3 shows the abbreviated and annotated design, as taken from the tutorial.

In this figure, each class is associated with a set of light-blue annotations that name the participant class of a design pattern as defined in the design patterns book [GHJV95]. We have added a few annotations over the original diagrams from the tutorial to make Client and other participants explicit that seem important but were missing.

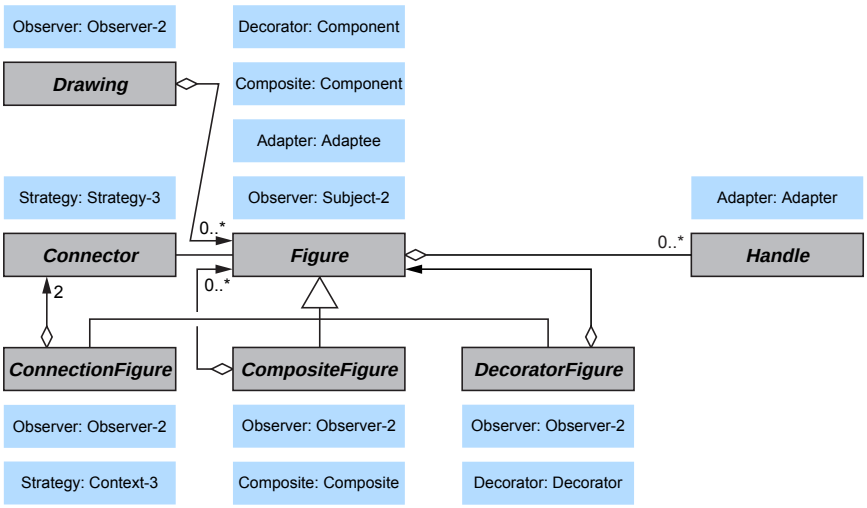

Figure 8-3: Use of design patterns in the Figure class hierarchy.

Figure 8-3 shows the use of the Observer, Adapter, Composite, Decorator, and Strategy design pattern. All of them where mentioned in the class definitions or could be derived from it.

8.2.2.2 Role model documentation

The design patterns mentioned in the class definitions above already illustrate some of the collaborative behavior of instances of these classes. Next to this information, the class definitions point to the use of the following patterns (which were omitted from the pattern-annotated class model above):

- *Property List.* A figure provides a generic set of properties, accessible using strings as their names.
- *Factory Method.* A figure creates connector and handle objects on demand.
- *Observer.* A connection figure observes its start and end connection point.
- *Manager.* A drawing manages several figures as its elements.

Reading the code suggests further role models:

- *Domain functionality.* All classes provide domain functionality not captured by any pattern.

These pattern instantiations and non-pattern role models are displayed in Figure 8-4. This figure shows the full class model behind the discussed design.

The following paragraphs describe the role models from the class model of Figure 8-4. The first part describes all role models that directly relate clients with the Figure class.

- The *Figure* role model lets a Client make use of a Figure object. The Figure class provides the Figure role type, and the Drawing and Handle classes provide the Client role type.
- The *FigureAttribute* role model lets a client get and set any kind of attribute object to a figure object. It is an instance of the Property List pattern. The Client gets or sets Attribute objects to the Provider object. The Figure class provides the Provider role type. The Object class provides the no-operation Attribute role type. Client is a free role type.

- The *FigureObserver* role model lets Observer objects (Java Listeners) observe any figure Subject object. It is an instance of the Observer pattern. The Figure class provides the Subject role type and the classes ConnectionFigure, CompositeFigure, DecoratorFigure, and Drawing provide the free Observer role type.

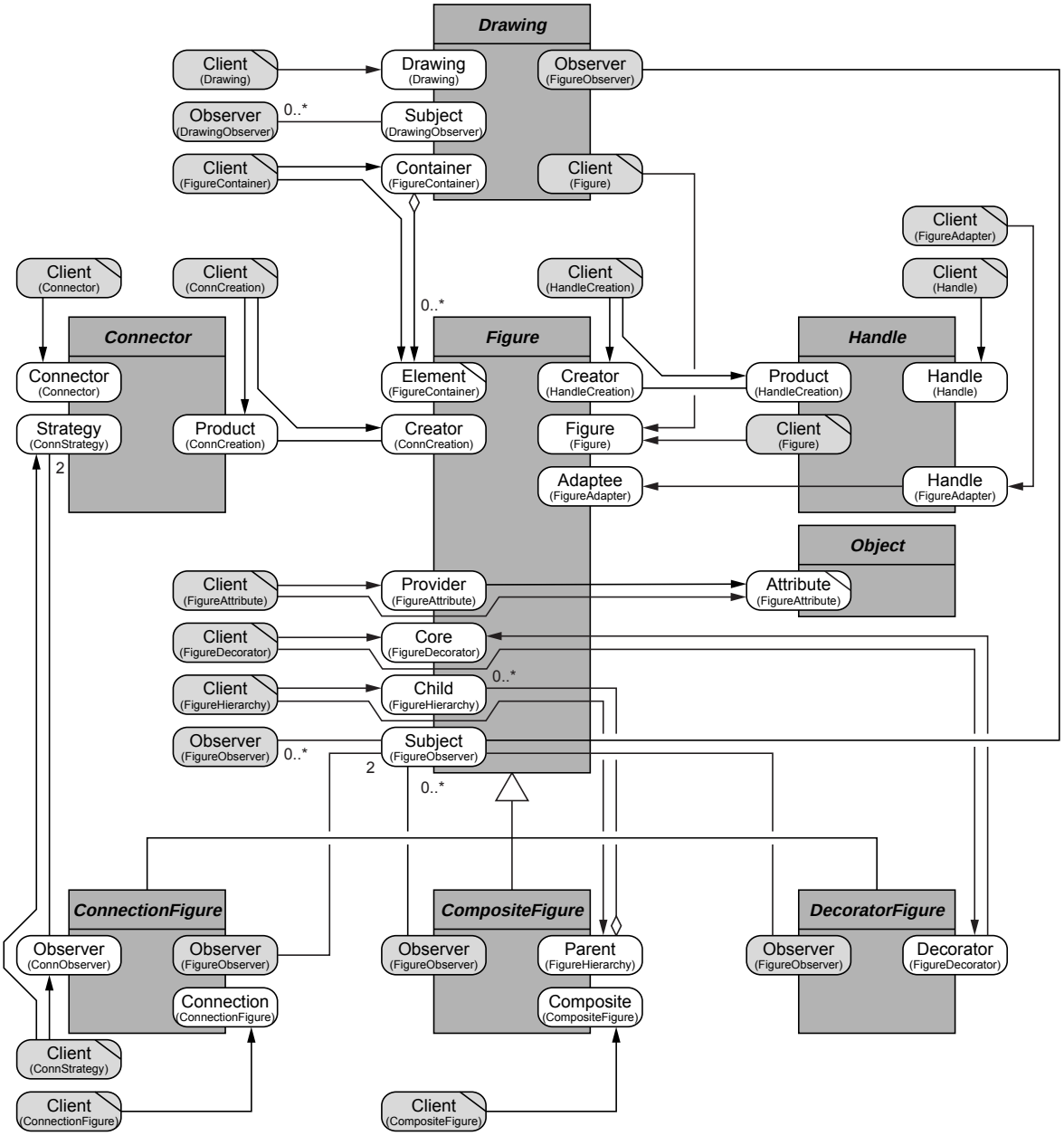

Figure 8-4: Role–model–enhanced class model of Figure class hierarchy.

The second part describes all role models that relate the Figure class with its subclasses.

- The *FigureDecorator* role model lets a client decorate any figure with another figure. It is an instance of the Decorator pattern. A Client object sets the Core object a decorating Decorator object. The Figure class provides the Core role type and the DecoratorFigure class provides the Decorator role type. Client is a free role type.
- The *CompositeFigure* role model lets a Client make use of a Composite figure object. The CompositeFigure class provides the Composite role type. Client is a free role type.

- The *FigureHierarchy* role model lets a Client configure a composite Parent figure with Child figure objects. It is an instance of the Composite pattern. The Figure class provides the Child role type and the CompositeFigure class provides the Parent role type. Client is a free role type.
- The *ConnectionFigure* role model lets a Client make use of a Connection figure object. The ConnectionFigure class provides the Connection role type. Client is a free role type.
- The *Connector* role model lets a Client make use of Connector objects. The Connector class provides the Connector role type. Client is a free role type.
- The *ConnectorCreation* (ConnCreation in Figure 8-4) role model lets a client request connector objects from a figure object. It is an instance of the Factory Method pattern. The Client object requests a new Product object from the Creator object. The Figure class provides the Creator role type and the Connector class provides the Product role type. Client is a free role type.
- The *ConnectorStrategy* (ConnStrategy in Figure 8-4) role model lets a client configure a connection figure with connector objects. It is an instance of the Strategy pattern. The Client configures the Context object with Strategy objects. The Context delegates the computation of a connection point to a Strategy. The ConnectionFigure class provides the Context role type and the Connector class provides the Strategy role type. Client is a free role type.

The third part describes how the Drawing class relates to the Figure class.

- The *Drawing* role model lets a Client make use of a Drawing object. The Drawing class provides the Drawing role type. Client is a free role type.
- The *DrawingObserver* role model lets observer objects register at and be notified about changes by a drawing. It is an instance of the Observer pattern. The Observer object observes the Subject object. The DrawingView class provides the Observer role type and the Drawing class provides the Subject role type.
- The *FigureContainer* role model lets clients add, find, and remove figure objects from a drawing object. It is an instance of the Manager pattern. The Container manages its Elements and provides them to Clients. The Drawing class provides the Manager role type and the Figure class provides the opaque Element role type, and the classes DrawingView and SelectionTool provide the Client role type.

Finally, this last part describes how the Handle class relates to the Figure class.

- The *Handle* role model lets a Client make use of a Handle object. The Handle class provides the Handle role type. Client is a free role type.
- The *HandleCreation* role model lets a client request handle objects from a figure object. It is an instance of the Factory Method pattern. The Figure class provides the Creator role type and the Handle class provides the Product role type. Client is a free role type.
- The *FigureAdapter* role model lets a handle object adapt its owning figure object to client requests. It is an instance of the Adapter pattern. The Handle class provides the Handle role type, Figure class provides the Adaptee role type, and the HandleTracker class provides the Client role type.

Of these 16 role models, 10 are pattern instances.

8.2.3 The Drawing and DrawingView classes

This second part of the JHotDraw framework design discussion describes the Drawing, DrawingView, and related classes. A drawing is a set of figures, displayed in a drawing view. Users manipulate the

figures through the drawing view. The discusses the classes Figure, Drawing, DrawingView, Tool, Painter, PointConstrainer, and DrawingEditor.

8.2.3.1 Original documentation

Figure 8-5 shows the Drawing, DrawingView, and related classes. The figure uses plain UML and therefore presents the structure of the class model only. Again, unnamed associations and aggregations have been derived from the abstract state definitions in the interfaces of the involved classes.

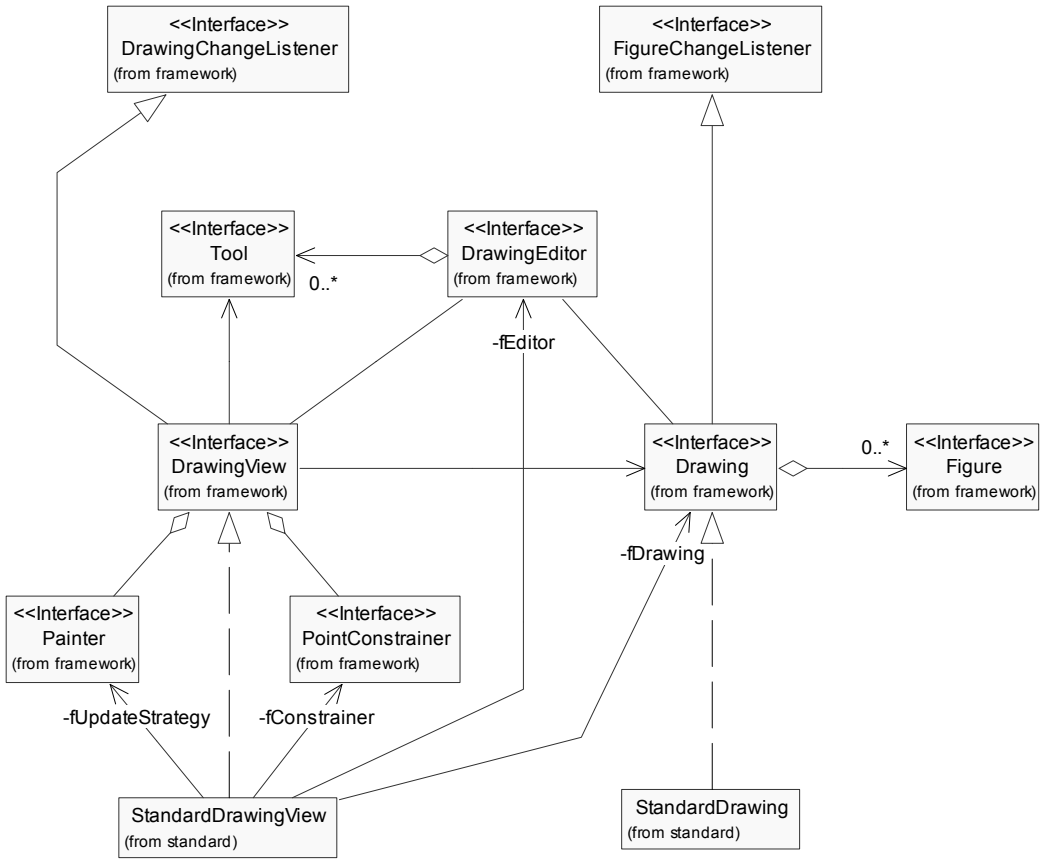

Figure 8-5: Structure of the class model of part of the Drawing and DrawingView classes.

The classes in Figure 8-5 have the following definition (as taken from the source code, and edited and adapted for the case study):

- *Figure, FigureChangeListener, Drawing.* See definition in Section 8.2.2.
- *DrawingChangeListener.* "A DrawingChangeListener is a listener of Drawing instance changes."
- *DrawingView.* "A drawing view renders a drawing and listens to its changes. It receives user input and delegates it to the current tool. [...] A drawing view observes a drawing for changes via the DrawingChangeListener interface. [...] A drawing view plays the Context role in the State pattern. Tool is the State. [...] DrawingView is the Context in the Strategy pattern with regard to the Painter. [...] DrawingView is also the Context for the PointConstrainer."
- *Painter.* "A painter encapsulates an algorithm to render something in a drawing view. A drawing view plays the Context role of the Strategy pattern, and the painter plays the Strategy role."

- *PointConstrainer*. "A point constrainer constrains a point. It can be used to implement different kinds of grids. [...] DrawingView is the Context, and PointConstrainer is the Strategy participant."
- *DrawingEditor*. "A drawing editor coordinates the different objects that participate in a drawing editor. [...] A drawing editor manages possibly several drawing views. DrawingEditor is a Mediator that decouples several participants." (Participant classes are Tool and DrawingView).
- *Tool*. "A tool defines a mode of a drawing view. All input events targeted to the drawing view are forwarded to its current tool. When it is done with an interaction, a tool informs its editor by calling the editor's toolDone() method. A tool is created once and reused. It is initialized/deinitialized with activate()/deactivate(). [...] Tool plays the role of the State. It encapsulates all state specific behavior. A drawing view plays the Context role of the State pattern.

The JHotDraw tutorial provides further information. Figure 8-6 shows the abbreviated and annotated design, as taken from the tutorial.

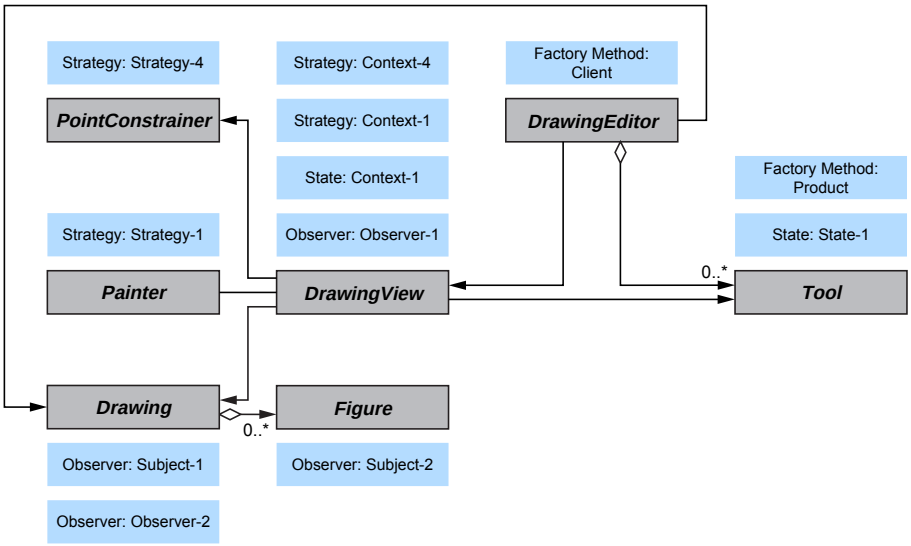

Figure 8-6: Use of design patterns for the Drawing and DrawingView classes.

Figure 8-6 shows the use of the Observer (repeatedly), Strategy (repeatedly), State, and Factory Method design patterns. Again, most of it could be derived straight from the class documentation. Please note that we added the Factory Method Client participant (to class DrawingEditor) over the original documentation. Most patterns are about collaborative behavior of objects, and the Client is one important participant in such a collaboration.

8.2.3.2 Role model documentation

Next to pattern information from the tutorial, the class definitions point to the use of the following patterns:

- *Mediator*. A drawing editor acts as a mediator for the drawing view, drawing, and tool colleagues.

Reading the code suggests further role models:

- *Domain functionality.* All classes provide domain functionality not captured by any pattern.

Figure 8-7 shows these pattern instances as role models together with the non-pattern role models of the domain functionality. This figure shows the full class model behind the discussed part of the design. All role models that have been introduced in the previous subsection are visually grayed-out in the figure and not discussed further.

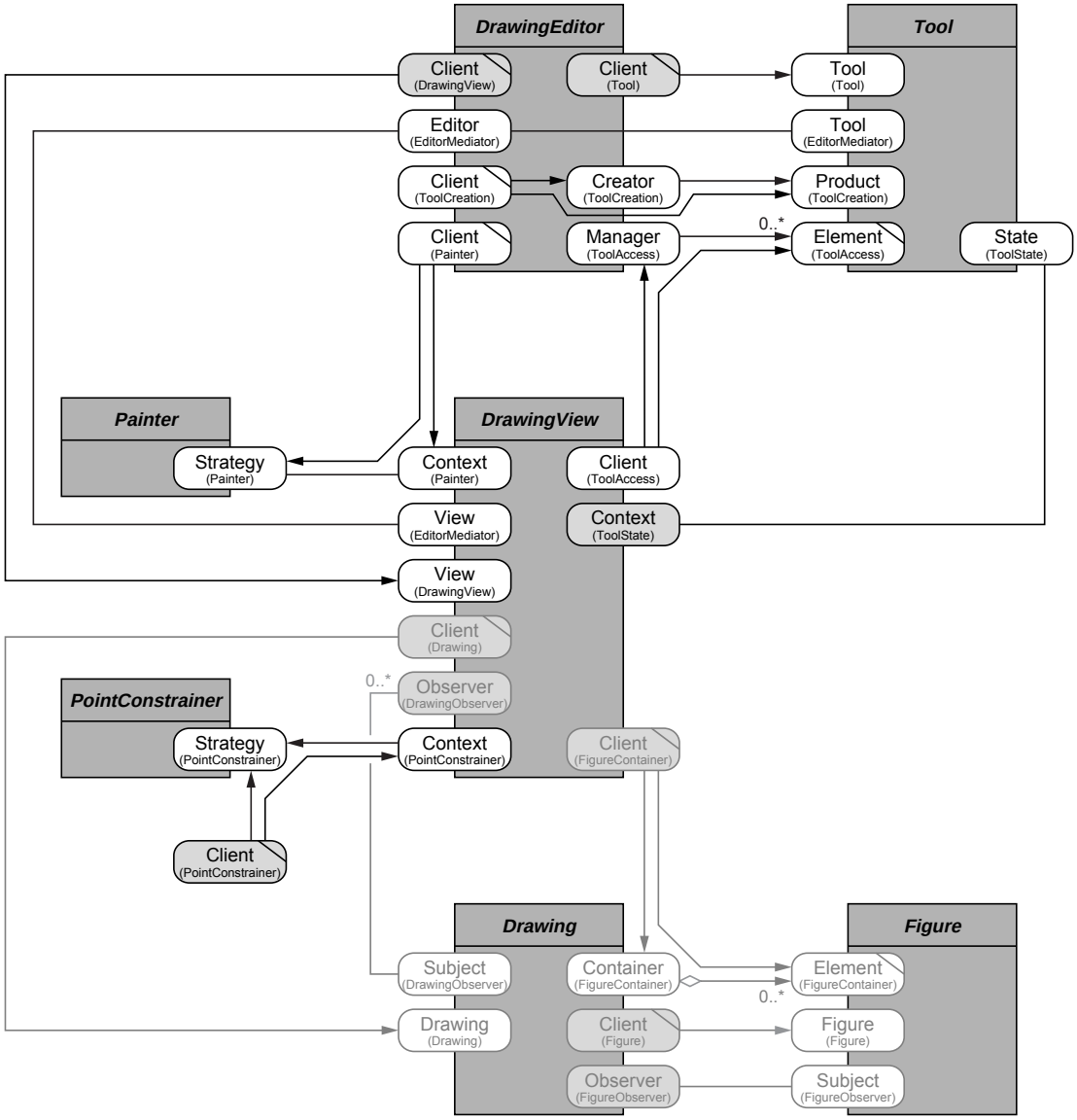

Figure 8-7: Class model of Drawing and DrawingView classes enhanced with role models.

The following paragraphs describe the role models from Figure 8-7 that have not been defined before (for the others, see Section 8.2.2).

The following first part describes all role models that focus on the DrawingView class.

- The *DrawingView* role model lets a Client make use of a drawing View object. The DrawingView class provides the View role type. Client is a free role type.
- The *Painter* role model lets a drawing view delegate the view update to a painter strategy, which calls back on the view to draw it. It is an instance of the Strategy pattern. The Painter class provides the Strategy role type, the DrawingView class provides the Context role type, and the DrawingEditor provides the Client role type.
- The *PointConstrainer* role model lets a drawing view delegate the computation of a grid to a constrainer object. It is an instance of the Strategy pattern. The PointConstrainer class provides the Strategy role type and the DrawingView class provides the Context role type. Client is a free role type.

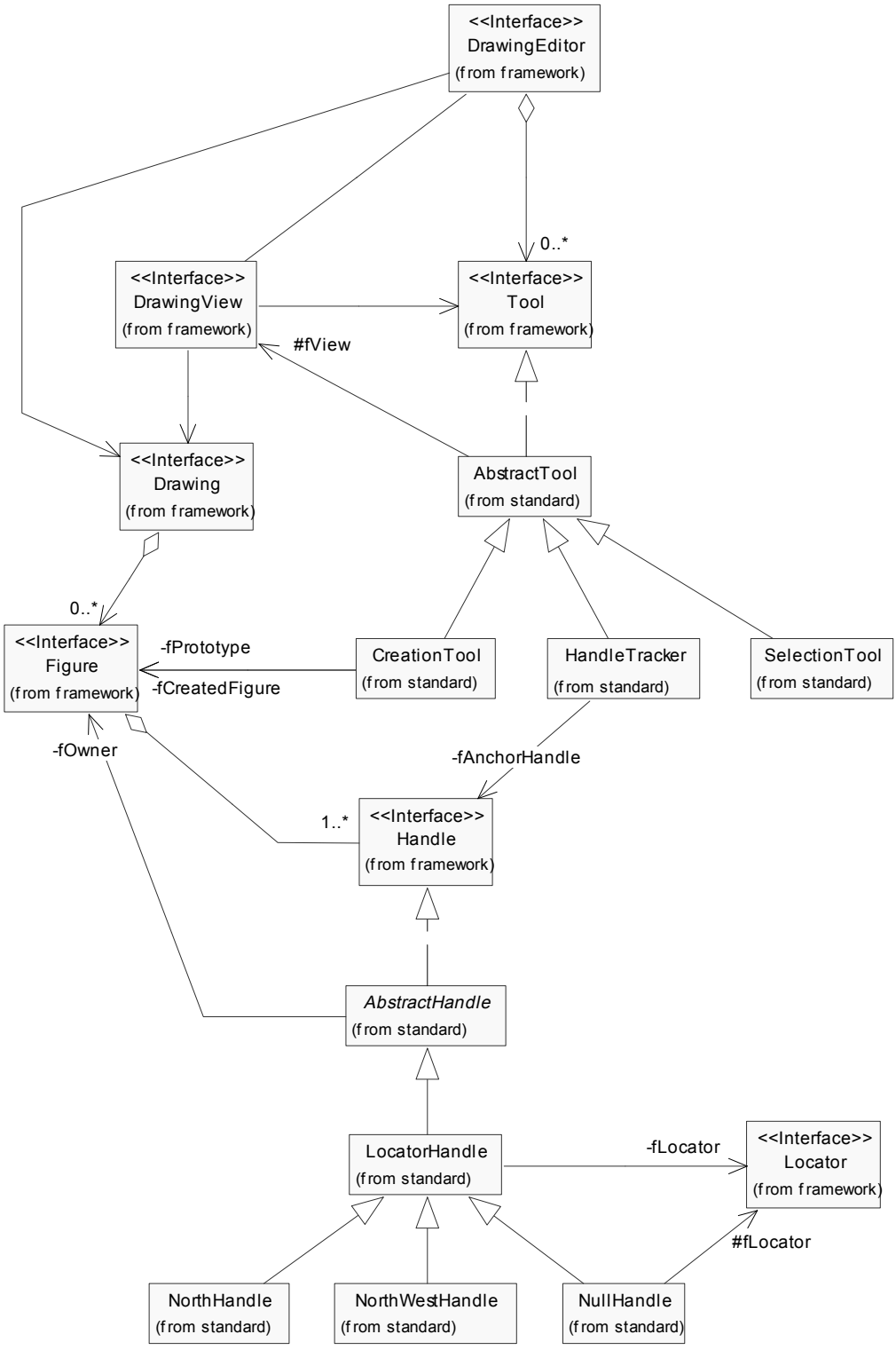

Figure 8-8: Structure of the class model of part of the DrawingEditor classes.

The second part describes all role models that focus on the DrawingEditor class.

- The *EditorMediator* role model decouples drawing views from the drawing and the current tool through an intermediate editor object. It is an instance of the Mediator pattern. The Editor is the mediator, and DrawingView and Tool objects are the colleagues. The DrawingEditor class provides the Editor role type, the DrawingView class provides the View role type, and the Tool class provides the Tool role type.
- The *ToolAccess* role model lets a client request the current tool from the editor. It is an instance of the Manager pattern. A Client object asks the Manager object to return a specific Element object, here the currently active Tool object. The Tool class provides the opaque Element role type, the DrawingEditor class provides the Manager role type, and the DrawingView class provides the Client role type.
- The *ToolCreation* role model lets the drawing editor create a tool object. It is an instance of the Factory Method pattern. The DrawingEditor class provides the Client and Creator role type and the Tool class provides the Product role type.

The third part describes all role models that focus on the Tool class.

- The *Tool* role model lets a client make use of a tool object. The Tool class provides the Tool role type and the DrawingEditor class provides the Client role type.
- The *ToolState* role model lets a client delegate detailed input handling to a tool object. It is an instance of the State pattern. The client object acts as the Context to the tool object that acts as the State. The Tool class provides the State role type and the DrawingView and SelectionTool classes provide the Context role type.

Of these 8 role models, 6 are pattern instances, cast as role models.

8.2.4 The DrawingEditor classes

This third part of the JHotDraw framework discussion describes the DrawingEditor and related classes. The drawing editor is the coordinating object that creates all command and tool objects. Drawing views manipulate figures of a drawing using these tool objects. During this direct manipulation, tools make use of handles.

8.2.4.1 Original documentation

Figure 8-8 shows the class model of the DrawingEditor class and its related Tool and Handle classes. The figure uses plain UML, and therefore presents only the structure of the class model. Again, unnamed associations and aggregations have been derived from the abstract state definitions in the interfaces of the involved classes.

The classes in Figure 8-8 have the following definition (as taken from the source code, and edited and adapted for the case study):

- *Figure, Drawing, DrawingView, DrawingEditor, Tool.* See definitions in Section 8.2.2 and 8.2.3.
- *CreationTool.* “A creation tool is a tool that is used to create new figures. The figure to be created is specified by a prototype. [...] A creation tool creates new figures by cloning a prototype.”
- *HandleTracker.* “A handle tracker is a tool that implements interactions with the handles of a figure.”
- *SelectionTool.* “A selection tool is a tool that is used to select and manipulate figures. A selection tool is in one of three states: background selection, figure selection, or handle manipulation. Different child tools handle the different states. [...] SelectionTool is the Context and a child tool is

the State participant of the State pattern. A selection tool delegates state specific behavior to its current child tool."

- *Handle.* "A handle is used to change a figure by direct manipulation. A handle knows its owning figure and provides methods to locate the handle on the figure and to track changes. [...] A handle adapts the operations to manipulate a figure to the common Handle interface. [...]"
- *TrackHandle.* TrackHandle is a (fake) placeholder class that represents any of the NorthHandle, NorthEastHandle, EastHandle, etc. classes. These classes represent the traditional handles of a rectangular figure.
- *NullHandle.* "A null handle is a handle that does not change the owned figure. Its only purpose is to show that a figure is selected. [...] A null handle lets you treat handles that do not do anything in the same way as other handles."
- *LocatorHandle.* "A locator handle is a handle that delegates the location requests to a locator object."
- *Locator.* "A locator is used to locate a position on a figure. [...] Locators encapsulate a strategy to locate a handle on a figure."

The design patterns mentioned in the class definitions above illustrate some of the collaborative behavior of instances of these classes. The JHotDraw tutorial provides further information. Figure 8-9 shows the abbreviated and annotated design, as taken from the tutorial.

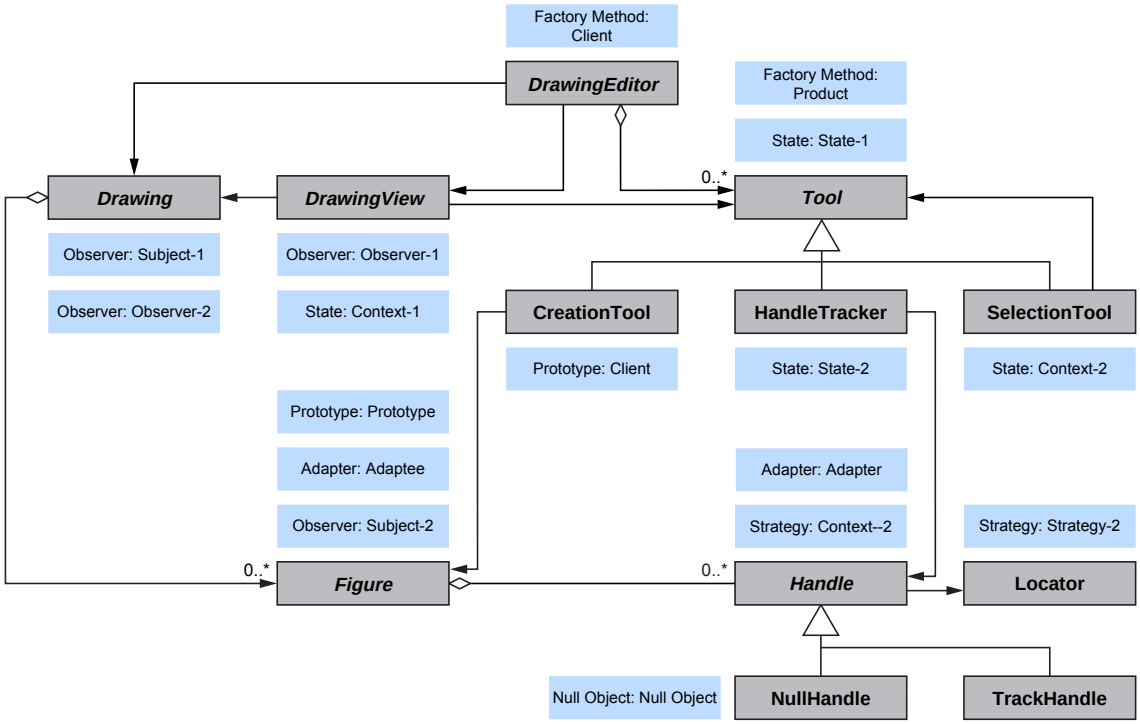

Figure 8-9: Use of design patterns for the DrawingEditor classes.

As mentioned earlier, to simplify the discussion, the Handle and LocatorHandle classes are merged to form a single Handle class. With it, Handle becomes the Strategy context for Locator objects.

This design now shows the use of the Observer (repeatedly), Strategy (repeatedly), State (repeatedly), Adapter, Prototype, Factory Method, and Null Object pattern.

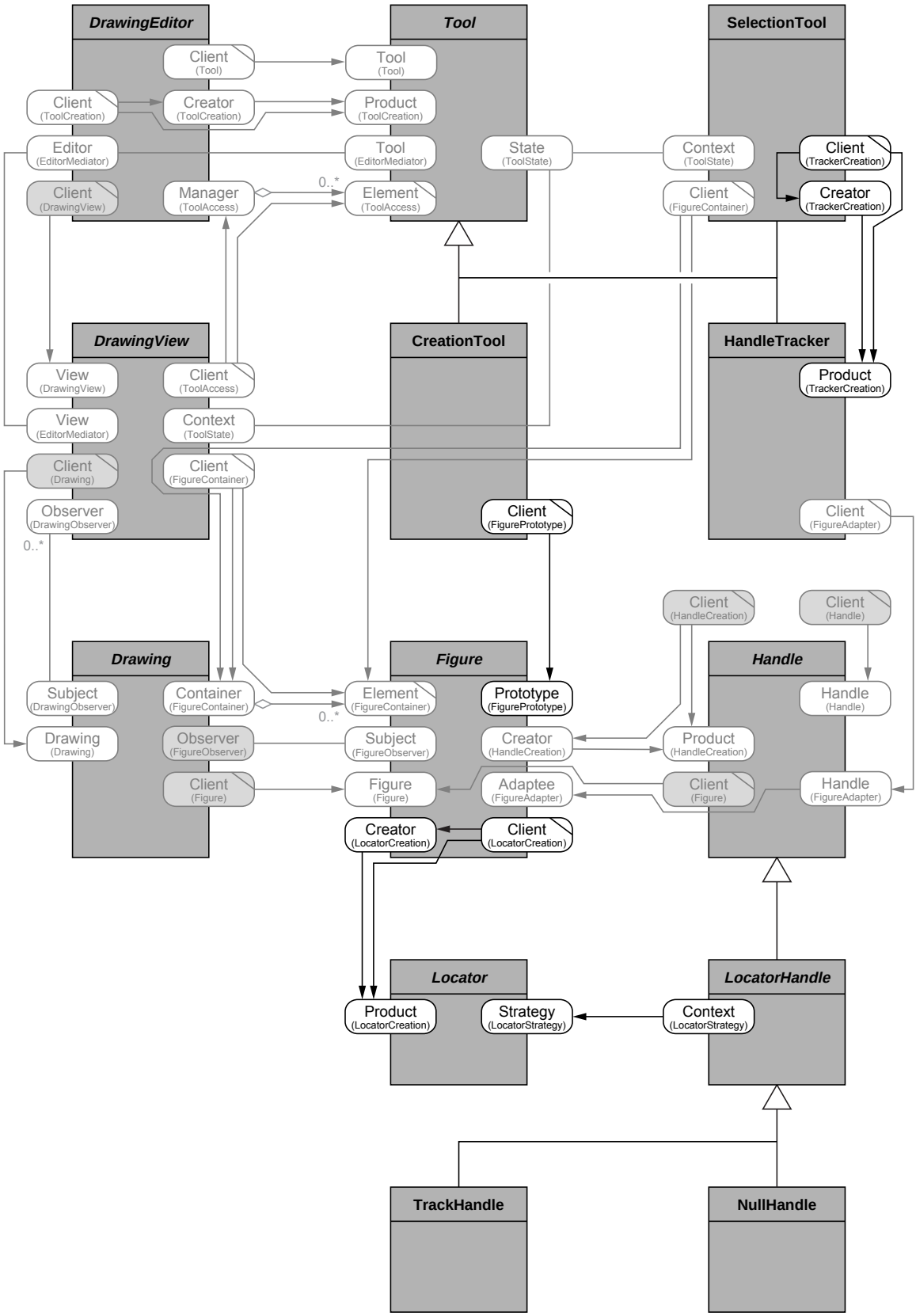

Figure 8-10: Role–model–enhanced class model of DrawingEditor classes.

8.2.4.2 Role model documentation

Next to pattern information from the tutorial, the class definitions point to the use of the following patterns:

- *Factory Method.* A figure creates handles with locator objects. A selection tool creates handle trackers.

Reading the code suggests further role models:

- *Domain functionality.* All classes provide dedicated domain functionality not captured by any pattern.

Figure 8-10 shows these pattern instantiations as role models and adds non-pattern role models that represent the domain functionality. The figure shows the full class model behind the classes selected for this part of the discussion.

The following paragraphs describe the role models of Figure 8-10 that have not yet been defined and discussed (for the other role models, see Section 8.2.2 and 8.2.3).

- The *TrackerCreation* role model lets a selection tool create a new handle tracker object for its own use. It is an instance of the Factory Method pattern. The SelectionTool class provides the Client and Creator role types and the HandleTracker class provides the Product role type.
- The *FigurePrototype* role model lets a Client create a new object by cloning the Prototype. It is an instance of the Prototype pattern. The Figure class provides the Prototype role type and the CreationTool class provides the Client role type.
- The *LocatorCreation* role model lets a figure object create locator objects. It is an instance of the Factory Method pattern. The Figure class provides both the Client and the Creator role type and the Locator class provides the Product role type.
- The *LocatorStrategy* role model lets a locator handle delegate the computation of its position on the drawing area to a locator object. It is an instance of the Strategy pattern. The LocatorHandle class provides the Context role type and the Locator class provides the Strategy role type.

All of these 4 remaining role models are pattern instances.

8.3 Experiences and evaluation

This final subsection presents some statistics from the case study. It examines the observations made during carrying out the case study. These observations are related to the complexity of classes, the complexity of object collaboration, the clarity of expected client behavior, and the reuse of design experience.

8.3.1 Statistics of the JHotDraw framework design

The JHotDraw framework provides us with the data shown in Table 8-1.

Number of classes	20
Number of role models	28

Number of pattern instances	20
Number of role types assigned to classes	66
Ratio of role types per class	3.3
Standard deviation of role types per class	3.02
Ratio of role models per class	1.4
Ratio of pattern instances per role model	0.71

Table 8-1: Raw data and computed figures from the JHotDraw framework.

These figures need to be put into context. The case study describes a large part of the interface architecture of the JHotDraw framework. It focuses on the key classes and omits less important implementation classes. If these less important classes were added, the role type/class and the pattern instances/role model ratios would decline. Less important classes typically add less than the average number of role types per class. Also, less important classes primarily add role models that are not pattern instances, because they represent a simple client/service relationship between a client and the domain functionality of the class.

The pattern instance/role model ration is particularly high, because JHotDraw is a very mature framework in which all design aspects have been worked out thoroughly. By following Kent Beck and Erich Gamma in choosing specifically those classes presented in this case study, we automatically focussed on design aspects that could be cast in pattern form.

8.3.2 Observations from the case study

During the definition of the role model interpretation of the JHotDraw framework design, I made the following observations:

- The JHotDraw documentation infrequently speaks of roles. Also, the way design aspects are presented is frequently close to how we speak about role models. However, there is no explicit mentioning of role models. There are only roles mentioned as part of the class documentation. Also, the original documentation uses the terms role and responsibility synonymously.
- Almost all of the design patterns map easily on role models. The notable exception is the use of the Null Object pattern, which is purely class implementation oriented. Also, understanding the class-based version of a pattern proved to be useful (despite role models) to understand the class hierarchies.
- Role modeling was easy to apply and did not contradict the original documentation in any way. Rather, it enhanced the existing information. It also recast the information in a way that unified the original set of heterogeneous documentation (patterns, code comments, source code) in a common form.
- It was necessary to read the source code to determine some of the role models. The role models that were not present in the original documentation manifested themselves with little if any operations in a role type (for example, LocatorCreation). Yet, these role models represent important design aspects.

- Design patterns are an important part of the documentation, but they do not capture all the collaborative behavior of objects. Object collaboration tasks that cannot be described as pattern instances are missed by a design patterns approach.
- The original documentation did not define well which classes were to act as clients of some other classes and which were not to act as clients. There was seldom a mentioning of which classes were supposed to act as clients of a particular feature of a class.

Based on these general observations, the following subsections conclude that role modeling is a more uniform and more complete way of describing a design than is the set of techniques employed by the original documentation. The following subsections review these observations in the light of the problems of managing class complexity, of managing object collaboration complexity, and of ensuring clarity of use-client requirements. Also, the reuse of experience through design patterns in JHotDraw is discussed.

8.3.3 Comparison of documentation techniques

Both the original JHotDraw documentation and the role modeling documentation are not a complete documentation. Rather they are just one part of a possible more complete documentation. However, each documentation is based on a specific set of techniques.

The JHotDraw documentation, as already pointed out, uses the following techniques:

- *JavaDoc class documentation.* Each relevant class is documented in the code. The text of the documentation describes the purpose of the class and how its instances collaborate with other objects from other classes. A class sometimes also names patterns it is involved in.
- *Pattern annotations.* A traditional class model is used to show the structural relationships between objects and classes. The classes are annotated with participant names of patterns. Each participant identifies (through its pattern) some structural and behavioral aspect of the annotated class.

Role modeling employs two techniques for documenting a design.

- *Class as composition of role types.* A class is described as the composition of a set of role types. (The specific documentation presented here omits a precise specification of this composition and only suggests it through the class definition itself.)
- *Class model with role models.* Class relationships are described by role models. The role models provide a uniform way of defining object collaboration. They both encompass design pattern applications and non-pattern based collaborative behavior.

The case study has not documented the classes and role models in detail. However, it has covered enough ground to compare the two different documentation approaches.

The following subsections compare the two sets of documentation with each other.

8.3.3.1 Documentation of classes

The Figure class serves as an example for the comparison. There is no significant difference between Figure and any other key class, so we can generalize from Figure to the other classes.

The Figure class defines the following roles, as derived from the source code, JavaDoc comments, and the tutorial.

- A figure knows how to display itself (primary domain functionality).
- A figure can be composed from several figures (Composite pattern).

- A figure is manipulated through a set of handles (Factory Method and Adapter pattern).
- A figure provides connectors that define connection points (Factory Method).
- A figure can have an open-ended set of attributes (Property List pattern).
- A figure sends out events to registered observers (Observer pattern).
- A figure can be cloned to create a new figure (Prototype pattern).
- A figure may act as the core to a decorating figure (Decorator pattern).
- A figure provides locator objects for locator handles (Factory Method).
- A figure acts as an element of a drawing (Manager pattern).

The last two responsibilities could only be found by reading the source code. They were not present in the JavaDoc comments or the tutorial. One possible reason is that they were simply forgotten. Another one is that a responsibility without operations (like a figure acting as a drawing element) is easily overlooked. A third possible reason is that the last example is not a pattern from [GHJV95].

Yet it is important to capture all the responsibilities. In the role–model–based documentation, all of these responsibilities are described using role types from specific role models.

These role types are: Figure.Figure, FigureHierarchy.Child, HandleCreation.Creator and FigureAdapter.Adaptee, ConnCreation.Creator, FigureAttribute.Provider, FigureObserver.Subject, FigurePrototype.Prototype, FigureDecorator.Core, LocatorCreation.Client and LocatorCreation.Creator, and FigureContainer.Element.

From this discussion, we can conclude the following:

- The role type documentation is more complete than the original documentation, because role modeling lets us capture all the responsibilities of a class. This includes behavioral responsibilities that do not come with operations of their own and that are not participants of pattern instances.
- The role type documentation is more homogeneous than the original documentation. It uses one concept, role type, rather than several (role, protocol, responsibility, pattern participant). Yet, this single uniformly applied concept delivers a more complete description of a class.

Thus, role modeling gives a more homogeneous and more complete definition of a class.

8.3.3.2 Documentation of collaboration tasks

We continue with the comparison of collaboration tasks using the Figure class. Above, we identified several role types of the Figure class. However, isolated roles do not describe how clients collaborate with a Figure instance. Yet, we can determine the object collaborations from the JHotDraw source code and its documentation.

- General clients use the domain functionality of a figure.
- CompositeFigure instances embed further figures (Composite pattern).
- Handle (subclass) instances are created by a figure (Factory Method pattern).
- Handle instances adapt a figure for interactive use (Adapter pattern).
- Connector instances are created by a figure (Factory Method pattern).
- General clients get and set figure attributes using generic key/value pairs (Property List pattern).
- FigureChangeListener (implementor) instances receive events from a figure (Observer pattern).
- CreationTool instances clone a figure prototype for new figures (Prototype pattern).

- DecoratorFigure (subclass) instances decorate any kind of figure (Decorator pattern).
- Locator instances are created by a figure (Factory Method pattern).
- A Drawing instance manages figures as its elements (Manager pattern).

Some of these collaborations could be derived from the JavaDoc comments, and some of them could be derived from the pattern annotations in the tuto rial. However, the last two collaborations were not documented, neither using JavaDoc comments, nor in the tutorial. One possible reason is again that they were easy to overlook, because they did not come with operations. Another reason is that they are not pattern instances and therefore could not be captured using pattern annotations.

Yet again, these missing collaborations make up an important part of what should be documented about the Figure class and its collaborations. In the role–model–based documentation, all of these collaborations are described using role models.

These role models are: Figure, FigureHierarchy, HandleCreation and FigureAdapter, ConnCreation, FigureAttribute, FigureObserver, FigurePrototype, FigureDecorator, FigureContainer, and Locator-Creation.

From this discussion, we can conclude the following:

- The role model documentation is more complete than the original documentation, because role modeling lets us capture all relevant collaboration tasks. This includes collaboration tasks that are not pattern instances and that are easy to overlook.
- The role model documentation is more homogeneous than the original documentation. It uses one concept, role model, compared to the concepts of class and pattern annotation. Yet, this single uniformly applied concept delivers a more complete description of a class and its collaborations.

Thus, role modeling gives a more homogeneous and more complete definition of the individual object collaboration tasks of a framework and hence of its overall object collaboration.

8.3.3.3 Documentation of requirements put upon use-clients

Finally, we consider the overall set of classes. None of the class documentation makes requirements put upon use-clients explicit. The client is always assumed to make proper use of the framework objects. Also, there is no hint towards which classes clients may use, and which they may not use.

Role modeling in contrast makes all Client role types explicit, whether they are derived from design pattern instantiations or not. In addition, by qualifying role types as free, role modeling lets developers specify which role types are visible to outside clients and may be used by them.

In the context of JHotDraw, we can therefore conclude that role modeling lets us specify requirements put upon use clients, whereas the original techniques do not provide such means.

8.3.4 Complexity of classes

Measured by the size of their role type sets, the most complex classes of the case study are the Figure, DrawingView, DrawingEditor, Drawing, and Tool class. These are 5 out of 20 discussed classes.

The Figure class is a complex class that can be described well and uniformly using role types. The interpretation of the design using role modeling demonstrates this: it provides all the information from the source code and the tutorial, and it adds information that has been missing from the original documentation.

The role–model–based description does not contradict the original documentation. The contrary is true: both the responsibilities defined in the class comments and the design pattern participants anno-

tating the class map directly on role types. The role–model–based description is an extension of the existing documentation, cast in a uniform way.

Traditional class-focussed documentation only provides the structural relationships between classes. The way JHotDraw is documented strongly suggests that this is insufficient to communicate its design. The tutorial tries to overcome this shortcoming by annotating the design with pattern participants. Yet, as the documentation has shown, role modeling delivers a more complete and more homogeneous picture of the design than the original documentation does.

The comparison between role modeling and the JHotDraw techniques, which are already more elaborate than the traditional purely class-focussed techniques, lets us conclude the following:

- *Learning and using complex classes.* As shown above, role modeling is a technique that lets us more effectively describe complex classes. It is simpler to use, because it uses a uniform approach, and it is more complete, because it lets us document all object collaboration tasks (rather than a few selected ones).

  More effectively describing a complex class means making it easier for developers to learn and later to use that class. The description of a class using role types employs a coherent and uniform way of separating the different design concerns involved. This helps developers better learn and use it.

We cannot conclude anything regarding the design process, because JHotDraw was not developed with role modeling in mind (even though use of the employed techniques suggest to me that the developer implicitly used role modeling or something very similar to it).

8.3.5 Complexity of object collaboration

Complex classes are one side of the coin, complex object collaborations the other. Instances of the core classes Figure, Drawing, DrawingView, DrawingEditor, and Tool collaborate with each other in several different ways. Understanding how these collaborations work is vital to understanding the framework design.

The behavior of instances of (subclasses of) Figure in relation to other objects can be described using role models. In contrast to the aforementioned set of heterogeneous techniques, role modeling lets us do this in a uniform and complete way. The role–model–based presentation of the framework captures all the information provided by the original developer and adds information that is missing.

Again, the role–model–based documentation does not contradict the original documentation but rather enhances it. It adds to it where necessary. Design patterns may uniformly document collaborative behavior. However, they do not do this completely; they miss out collaboration tasks that cannot be described as pattern instances.

In the context of JHotDraw, these arguments based on the comparison of the two sets of documentation, let us conclude the following on the use of role modeling for complex object collaborations.

- *Learning and using complex collaborations.* As shown above, role modeling is a technique that lets us more effectively describe object collaboration behavior than possible with the techniques employed in the JHotDraw documentation. It is more effective than design patterns, because patterns ignore role models not based on patterns.

  More effectively describing object collaboration behavior means making it easier for developers to learn and use the classes involved in the collaboration. The need for breaking up the overall collaboration into pieces is recognized in the original JHotDraw documentation through the use of design patterns, but is only carried out fully through the use of role models.

JHotDraw was not designed with role modeling in mind, so we cannot conclude anything on the design process of the object collaboration tasks.

8.3.6 Clarity of requirements put upon use-clients

Role modeling makes requirements put upon use-clients explicit, as far as the method allows. The original documentation does not address this problem.

In the context of JHotDraw, this lets us conclude:

- *Learning requirements put upon use-clients, and using a framework according to these requirements.* Both are eased through role modeling, because the original techniques do not provide any support here.

JHotDraw was not designed with role modeling in mind, so we cannot conclude anything on the process of defining the requirements put upon use-clients.

8.3.7 Reuse of experience through design patterns

JHotDraw, and hence this case study, shines when it comes to reuse of experience through design patterns. JHotDraw exhibits a high “design patterns density”, i.e., its design is based on many pattern instances. However, the JHotDraw pattern annotations and the role modeling technique have different characteristics.

Annotating classes with participant names from a design pattern indicates that instances of the class exhibit behavior within the context of the pattern application according to the definition of the particular participant. However, there is no 1:1 mapping between a design pattern as presented in [GHJV95], and a specific class structure, nor is there a 1:1 mapping between a participant and a specific type or interface. It is not clear how specific operations and operation signatures look like, because a pattern is not the same as its application.

Therefore, annotating classes with pattern participants gives hints to users about the collaborative behavior of its instances, but does not precisely specify this behavior. Users will always have to read the class documentation and the source code to determine which operations belong to which participant.

Role models, on the other hand, are always a concrete rather than an abstract design artifact. The involved role types specify precisely (within the capabilities of the chosen specification mechanism) the behavior of objects conforming to this type. However, to get a quick grasp at a role model that is a pattern instance, users have to map a pattern participant onto a role type. Here, the role–model–based design patterns catalog [Rie97c] helps significantly, because it already casts the pattern in role model form.

Therefore, with role modeling users do not have to bridge a gap between a design pattern and a concrete design artifact. There is no hurdle of understanding a design pattern first, before they can understand a design.

However, to reuse their design experience, they must connect the specific role model at hand with an abstract design pattern. As I have described earlier [Rie96a], role modeling makes it easier to represent and apply design patterns, because it adds more flexibility to allocating role types to classes.

Role modeling has a lower entrance hurdle than the original JHotDraw documentation technique: it does not require developers to know a pattern to understand a design aspect. However, to fully benefit from it, design patterns should be known. Role modeling then makes it easy to recognize these patterns, because it more flexibly allows their application in a design.

9

Thesis Validation

This chapter validates the dissertation thesis. It first reviews the thesis and then devises a validation strategy. Based on the form of the thesis statement, the validation strategy is to split the thesis up into nine parts, each of which can be validated individually. Every sub-validation is carried out using general arguments about object-oriented framework design that are backed by the observations made and experiences gained in the case studies. The validation of the dissertation thesis follows from the sub-validations.

9.1 Thesis review and validation strategy

Chapter 2 presents the final version of the thesis of this dissertation:

> **Thesis statement of dissertation (final version, taken from Chapter 2)**
>
> Role modeling for framework design makes the following activities easier to carry out for the expert framework developer and user than is possible with traditional class-based approaches:
>
> - designing and redesigning a framework;
> - learning a framework from its documentation;
> - using a framework that is already understood;
>
> The following problems are addressed and their severity is reduced:
>
> - complexity of classes;

- complexity of object collaboration;
- clarity of requirements put upon use-clients.

This form of the thesis helps to determine what to prove and which constraints to adhere to while doing so.

First, a validation of the thesis must be based on a comparison between the "traditional class-based approach" and the role modeling approach. Unless stated otherwise, role modeling always means "role modeling for framework design as defined in this dissertation". Traditional class-based approach means what you can natively express using classes as the only central modeling concept (for example, UML). Also, the discussion is restricted to expert developers and users only.

Then, the thesis is split up into different parts: designing and redesigning a framework, learning a framework from its documentation, and using an already understood framework. These are separate activities that can be discussed and validated independently.

Finally, the thesis defines in which respect role modeling eases the different activities by stating which problems it addresses: class complexity, object collaboration complexity, and clarity of requirements and constraints put upon use-clients.

This leads to a matrix of sub-theses (claims) to be validated, displayed in Table 9-1.

(activity, problem) matrix	*designing and redesigning a framework*	*learning a framework from its documentation*	*using an already understood framework*
complexity of classes	Validity to be shown.	Validity to be shown.	Validity to be shown.
complexity of object collaboration	Validity to be shown.	Validity to be shown.	Validity to be shown.
clarity of requirements put upon use-clients	Validity to be shown.	Validity to be shown.	Validity to be shown.

Table 9-1: The dissertation thesis broken up into nine sub-theses.

The matrix has the set of activities as its X-dimension, and the set of problems as its Y-dimension. A matrix cell is referenced using A for activity, P for problem, and a digit for the particular activity and problem. For example, A2P1 is the top middle matrix cell (activity: learning a framework; problem: complexity of classes).

For each cell in the matrix, it needs to be shown that the X dimension, the activity, becomes easier (or stays the same) with respect to the Y dimension, the problem, if one compares the traditional class-based approach with the role modeling approach.

Because the dissertation thesis is viewed as the conjunction of these nine sub-theses, the overall thesis validation becomes the conjunction of the nine sub-validations.

The matrix form suggests a validation strategy in which each sub-thesis is assessed and validated individually. For nine sub-theses, this is a tedious and repetitive undertaking, because many arguments and experiences apply to several of them.

A better validation strategy is to walk through a set of key properties of the role modeling approach and show how they lead to arguments that validate one or more of the nine sub-theses. In a second step, for each sub-thesis, the arguments can then be put together to form the overall validation of the sub-thesis. With all nine sub-theses done, the overall thesis validation is also done.

The next subsection carries out the thesis validation using this strategy.

9.2 Thesis validation

This subsection presents the actual validation of the dissertation thesis. It follows the validation strategy outlined above: it first walks through a set of key properties of the role modeling method, and then consolidates the resulting arguments for each sub-validation. The thesis validation follows as the conjunction of the sub-validations.

9.2.1 Describes class as composition of role types

Role modeling, as defined in this dissertation, lets developers describe classes as compositions of distinct role types. Or, viewed the other way round, class interfaces are broken up into role types. A role type is used in two different contexts. In the first context, the role type is part of a role model, where it is defined. In the second context, a role type is part of the role type set of a class, where the class defines how the role types are composed to determine the behavior of its instances. Traditional class-based modeling offers no appropriate means to split up a class interface into distinct parts.

When dealing with a class, developers switch between the two views. On the one hand, they focus on the role type in the context of its role model to define, use, or understand the specific aspect of the class being described by the role type. On the other hand, they focus on the composition of the role types to understand what makes up the class as a whole and how acting in the context of one role will cause actions in the context of another role of instances of that class.

Describing a class as the composition of role types and breaking up a class interface into role types separates design concerns along the lines of object collaboration tasks. This property of the approach helps with the following activity/problem pairs:

- *Designing complex classes of a framework (A1P1).* Role modeling makes it easier to design a class and its interface, because keeping the role model and the class view separate and being able to easily switch between them reduces the complexity of the design task.

  The experiences with designing the Geo frameworks and the Tools framework support this argument. For example, the Geo system has several complex service interfaces. Splitting them up into different role types significantly reduced the complexity of the remaining design task.

- *Learning complex classes of a framework (A2P1).* Role modeling makes it easier to understand a class, because keeping the role model and the class view separate and being able to easily switch between them reduces the complexity of the learning task.

  It was a common experience in the Geo project that it is easier to learn key classes from a framework designed and documented using role modeling than possible with documentation based on traditional class based modeling. The same argument holds true for the JHotDraw framework.

- *Using a class from an well-understood framework (A3P1).* Role modeling makes it easier to use a class, because it is better understood, and because developers can focus on the different uses of a class independently of each other.

  Describing classes as compositions of role types eased using key classes from the Geo frameworks. Also, when designing and implementing client classes, the focus is on single role types rather than full classes. This reduces the complexity of defining client classes further.

The property «describes class as composition of role types» of the role modeling approach therefore significantly eases designing, learning, and using complex classes of a framework. This addresses the activity problem pairs A1P1, A2P1, and A3P1.

9.2.2 Breaks up relationship descriptions into role models

Role modeling, as defined in this dissertation, breaks up the object association or aggregation descriptions of traditional class-based modeling into role models. A role model focuses on one particular object collaboration task, while a traditional object relationship description comprises all possible tasks carried out on an instance of the relationship description. The object relationship description between two classes becomes the sum of the role relationships descriptions between role types from role models that connect these two classes. Traditional class-based modeling offers only object relationship descriptions between classes, but no role models.

Role models can be viewed as zooming in on an object relationship description. They detail what is going on between the involved classes. Developers can therefore maintain two different views on how classes relate to each other. They can see the traditional object relationship descriptions that determine, which object may relate to which other object in what quantities, etc., and they can zoom in on any such relationship description to determine the different object collaboration tasks carried out using instances of these relationship descriptions.

Chapters 3 and 4 have shown that the role model view is an extension of the relationship description view, and that the object relationship descriptions between classes can be derived from role models. The object relationship descriptions determine the base skeleton, and the role models describe the behavior of objects working along this predefined structure. These complementary views help with the different activities and their problems.

Breaking up relationship descriptions into role models eases handling the complexity of object collaboration. This property of the approach helps with the following activity/problem pairs:

- *Designing the object collaboration of a framework (A1P2).* Role modeling eases designing the framework's object collaboration, because developers can zoom in on the object relationship structure, determine the tasks carried out along its line, and switch back to the big picture again. They can work on a detail level that is best for the current design issue, be it the overall structure or the individual collaboration tasks.

  The experiences with the Geo frameworks and the Tools framework directly support this argument. During design sessions team members continuously switched between the class structure and their relationship descriptions on the one hand, and the detailed discussion of how instances of these classes collaborate on specific tasks using role models on the other hand. Many of these role models were design pattern instantiations, which made them a well-defined design issue in itself.

- *Learning the object collaboration of a framework (A2P2).* Object relationship descriptions tell us about the structural relationships between objects of a framework, but they tell us nothing about the tasks they carry out. Role models do that. Role modeling significantly eases learning the object collaboration, because it makes information explicit that would otherwise be lost or described in other (then sub-optimal) ways.

  Again, the experiences from the case studies directly support this argument. When team members had to understand a new design or learn an existing framework, the two views of object relationship descriptions and role models helped them better and faster understand the intent of a design. Also, team members could more readily recognize design patterns based on the catalog of role model design patterns [Rie97a]. This helped reusing prior experience.

- *Using classes from an well-understood framework (A2P3).* Using role models, use-clients can be precise about what they want from a framework class (by picking up a specific free role type).

Using object relationship descriptions, use-clients always get a bundle of role models, of which they may only be interested in one. Also, in traditional class-based modeling, use-clients without operations are not represented at all, even if they play a significant role. Role models provide free role types even for use-clients that have no operations. Hence important information is documented that gets lost in class-based modeling.

The case study experiences support this argument. For example, the Geo service interfaces clearly separated the different issues of using the services as free role types. Team members could therefore distinguish between the different tasks like retrieving the service object, configuring it with information, and using its primary functionality. These tasks could be carried out by different objects and were not bound to one specific client class.

The property «breaks up relationship descriptions into role models» of the role modeling approach therefore significantly eases designing, learning, and using a framework that has complex object collaboration. This addresses the activity/problem pairs A1P2, A2P2, and A3P2.

9.2.3 Makes requirements on clients explicit

Role modeling, as defined in this dissertation, makes requirements and constraints put upon use-client classes explicit. The requirements and constraints are described using role types and role constraints. The role types are the free role types of a framework, and the role constraints are those constraints that relate to free role types. How exactly requirements and constraints on clients are specified depends primarily on the chosen type specification mechanism.

Traditional class-based modeling offers two possibilities to define requirements and constraints put upon clients:

- classes or interfaces that client classes have to inherit from;
- class specifications as part of a framework class interface.

This has the following problems:

- Making clients inherit from classes to enforce constraints is too heavyweight an approach, if clients have to inherit unwanted baggage (implementation state, operation implementations). Partially or fully implemented classes are not always an adequate means for specifying client constraints.
- Making clients implement specific interfaces is frequently better than making them inherit from classes. If an interface represents a role type, the role modeling approach and the traditional class-based approach are equivalent. However, in the traditional approach, such interfaces are used only if operations are associated with them. No-operation free role types are missed.
- Making requirements and constraints put upon clients part of a framework class specification forces developers of any client for any collaboration task to keep in mind the whole specification. Splitting it up into role types separates different behavioral aspects of the class and therefore reduces complexity. (See the arguments of why to break up a class interface into role types).

Moreover, the traditional approach to requirements and constraints specification puts the specification into the wrong place, namely the framework class. It is better to put it into the free role types, because the requirements are requirements on clients, and need to be adhered to by them, and not by the framework classes. However, as long as there was no concept of free role type, there was no possibility of making explicit that use-clients are responsible for adhering to the specifications. Thus, they were specified as part of a framework class interface.

Finally, the free role types are part of free role models. The additional view on object collaboration tasks provides the benefits of breaking up class interfaces into role types and breaking up object rela-

tionship descriptions into role models. This also applies to the specification of requirements and constraints put upon clients. (See the discussion of these role modeling properties above).

For these reasons, role modeling provides a better background for specifying the requirements and constraints put upon clients than traditional class-based modeling. Role modeling therefore helps with the following activity/problem pairs:

- *Designing use-client requirements (A1P3).* Free role types are a better means of enforcing requirements and constraints than classes. They are also a better means than interfaces, if these are only used to specify requirements that are attached to operations. Also, role types stem from role models, and therefore show how they tie in with the framework. Traditional class-based modeling falls short here as well.

  The experiences with the Geo frameworks and the Tools framework support this argument. For every role type provided by a framework class, the defining role model had to specify at least one client role type. Team members therefore always had to ask themselves what a specific role type was good for, and how the counterpart in the described collaboration task had to look like. This is in contrast to earlier experiences with traditional class-based modeling, where it was easy to forget about clients, and that they also had to fulfill requirements and adhere to constraints.

- *Learning how to use a framework (A2P3).* Free role types specify requirements and constraints well and completely (as far as possible with the chosen type specification mechanism). Moreover, they provide separation of concerns as discussed above in the subsections on breaking up class interfaces and object collaborations. Both make it easier to learn how to use a framework.

  Again, this argument is supported by the experiences from the case studies. The primary reason is the separation of concerns achieved by defining free role types and by breaking up the relationship descriptions into role models. In class-based modeling, client requirements are either not specified or, at least, they are more difficult to understand. Thus, role modeling reduced the complexity of understanding all requirements and constraints.

- *While using a framework, adhering to its requirements and constraints (A3P3).* Every use-client has to specify explicitly how it ties in with the framework by stating which free role types it uses. None can be forgotten or omitted. This requirement allows checking for proper use of the framework.

  This argument is also supported by the case study experiences. First, learning a Geo framework more easily also made it easier to lay out how to use it. Second, when using it, the free role types and their description where a constant reminder and measure of how use-client classes had to look like and how their implementation had to behave.

The property «makes requirements on clients explicit» of the role modeling approach therefore eases designing, learning, and using a framework with respect to the requirements and constraints put upon clients. This addresses the activity/problem pairs A1P3, A2P3, and A3P3.

9.2.4 Supports reuse of experience

Role modeling, as defined in this dissertation, better supports developers in reusing previous design experience than possible with a traditional class-based approach. Reuse of design experience can take on two forms in this context. First, developers may adapt earlier designs to new requirements and derive the new design from this, and second, developers may apply design patterns [GHJV95, RZ96].

In comparison to class-based modeling, role modeling eases reusing experience, because the separation of concerns it achieves makes designs more readily decomposable into pieces and recomposable from these pieces. Role modeling supports this particular well, because it works along the lines of object collaboration, which are a main focus of design and also of reusing experience.

- *Reuse of experience through design adaptation.* In this form of reuse, a developer reconsiders an old design, takes out the pieces not needed in the new design, changes the pieces according to new requirements, and adds new pieces as required for the new design. The result is the new design. An example are the Geo service interfaces, which are very similar, but differ in the details of the actual services provided.

  If the old design has a clear role modeling description, reusing experience this way is made easier, because the old design already separates the different concerns that become the individual focus of attention when adapting the design for a new situation. The design is adapted along the lines of the object collaboration tasks that are described by the role models of the old and then new design.

- *Reuse of experience through design patterns.* In this form of reuse, an experienced developer recognizes a design problem and recalls a design pattern that he applies to solve the problem in the given context. Design patterns can take on many different forms. The most common object-oriented design patterns have been described in the seminal work of Gamma et al. [GHJV95]. Each of the design patterns from this catalog makes one specific design aspect flexible so that it can be changed easily.

  Most of these patterns are about the distribution of responsibilities among the objects of a design, and how the responsibilities are assigned to classes. Using role modeling terminology, each pattern instance focuses on one specific object collaboration task, and the assignment of responsibilities to classes corresponds to putting role types from a role model into the role type sets of classes. Of the patterns from the design patterns catalog, the larger part can be described well using role modeling [Rie97a, Rie97c]. (Also see Appendix D.)

  In a design, a design pattern application needs to be composed with other design pattern applications and role models. Recasting design patterns in role model form makes it easier to keep the different collaboration tasks separate, and thereby eases the reuse of experience. As I have demonstrated, using role models for describing design patterns makes them more flexible and more easily applicable, without making them use their particular patterns quality [Rie96a]. Traditional class-based modeling offers no such support.

Being able to reuse experiences addresses every kind of problem in object-oriented framework design. Therefore, role modeling better supports all of the activities and eases all of the problems than class-based modeling does (in this respect).

When designing the Geo system, the team made use of its own version of the design patterns catalog, which recast all the common design patterns in role model form. This catalog was a constant companion, and helped team members in all of the activities regarding all of the problems.

The use of design patterns is ubiquitous in the Geo frameworks, the Tools framework, and the JHotDraw framework. They have been applied to form or used to describe free role models as well as internal role models. They serve to more readily understand the framework's client collaboration as well as its inner workings. And they support using the framework.

Therefore, the property «supports reuse of experience» of the role modeling approach eases designing, learning, and using a framework with respect to all stated problems. This addresses the activity/problem pairs A1P1 through to A3P3.

9.2.5 Consolidation of validation

There are further properties of role modeling that help with framework design and use (for example, frameworks are made explicit design artifacts with well-defined boundaries) and that have not been discussed in the previous subsections. However, the purpose of this section is to validate the dissertation thesis. Therefore, properties that do not directly contribute to this validation have been omitted.

The previous subsections discussed the following properties and examined their effect on the activity/problem pairs derived from the dissertation thesis.

- *Section 9.2.1: Describes class as composition of role types.* Helps ease the problem of dealing with the complexity of classes for all three activities.
- *Section 9.2.2: Breaks up relationship descriptions into role models.* Helps ease the problem of dealing with complex object collaboration of a framework for all three activities.
- *Section 9.2.3: Makes client requirements explicit.* Helps ease the problem of specifying requirements and constraints put upon use-clients of a framework for all three activities.
- *Section 9.2.4: Supports reuse of experience.* Helps ease every problem for every activity, because developers have experience to reuse for all of them.

Table 9-2 shows where to find the arguments that prove a particular activity/problem pair.

(activity, problem) matrix	*designing and redesigning a framework*	*learning a framework from its documentation*	*using an already understood framework*
complexity of classes	• 9.2.1 • 9.2.4	• 9.2.1 • 9.2.4	• 9.2.1 • 9.2.4
complexity of object collaboration	• 9.2.2 • 9.2.4	• 9.2.2 • 9.2.4	• 9.2.2 • 9.2.4
clarity of requirements put upon use-clients	• 9.2.3 • 9.2.4	• 9.2.3 • 9.2.4	• 9.2.3 • 9.2.4

Table 9-2: Where to find the arguments for the sub-validations.

The validation of the dissertation thesis is the conjunction of the validation of the nine sub-theses. Table 9-2 shows in which subsection each sub-thesis has been addressed and validated. For each sub-thesis, there are at least two major validating arguments. Because the thesis dissertation is the conjunction of these sub-validations, Table 9-2 concludes the thesis validation.

9.3 Summary (meaning of validation)

The thesis validation shows that role modeling as defined in this dissertation is superior to traditional class-based modeling with respect to the problems stated initially in this work. As reviewed in Chapter 2, related work has also tried to address some of the problems that role modeling solves. However, none of this related work achieves this density of problem solving as role modeling.

It is the nature of a validation to focus exclusively on what needs to be validated and to ignore any other consequences. However, it may be exactly the side effects, why a specific statement, in this case the dissertation thesis, was set up in the first place. Thus, the validation does not tell the full truth about why role modeling represents an improvement over current practice.

In explaining the thesis as the sum of several distinct parts, it became possible to more precisely describe the thesis, and to devise a validation strategy based on breaking it up into parts. Doing so, however, the whole, which is more than the sum of its parts, got lost. Much of the power of role modeling for framework design as defined in this dissertation stems from the interaction of the different parts. Without role types, there would be no role models. Without role models, there would be no precise

definition of client interaction. Without role models, there would only be limited increase of reuse of experience. Etc.

However, for the purposes of the thesis validation, this is irrelevant. It needed to be shown that role modeling is a significant improvement over current practice, and this has been done. How the validation result is used is not to be defined by the dissertation. It will become apparent in the practice of using role modeling for framework design.

10

Conclusions

This chapter sums up the contributions made by the dissertation, points towards future work, and provides final conclusions on the results of this dissertation.

10.1 Contributions

The primary result of this dissertation is that role modeling for framework design, as described in Chapters 3 and 4, makes the design of object-oriented frameworks easier than is possible with traditional class-based approaches. This claim is detailed in Chapter 2. Naturally then for a dissertation, most parts of the exposition focus on validating precisely this claim. Chapters 6 to 8 provide case studies of using this method, and Chapter 9 validates the thesis based on the case studies.

Next to validating the thesis, this dissertation presents a novel modeling approach to framework design. It is both a precondition for the thesis as well as a major achievement in itself. Role modeling for framework design achieves the following results:

- *Novel role modeling concepts.* This dissertation introduces role constraints as a new concept to more precisely define role models. Role constraints let developers specify how roles may or may not come together in an object. Earlier role modeling approaches offer no such description mechanism.
- *Integration of role modeling with class-based modeling.* Earlier role modeling approaches, in particular [Ree96], view role modeling and class-based modeling as different paradigms. Andersen, following up on Reenskaug, brings classes back into the picture, but still views them as largely unrelated to roles [And97].

This dissertation shows how role modeling can be integrated with traditional class-based modeling so that the respective strengths are added and the weaknesses are dropped.

  - The dissertation revises the existing concepts of role, role type, role model, class, and class model to better fit together, and is precise about the distinction between type level (role type, class, role model, class model) and instance level (role, object collaboration task, object collaboration).
  - The dissertation demonstrates the complementary focus of classes and role models. A role model describes how objects collaborate for one specific object collaboration task. A class defines how several roles from different collaboration tasks come together in one object, thereby defining how to bridge and integrate the tasks.

- *Introduction of an explicit framework concept.* The modeling method gives a precise definition of what a framework is and what its properties are. The definition of the framework concept is based on the notions of free role model, built-on class set, and extension point classes.
  - Using free role models, developers specify how a framework is to be used by clients.
  - Using built-on class sets and free role models, developers specify how a framework builds on other frameworks and how it depends on its environment. While software engineering has long understood that next to provided functionality also required functionality needs to be specified [Wir82, PN86], facilities to do so in framework design have been missing.
  - Using extension-point class sets, developers can specify how a framework may be extended.

  These concepts are unique to frameworks and let developers speak about and deal with frameworks in a framework-specific way. A framework becomes an explicit design artifact rather than just another class model.

In addition, role modeling for framework design provides excellent means for describing design patterns and for showing how they are used in the context of object-oriented frameworks.

- *Description of design patterns.* Role modeling as defined in this dissertation is a more general way of illustrating design patterns. A role model illustration of a pattern can be applied in more ways than a class-based illustration. A role model illustration adds to a class-based illustration, because a class-based illustration typically suggests too rigid a structure of pattern application.
- *Application of patterns in framework design.* Because role models are prepared for composition right from the start, they show well how pattern applications compose and overlap in framework design. Class-based modeling offers no such facilities. Annotating classes as participants of a pattern instantiation goes into the right direction but stops halfway. Role modeling goes all the way.

These contributions are described in more detail and validated in the main body of the dissertation. Some of them have also been published at conferences and in journals [Rie96a, Rie97c, RG98, RBGM99].

10.2 Future work

This dissertation work opens several venues for future work, both with a narrow focus on role modeling, and a larger focus on frameworks and design patterns.

- *Choice of a type specification mechanism.* The dissertation suggests no specific type specification mechanism. Any mechanism that provides types, subtyping, and type composition suffices. How-

ever, some type specification mechanisms may be more convenient and more effective to use than others. Therefore, a mechanism custom-tailored to the needs of role modeling may be developed.

- *Introducing dynamic behavior specifications.* Because type specification issues are largely ignored in this dissertation, not much is said about dynamic behavior specifications. However, role models are best described not only by individual role types, but also by descriptions of the allowed collaborative behavior of objects acting according to the role types. Therefore, Reenskaug's or Andersen's mechanism to describe collaborative behavior of object roles may be adapted [Ree96, And97], or a new one may be developed.

- *Composition of role types independently of classes.* Role modeling for framework design as presented in this dissertation composes role types to derive classes. I have never found a real need to compose role types to become composite role types. However, it may be a nice-to-have feature that could be useful once it is available.

Future work of this kind may directly build on current type specification mechanisms. However, existing mechanisms need to be adapted to fit role modeling in such a way that the separation of concerns achieved by role modeling is maintained. The benefit of reduction in complexity that role modeling achieves is directly based on the separation of concerns it provides.

- *Inheritance relationship between role models (addressing the problem of covariant redefinition).* The dissertation does not introduce a concept of inheritance between role models. Any role model that could be viewed as a specialization of an existing role model is viewed as a different unrelated role model. It seems helpful in many situations, however, to view one role model as a specialization of another more general role model. For example a simple PersonModel/PersonView role model might be specialized to form a CustomerModel/CustomerView role model.

  Inheritance between role models might give a new twist to the problem of covariant redefinition of operation signatures. The covariant redefinition of parameters of an operation in a subclass (and the contravariant redefinition of return value or object types) serves to ensure that class hierarchies are specialized in parallel. The covariant redefinition of operations of a class is always carried out with a particular partner class in mind. Thus, covariant redefinition is about ensuring constraints on a set of allowed collaboration tasks (rather than individual classes). Expressing these collaboration tasks is all what role models are about.

- *Extension of programming languages with role modeling concepts.* It would certainly be helpful to support the implementation of a role–model–based design with dedicated programming constructs. Such a role-oriented programming language might provide concepts for directly and conveniently expressing role types and role models.

  Current work already points into that direction. Van Hilst presents a role programming method using C++ templates [Van97], and Kendall uses aspect-oriented programming to more easily implement role–model–based designs [Ken99].

- *Support for design patterns and design templates.* Role modeling lets developers more easily apply and recognize design patterns in object-oriented designs (than is possible with traditional class-based designs). A dedicated design notation for specifying design templates for design patterns may be based on role modeling rather than class-based modeling [Rie96a]. Then, the application of a design pattern leads to a specific role model that can be easily composed with other role models in the context of an object-oriented design.

  On the pattern/template level, something alike to composite role types is going to be helpful, as illustrated by the Bureaucracy pattern [Rie98]. The Bureaucracy pattern is a composite pattern in which different role types from different design patterns are composed to form the pattern/template level equivalent of a composite role type.

- *Empirical assessment of use of design patterns in framework design.* The case studies of Chapters 6 to 8 provide some empirical data about the frequency of use of design patterns in framework de-

sign. Role models can be used as a kind of object-oriented function-point, that is, as an atomic unit of functionality in framework design. Thus, role models may serve as a coarse-grained measure for complexity in framework design.

An analysis of frameworks described using role modeling can provide us with a figure about the frequency of design pattern application in relation to the overall functionality (= total number of role models in a framework's design). Arriving at a statement like "60% of a framework's functionality can be described using design patterns" is a valuable result to justify further research into design patterns.

Finally, marrying component-based design with object-oriented frameworks is a whole new research area. It does not follow directly from this dissertation, but it should take the new understanding of frameworks gained through this dissertation into account. Then, role modeling is likely to find its way into component design and implementation. Role modeling might even be reintroduced on a component framework level to better describe how components collaborate.

10.3 Final conclusions

Role modeling provides separation of concerns in a way that is highly beneficial to class-based design of frameworks. This dissertation shows how to marry role modeling with class-based design of frameworks. Role modeling for framework design has the following properties:

- It reduces complexity of classes in framework design.
- It reduces complexity of object collaboration in framework design.
- It better supports specifying requirements put upon use-clients of a framework.
- It lets developers apply and recognize design patterns in frameworks more easily.
- It provides new and revised role modeling concepts for more precise framework design.
- It makes frameworks first class citizens of software architecture.

Role modeling for framework design combines the strengths of role modeling with those of class-based modeling while leaving out their weaknesses. It is therefore an evolutionary extension of current methods that preserves existing investments. Finally, role modeling for framework design is the first comprehensive modeling method to make frameworks explicit design artifacts.

A

References

ABGO93 A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. “An Object Data Model with Roles.” In *Proceedings of the 19th International Conference on Very Large Databases.* San Mateo, CA: Morgan Kaufmann, 1993. Page 39-51.

AC96 Martin Abadi and Luca Cardelli. “On Subtyping and Matching.” *ACM Transactions on Programming Languages and Systems* 18, 4 (July 1996). Page 401-423.

AG96 Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

All97 Robert J. Allen. *A Formal Approach to Software Architecture.* Ph.D. Thesis, CMU-CS-97-144. Pittsburgh, PA: Carnegie Mellon University, 1997.

And97 Egil P. Andersen. *Conceptual Modeling of Objects.* Ph.D. Thesis. Oslo, Norway: University of Oslo, 1997.

Bäu98 Dirk Bäumer. *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme.* Dissertation. Hamburg, Germany: Universität Hamburg, 1998.

BBE95 Andreas Birrer, Walter Bischofberger, and Thomas Eggenschwiler. “Wiederverwendung durch Framework-Technik-Vom Mythos zur Realität”. *OBJEKTSpektrum* 5 (1995). Page 18-26.

BC98 Kent Beck and Ward Cunningham. “A Laboratory for Teaching Object-Oriented Thinking.” In *Proceedings of the 1989 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA ’89). ACM Press, 1989. Page 1-6.

BCC+96 Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch, and John Vlissides. “Industrial Experience with Design Patterns.” In *Proceedings of the 18th International Conference on Software Engineering* (ICSE 18). IEEE Press, 1996. Page 103-114.

BCG95 William Berg, Marshall Cline, and Mike Girou. “Lessons Learned from the OS/400 OO Project.” *Communications of the ACM* 38, 10 (October 1995): 54-64.

Be97 Be, Inc. *Be Developer Guide.* O’ Reilly, 1997.

BGK+97 Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, and Heinz Züllighoven. “Framework Development for Large Systems.” *Communications of the ACM* 40, 10 (October 1997). Page 52-59.

BGR96a Walter Bischofberger, Michael Guttman and Dirk Riehle. “Architecture Support for Global Business Objects: Requirements and Solutions.” In *Joint Proceedings of the SIGSOFT ’96 Workshops* (ISAW-2). Edited by Laura Vidal, Anthony Finkelstein, George Spanoudakis, and Alexander L. Wolf. ACM Press, 1996. Page 143-146.

BGR96b Walter Bischofberger, Michael Guttman and Dirk Riehle. “Global Business Objects: Requirements and Solutions.” In *Proceedings of the Ubilab Conference '96, Zürich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Germany: Universitätsverlag, 1996. Page 79-98.

BHH+97 Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. “Towards a Formalization of the Unified Modeling Language.” In *Proceedings of the 1997 European Conference on Object-Oriented Programming* (ECOOP ’97). Springer Verlag, 1997. Page 344-366.

BHKS97 Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. “A Graphical Description Technique for Communication in Software Architectures.” In *Software Architectures and Design Patterns in Business Applications.* Edited by Manfred Broy, Ernst Denert, Klaus Renzel, and Monika Schmidt. Technical Report TUM-I9746. Munich, Germany: Technische Universität München, 1997.

BMR+96 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

Box98 Don Box. *Essential COM.* Addison-Wesley, 1998.

BR98 Dirk Bäumer and Dirk Riehle. “Product Trader.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 29-46.

BRSW00 Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. “Role Object.” In *Pattern Languages of Program Design 4.* Edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison-Wesley, 2000. Page 15-32. Originally published as: Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. “Role Object.” In *Proceedings of the 1997 Conference on Pattern Languages of Programming* (PLoP ’97). Washington University Department of Computer Science, Technical Report WUCS-97-34, 1997. Paper 2.1.

BRS+98 Dirk Bäumer, Dirk Riehle, Wolf Siberski, Carola Lilienthal, Daniel Megert, Karl-Heinz Sylla, and Heinz Züllighoven. *Values in Object Systems.* Ubilab Technical Report 98.10.1. Zurich, Switzerland: UBS AG, 1998.

CIM92 Roy H. Campbell, Nayeem Islam, and Peter Madany. “Choices, Frameworks and Refinement.” *Computing Systems* 5, 3 (Summer 1992): 217-257.

Cox87 Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1987.

CP95 Sean Cotter, with Mike Potel. *Inside Taligent Technology.* Addison-Wesley, 1995.

DH72 Ole-Johan Dahl and C. A. R. Hoare. “Hierarchical Program Structures.” In *Structured Programming*. Edited by Ole-Johan Dahl, Edsger W. Dijkstra and C. A. R. Hoare. Academic Press, 1972.

DW98 Desmond F. D' Souza and Alan C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley Longman, 1998.

EF97 Eric Evans and Martin Fowler. "Specifications." In *Proceedings of the 1997 Conference on Pattern Languages of Programming* (PLoP '97). Washington University Department of Computer Science, Technical Report WUCS-97-34, 1997. Paper 2.3.

FHG98 Donald Firesmith, Brian Henderson-Sellers, and Ian Graham. *OPEN Modeling Language.* Cambridge University Press, 1998.

FK97 Robert G. Fichman and Chris F. Kemerer. "Object Technology and Reuse: Lessons from Early Adopters." *Computer* 30, 10 (October 1997). Page 47-59.

FS97 Mohamed E. Fayad and Douglas C. Schmidt (editors). Special Issue on Object-Oriented Application Frameworks. *Communications of the ACM* 40, 10 (October 1997).

FSJ99 Mohamed Fayad, Douglas Schmidt, and Ralph Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* Wiley & Sons, 1999.

Gam98 Erich Gamma. "Advanced Design with Patterns and Java." Tutorial given at the *1998 European Conference on Java and Object Orientation.* Copenhagen, Denmark, 1998. See Appendix E for a pointer to the tutorial.

GHJV95 Erich Gamma. Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

GR89 Adele Goldberg and David Robson. *Smalltalk-80—The Language.* Addison-Wesley, 1989.

GSR96 Georg Gottlob, Michael Schrefl, and Brigitte Röck. "Extending Object-Oriented Systems with Roles." *ACM Transactions on Information Systems* 14, 3 (July 1996). Page 268-296.

Hal96 Terry Halpin. "Business Rules and Object Role Modeling." *Database Programming and Design* 9, 10. San Mateo, CA: Miller Freeman. Page 66-72.

Hal98 Terry Halpin. "UML Data Models from an ORM Perspective." *Journal of Conceptual Modeling* (http://www.inconcept.com/jcm), April 1998.

HHG90 Richard Helm, Ian M. Holland and Dipayan Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." In *Proceedings of the 1990 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '90). ACM Press, 1990. Page 169-180.

HO93 William Harrison and Harold Ossher. "Subject-Oriented Programming (A Critique of Pure Objects)." In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '93). ACM-Press, 1993. Page 411-428.

HS88 Daniel Hoffman and Richard T. Snodgrass, "Trace Specifications: Methodology and Models" *IEEE Transactions on Software Engineering* 14, 9 (September 1988). Page 1243-1252.

Hür94 Walter L. Hürsch. "Should Superclasses be Abstract?" In *Proceedings of the 1994 European Conference on Object-Oriented Programming* (ECOOP '94, LNCS 821). Edited by Mario Tokoro and Remo Pareschi. Springer-Verlag, 1994. Page 12-31.

JF88 Ralph E. Johnson and Brian Foote. "Designing Reusable Classes." *Journal of Object-Oriented Programming* 1, 2 (June/July 1988). Page 22-35.

Joh92 Ralph E. Johnson. "Documenting Frameworks using Patterns." In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '92). ACM Press, 1992. Page 63-70.

JW98 Ralph Johnson and Bobby Woolf. “Type Object.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 47-65.

Ken99 Elisabeth A. Kendall. “Role Model Designs and Implementations with Aspect Oriented Programming.” Unpublished manuscript.

KLM+97 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-Oriented Programming.” In *Proceedings of the 1997 European Conference on Object-Oriented Programming* (ECOOP ’97). Springer Verlag, 1997. Page 220-242.

KO96a Bent Bruun Kristensen and Kasper Osterbye. “Roles: Conceptual Abstraction Theory and Practical Language Issues.” *Theory and Practice of Object Systems* 2, 3 (1996). Page 143-160.

Lew95 Ted Lewis (editor). *Object-Oriented Application Frameworks.* Greenwich: Manning, 1995.

LH89 Karl J. Lieberherr and Ian M. Holland. “Assuring Good Style for Object-Oriented Programs.” *IEEE Software* 22, 9 (September 1989). Page 38-48.

Lie95 Karl J. Lieberherr. *Adaptive Object-Oriented Software.* Boston, MA: PWS Publishing Company, 1995.

LKA+95 David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. “Specification and Analysis of System Architecture Using Rapide.” *IEEE Transactions on Software Engineering* 21, 4 (April 1995). Page 336-355.

Lop97 Cristina Lopes. *D: A Language Framework for Distributed Programming.* Ph.D. Thesis. Boston, MA: Northeastern University, 1997.

LW93a Barbara Liskov and Jeannette Wing. “Specifications and Their Use in Defining Subtypes.” In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’93). ACM Press, 1993. Page 16-28.

LW93b Barbara Liskov and Jeannette Wing. “A New Definition of the Subtype Relation.” In *Proceedings of the 1993 European Conference on Object-Oriented Programming* (ECOOP ’93). LNCS-707. Springer-Verlag, 1993. Page 118-141.

LW94 Barbara H. Liskov and Jeannette M. Wing. “A Behavioral Notion of Subtyping.” *ACM Transactions on Programming Languages and Systems* 16, 6 (November 1994). Page 1811-1841.

Mac82 B. J. MacLennan. “Values and Objects in Programming Languages.” *ACM SIGPLAN Notices* 17, 12 (December 1982). Page 70-79.

McA95a Jeff McAffer. *A Meta-Level Architecture for Prototyping Object Systems.* Ph.D. Thesis. Tokyo, Japan: University of Tokyo, 1995.

McA95b Jeff McAffer. “Meta-level Programming with CodA.” In *Proceedings of the 1995 European Conference on Object-Oriented Programming* (ECOOP ’95). LNCS-952. Springer-Verlag, 1995. Page 190-214.

Mey91 Bertrand Meyer. “Design by Contract.” *Advances in Object-Oriented Software Engineering*. Edited by Dino Mandrioli und Bertrand Meyer. Prentice-Hall, 1991. Page 1-50.

Mey92 Bertrand Meyer. *Eiffel. The Language*. Prentice-Hall, 1992.

Mey92b Bertrand Meyer. “Applying Design By Contract.” *IEEE Computer* 25, 10 (October 1992). Page 40-51.

MDEK95 Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. “Specifying Distributed Software Architectures.” In *Proceedings of 5th European Software Engineering Conference* (ESEC ’95). Springer-Verlag, 1995.

Ode98 James J. Odell. *Advanced Object-Oriented Analysis and Design Using UML*. Cambridge University Press, 1998.

OKH+95 Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. “Subject-Oriented Composition Rules.” In *Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’95). ACM Press, 1995. Page 235-250.

PN86 Ruben Prieto-Diaz and James M. Neighbors. "Module Interconnection Languages." Journal of Systems and Software 6, 4 (November 1986): 307-334.

RAB+92 Trygve Reenskaug, Egil P. Andersen, Arne Jorgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Nêss-Ulseth, Gro Oftedal, Anne Lise Skaar, and Pâl Stenslet. “OORASS: seamless support for the creation and maintenance of object-oriented systems.” *Journal of Object-Oriented Programming* 5, 6 (October 1992). Page 27-41.

RBGM99 Dirk Riehle, Roger Brudermann, Thomas Gross, and Kai-Uwe Mätzel. “Pattern Density and Role Modeling of an Object Transport Service.” *ACM Computing Surveys* 31, 2 (June 1999). To appear.

RD99a Dirk Riehle and Erica Dubach. “Working with Java Interfaces and Classes. Part 1.” *Java Report* 4, 7 (July 1999). Page 35pp.

RD99b Dirk Riehle and Erica Dubach. “Working with Java Interfaces and Classes. Part 2.” *Java Report* 4, 10 (October 1999). Page 34pp

Ree96 Trygve Reenskaug, with Per Wold and Odd Arild Lehne. *Working with Objects*. Greenwich: Manning, 1996.

Rie96a Dirk Riehle. “Describing and Composing Patterns Using Role Diagrams.” In *Proceedings of the 1996 Ubilab Conference, Zürich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Germany: Universitätsverlag Konstanz, 1996. Page 137-152. Originally published in *Proceedings of the 1st International Conference on Object-Orientation in Russia* (WOON ’96). Edited by Alexander V. Smolyaninov and Alexei S. Shestialtynov. St. Petersburg, Russia: Electrotechnical University, 1996. Page 169-178. See Appendix E for a pointer to this publication.

Rie96c Dirk Riehle. “Patterns for Encapsulating Class Trees.” In *Pattern Languages of Program Design 2.* Edited by John M. Vlissides, James O. Coplien and Norman L. Kerth. Addison-Wesley, 1996. Page 87-104.

Rie97a Dirk Riehle. *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose.* Ubilab Technical Report 97.1.1. Zürich, Switzerland: Union Bank of Switzerland, 1997.

Rie97c Dirk Riehle. “Composite Design Patterns.” In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’97). ACM Press, 1997. Page 218-228.

Rie97d Dirk Riehle. “Arbeiten mit Java-Schnittstellen und –Klassen (Teil 1 von 2).” *Java Spektrum* 5/97 (September/October 1997). Seite 26-33.

Rie97e Dirk Riehle. “Arbeiten mit Java-Schnittstellen und –Klassen (Teil 2 von 2).”. *Java Spektrum* 6/97 (November/Dezember 1997). Seite 35-43.

Rie98 Dirk Riehle. “Bureaucracy.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 163-186.

Rit97 Antonio Rito Silva. “Framework, Design Patterns, and Pattern Language for Object Concurrency.” In *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA ’97).

RS95 Dirk Riehle and Martin Schnyder. *Design and Implementation of a Smalltalk Framework for the Tools and Materials Metaphor.* Ubilab Technical Report 95.7.1. Zürich, Switzerland: Union Bank of Switzerland, 1995.

RSB+98 Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. “Serializer.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 293-312.

RZ95 Dirk Riehle and Heinz Züllighoven. “A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor.” In *Pattern Languages of Program Design.* Edited by James O. Coplien and Douglas C. Schmidt. Addison-Wesley, 1995. Page 9-42.

RZ96 Dirk Riehle and Heinz Züllighoven. “Understanding and Using Patterns in Software Development.” *Theory and Practice of Object Systems* 2, 1 (1996). Page 3-13.

Sch98 Bruno Schäffer. *Design and Implementation of Smalltalk Mixin Classes.* Ubilab Technical Report 98.11.1. Zurich, Switzerland: UBS AG, 1998.

SBF96 Steve Sparks, Kevin Benner, and Chris Faris. “Managing Object-Oriented Framework Reuse.” *Computer* 29, 9 (September 1996): 52-61.

SG96 Mary Shaw and David Garlan. *Software Architecture—Perspectives on an Emerging Discipline.* New Jersey: Prentice Hall, 1996.

Sie96 Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley & Sons, 1996.

Som98 Peter Sommerlad. “Manager.” In *Pattern Languages of Program Design 3.* Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 19-28.

Str94 Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

Sun96a James Gosling, Frank Yellin, and the Java Team. *The Java Application Programming Interface, Volume 1.* Addison-Wesley, 1996.

Sun96b James Gosling, Frank Yellin, and the Java Team. *The Java Application Programming Interface, Volume 2.* Addison-Wesley, 1996.

Szy98 Clemens Szyperski. *Component Software-Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

Tal95 Taligent Inc. *The Power of Frameworks.* Addison-Wesley, 1995.

UML97a Rational Software Corporation et al. *UML v1.1 Semantics.* Santa Clara, CA: Rational Software Corporation, 1997.

UML97b Rational Software Corporation. *UML v1.1 Notation Guide.* Santa Clara, CA: Rational Software Corporation, 1997.

Van97 Michael VanHilst. *Role Oriented Programming for Software Evolution.* Ph.D. Thesis. Seattle, WA: University of Washington, 1997.

Vli98 John Vlissides. *Pattern Hatching.* Addison-Wesley Longman, 1998.

VN96 Michael VanHilst and David Notkin. “Using Role Components to Implement Collaboration-Based Designs.” In *Proceedings of the 1996 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA ’96). ACM Press, 1996. Page 359-369.

WG95 André Weinand and Erich Gamma. “ET++ A Portable Homogeneous Class Library and Application Framework.” In *Object-Oriented Application Frameworks.* Edited by Ted Lewis. Greenwich: Manning, 1995. Page 154-194.

WGM89 André Weinand, Erich Gamma, and Rudolf Marty. “Design and Implementation of ET++, a Seamless Object-Oriented Application Framework.” *Structured Programming* 10, 2 (Juni 1989): Page 63-87.

Wir82 Wirth, N. (1982). Programming in Modula-2. Berlin, Heidelberg: Springer-Verlag.

WJS95 Roel Wieringa, Wiebren de Jonge, and Paul Spruit. “Using Dynamic Classes and Role Classes to Model Object Migration.” *Theory and Practice of Object Systems* 1, 1 (1995) Page 61-83.

Woo98 Bobby Woolf. “Null Object.” In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 5-18.

WSP+92 Peter Wegner, William Scherlis, James Purtilo, David Luckham and Ralph Johnson. “Object-Oriented Megaprogramming.” In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA ’92). ACM Press, 1992. Page 392-396.

WWW90 Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

WZ88 Peter Wegner and Stanley B. Zdonik. “Inheritance as an Incremental Modification Mechanism or What Like Is and Isn’t Like.” In *Proceedings of the 1988 European Conference on Object-Oriented Programming* (ECOOP ’88). LNCS 322. Springer-Verlag, 1998. Page 55-77.

B

Glossary

Class

A class is the definition of a (possibly infinite) set of objects, called its instances. A class defines a non-empty set of role types, a composition function, and a class type. The composition function, applied to all role types, results in the class type.

See Definition 3-16 and 3-2.

Class, built-on

A built-on class of a framework is the class of a built-on object. It is connected to the framework through one or more role models.

See Definition 4-8.

Class, extension

An extension class of a framework is a subclass of an extension-point class of a framework.

See Definition 4-13.

Class, extension-point

An extension-point class is a framework class that may be subclassed by framework-external classes.

See Definition 4-10.

Class, use-client

A use-client class of a framework is the class of a use-client object. It is connected to the framework through one or more role models.

See Definition 4-3.

Class set, built-on

The built-on class set of a framework is the set of all built-on classes of the framework.

See Definition 4-9.

Class set, extension-point

The extension-point class set of a framework is the set of all extension-point classes of the framework.

See Definition 4-11.

Class model

A class model is a set of classes and a set of role models. The classes relate to each other by inheritance and object relationship descriptions between role types. The class relationship graph must be non-partitioned.

See Definition 3-20 and 3-11.

Composition function

A composition function composes types. It is used as part of a class definition, where it composes the role types of a class to form the class type.

See page 37.

Framework

A framework is a class model, together with a free role type set, a built-on class set, and an extension-point class set.

See Definition 4-1.

Framework extension

A framework extension is a set of classes. Each class is either an extension class of the framework or a class that is transitively connected with at least one extension class through a role model.

See Definition 4-14.

Framework extension, domain-specific

A domain-specific framework extension is a framework extension that is not a framework, but that can be used by different applications in the same domain.

See Definition 4-15.

Framework extension, application-specific

An application-specific framework extension is a framework extension that is not a framework and that can be used by one specific application only.

See Definition 4-16.

Inheritance

An inheritance is a pair of classes (X, Y) such that any instance of class Y can be substituted in a context where an instance of class X is expected.

See Definition 3-9.

Object

An object is an opaque runtime entity of a system that provides state and operations to query and change that state. An object has a lifecycle: It is created, may change over time, and is possibly deleted. Objects can be identified unambiguously; identity is an intrinsic property of every object.

See Definition 3-1.

Object, built-on

A built-on object of a framework is a framework-external object that a framework object makes use of in an object collaboration task.

See Definition 4-7.

Object, use-client

A use-client object of a framework is a framework-external object that makes use of one or more framework objects in an object collaboration task.

See Definition 4-2.

Object aggregation

An object aggregation is a pair of objects (x, y), stating that an object x aggregates an object y as a part of it. To aggregate an object means to control it, not only to make use of it, but to determine its lifetime and accessibility as well.

See Definition 3-7.

Object aggregation description

An object aggregation description is a pair of types (X, Y) that determines possible runtime object aggregations. An aggregation between two objects (x, y) conforms to the aggregation description if x is of type X or a subtype of X, and if y is of type Y or a subtype of Y.

See Definition 3-8.

Object association

An object association is a pair of objects (x, y), stating that an object x holds a reference to another object y of which it may or may not make use.

See Definition 3-5.

Object association description

An object association description is a pair of types (X, Y) that determines possible runtime object associations. An association between two objects (x, y) conforms to the association description if x is of type X or a subtype of X, and if y is of type Y or a subtype of Y.

See Definition 3-6.

Object collaboration

An object collaboration is a set of objects that relate to each other by object relationships. An object collaboration is said to be valid if it conforms to a class model.

See Definition 3-10.

Object collaboration task

An object collaboration task is an object collaboration and a set of roles objects play in the collaboration. The object relationship graph must be non-partitioned.

See Definition 3-17.

Object system.

An object system is an object collaboration.

See page 31.

Role

A role is an observable behavioral aspect of an object.

See Definition 3-12.

Role constraint

A role constraint is a value from the set {role-implied, role-equivalent, role-prohibited, role-dontcare}. For every given pair of role types (R, S) from a role model one such value is defined.

See Definition 3-19.

Role-dontcare constraint

A role-dontcare value for a pair of role types (R, S) defines that an object playing a role r of role type R has no constraints with respect to another role s of role type S within the given collaboration task. The role s may or may not be available together.

See page 38.

Role-equivalent constraint

A role-equivalent value for a pair of role types (R, S) defines that an object playing a role r defined by role type R is always capable of playing a role s defined by role type S, and vice versa. That is, role r and role s imply each other. This relationship is symmetric and transitive.

See page 38.

Role-implied constraint

A role-implied value for a pair of role types (R, S) defines that an object playing a role r defined by role type R is always capable of playing a role s defined by role type S. That is, role r implies role s. This relationship is transitive.

See page 38.

Role-prohibited constraint

A role-prohibited value for a pair of role types (R, S) defines that an object playing role r defined by role type R may not play role s defined by role type S within a given collaboration task. That is, role r prohibits role s for the task. This relationship is symmetric and transitive.

See page 38.

Role model

A role model is a set of role types that relate to each other by object relationship descriptions and role constraints. The role type relationship graph must be non-partitioned.

See Definition 3-18.

Role model, free

A free role model of a framework is a framework-defined role model that has one or more free role types.

See Definition 4-5.

Role type

A role type is a type that defines the behavior of a role an object may play. It defines the operations and the state model of the role, as well as the associated semantics.

See Definition 3-13.

Role type, callback

A callback role type is a free role type of a framework that has a non-empty set of operations. It may be picked up by higher-layer classes. Callback role types are the role modeling equivalent of callback interfaces as used by traditional coupling mechanisms.

See Definition 4-12.

Role type, free

A free role type of a framework is a role type of a framework-defined role model that may be picked up by use-client classes by putting it into their role type sets.

See Definition 4-4.

Role type, no-operation

A no-operation role type is a role type that defines no operations.

See Definition 3-14.

Role type, no-semantics

A no-semantics role type is a no-operation role type that defines neither state nor behavior.

See Definition 3-15.

Role type set, free

The free role type set of a framework is the set of all free role types of a framework.

See Definition 4-6.

Value

A value is an atomic entity from the abstract and invisible universe of values. A value cannot be perceived directly, but only through occurrences of its representations. The representations are interpreted by means of interpretation functions. These interpretation functions return further (occurrences of representations of) values; they do not change the value.

See Definition 3-3.

Value type

A value type is a type that specifies a set of values together with the interpretation functions applicable to representations of members of this set.

See Definition 3-4.

C

Notation Guide

This appendix describes the graphical notation used in the diagrams of this dissertation. The notation uses diagrammatic UML syntax where possible, and adds to it where necessary.

C.1 Classes and role types

A class is depicted as a rectangle, with the class name set in bold font at the top of the rectangle. If the class name is set in Italics, the class is abstract. Class and class type is used synonymously.

A role type is depicted as an oval. The name of the role type is prominently centered in the oval. Set below it, in parentheses and a smaller font, is the name of the role model the role type is defined by.

Figure C-1 depicts an example class and several example role types. Here, a class ResourceService provides the three role types Service, Provider, and Singleton of the role models ResourceService and RSSingleton, respectively. Thus, the qualified name of the role types is ResourceService.Service, RSSingleton.Provider, and RSSingleton.Singleton.

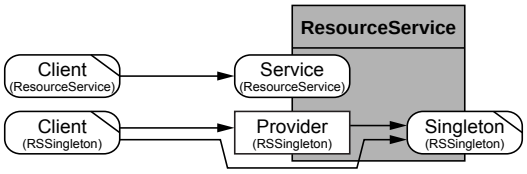

Figure C-1: Example class with role types.

Figure C-1 lets us distinguish between instance-level role types and class-level role types. An instance-level role type (a “regular” role type) is depicted as the aforementioned oval. An instance-level role type defines behavior that *instances* of a class conform to. A class-level role type is depicted as a rectangle (RSSingleton.Provider in Figure D-1). A class-level role type defines behavior that the *class object* representing the class conforms to.

Finally, Figure D-1 lets us distinguish between role types that have operations and role types that have no operations (still, expected behavior may be defined for these no-operation role types). A no-op role type is depicted with a gray mark in the upper right corner. Examples are the ResourceService.Client, the RSSingleton.Client, and the RSSingleton.Singleton role types. This property applies to instance-level and class-level role types alike.

Classes may relate to each other using class inheritance. Figure C-2 shows how the ResourceService class inherits from an abstract Service class (is a subclass of it).

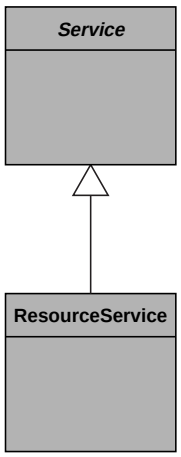

Figure C-2: Class inheritance.

The arrow inheritance symbol in Figure D-2 is taken from UML. See Chapter 3 for a discussion of the semantics of class inheritance in the context of role modeling.

C.2 Object relationships

Object relationship descriptions are relationships between types that constrain how objects conforming to these types may reference each other at runtime. Object relationship descriptions only relate the same kind of types with each other: class types with class types and role types with role types. Also, when connecting role types, object relationship descriptions are constrained to connect only role types of the same the role model.

Object relationship descriptions are depicted as connections between types. All relationship descriptions can be annotated with further information like direction and cardinality. Their meaning is taken from UML.

Figure C-3 shows three object relationship descriptions between role types as they may occur in a role model.

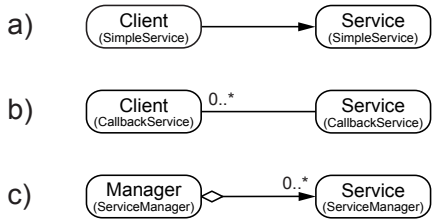

Figure C-3: Object relationships in role models

These three cases have the following meaning:

a) The arrow between the Client role type and the Service role type depicts a unidirectional use-relationship between an object playing the Client role and an object playing the Service role.

b) The line between the Client role type and the Service role type depicts a bi-directional use-relationship between objects, each of which plays one of the roles.

c) The diamond at the start of the arrow indicates ownership of the object pointed at, and the star at the end of the arrow indicates a cardinality of many (following UML rules).

The scoping of the object relationship descriptions by a role model or class model is important. No role type may relate to a role type from another role model by an object relationship description. See Chapter 3 for a discussion of how classes and role types relate to each other with respect to object relationship descriptions.

C.3 Class and role models

A class model is a set of interrelated classes, tied together by class inheritance and object relationship descriptions. A role model is a set of role types, tied together by role constraints (see below) and object relationship descriptions.

Figure C-4 shows three example role models, one binary, two ternary. The role model name is set below the role type name. The role model name qualifies the role type name so that role types with the same name, for example the three Client role types, can be distinguished.

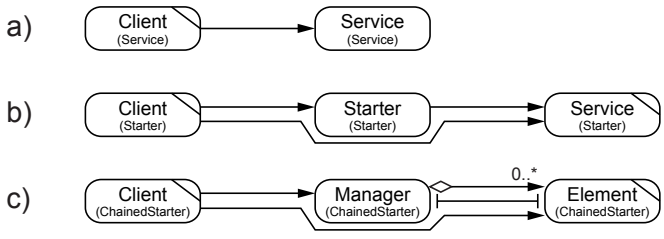

Figure C-4: Three example role models.

Figure C-5 shows one class model, consisting of three classes, and three example role models (from above).

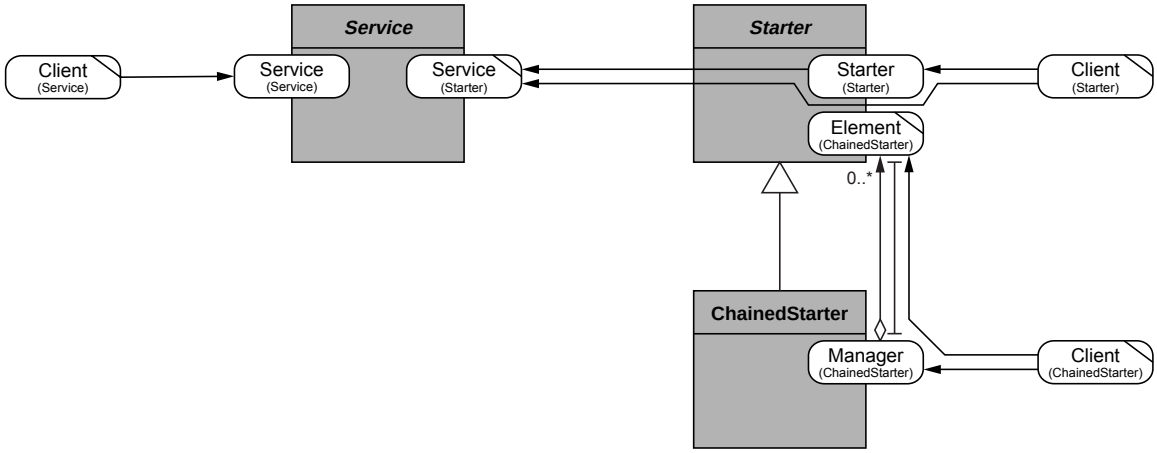

Figure C-5: Example class model.

The example class model does not show its free role types. Free role types are color-coded with a gray background (see the section on frameworks below).

C.4 Role constraints

Role constraints are values that define how roles played according to two role types from the same role model may come together in one object. For any given pair of role types, there is one role constraint value. The default value is role-dontcare (see below).

Role constraints are depicted as connections between role types. Figure C-6 shows the four different role constraints that may occur in a role model.

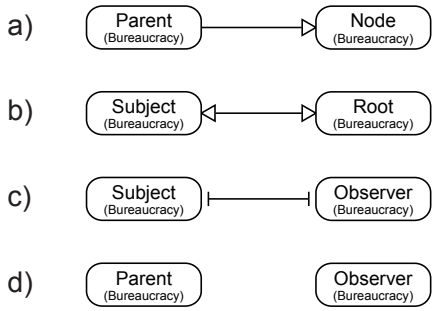

Figure C-6: Role constraints in a role model.

These four cases have the following meaning:

a) *Role-implied.* The white-headed unidirectional arrow depicts a role-implied role constraint. The role constraint value of (Parent, Node) is role-implied, the role relationship value of (Node, Parent) is role-dontcare.

b) *Role-equivalent.* The white-headed bi-directional arrow depicts a role-equivalent role constraint. The role constraint value of both (Subject, Root) and (Root, Subject) is role-equivalent.

c) *Role-prohibited.* The bar-headed bi-directional arrow depicts a role-prohibited role constraint. The role constraint value of both (Child, Root) and (Root, Child) is role-prohibited.

d) *Role-dontcare.* The missing of a symbol depicts a role-dontcare role constraint. The role constraint value of both (Client, RootClient) and (RootClient, Client) is role-dontcare.

Role constraints connect role types within one role model. A role-prohibited constraint is scoped by the runtime object collaboration task. It is possible for an object to play roles according to two different role types (even if there is a role-prohibited constraint between them) given that these roles are played in different role model instances, that is different object collaboration tasks.

C.5 Role model shorthands

Some role model compositions keep recurring throughout the examples. The primary examples are Object Creation and Singleton Access. Therefore, the dissertation uses shorthands to conveniently represent these recurring compositions as one single role model. Such a shorthand is much like a template in that it needs to be applied and adapted to a specific situation.

Figure C-7 shows an example of the applied Object Creation shorthand, here for the creation of a ResourceService instance.

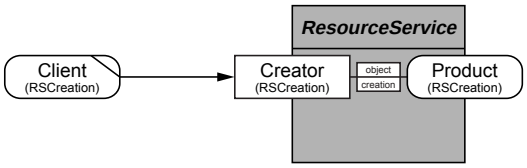

Figure C-7: Object Creation shorthand, applied to the ResourceService example.

Figure C-8 shows the expanded form of the Object Creation shorthand.

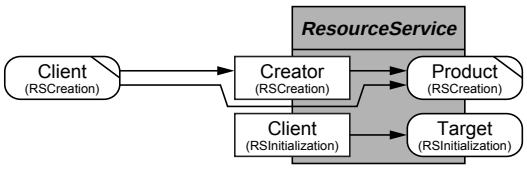

Figure C-8: Expanded form of the applied Object Creation shorthand.

Figure C-9 shows an example of the Singleton Access shorthand, here for the access to a single ResourceService instance.

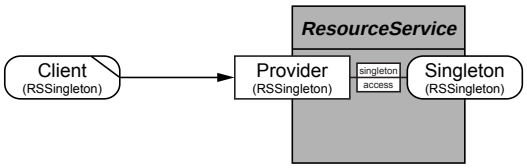

Figure C-9: Singleton Access shorthand, applied to the ResourceService example.

Figure C-10 shows the expanded form of the Singleton Access shorthand.

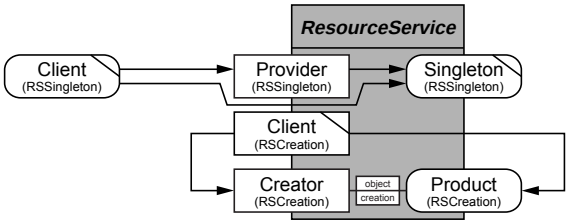

Figure C-10: Expanded form of the Singleton Access shorthand.

Another candidate for a shorthand is the composition of a Manager role model, in which a Manager object maintains a dictionary of Elements, with the lookup functionality of a dictionary and its key objects.

Composite (compound) pattern applications [Rie97c] are not a suitable subjects for shorthands, because it is important to see the constituting role models. Shorthands should be used only for trivial role model compositions.

If non-trivial role model compositions are needed frequently and lead to cluttering up the figures, an explicit semantic construct for composing role models can be introduced and given a diagrammatic representation.

C.6 Frameworks

Frameworks are class models with well-defined boundaries. A visual border with the framework's name attached to it surrounds the framework classes. Also, free role types of the framework are color-coded in gray.

Figure C-11 shows a simple framework based on the Service and Starter class model from above.

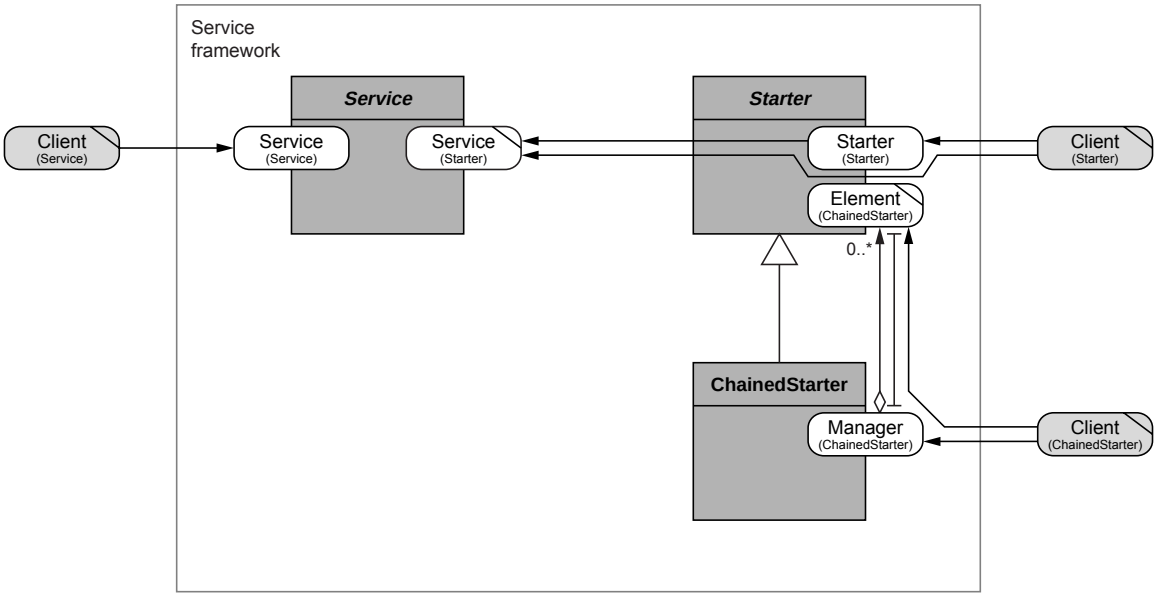

Figure C-11: Example framework.

The framework comprises the Service, Starter, and ChainedStarter classes. It offers three free role types for client classes to pick up: Service.Client, Starter.Client, and ChainedStarter.Client. The color-

coding of these free role types is maintained, even if they are assigned to classes in a client context, because they are still free for other clients to pick them up.

D

Design Patterns

This appendix describes several design patterns that are used in the main body of the dissertation using a role model form. The patterns are essentially the same as found in their original documentation, except that the role model form portrays them in a different light. The use of role models lets us more flexibly assign role types to classes than the class-based form allows us to do. While the class diagrams of the patterns frequently might look simpler than the role model diagrams, this is not the case. The complexity of a pattern is always the same, independently of its presentation form.

The pattern descriptions in this appendix have the following properties:

- The descriptions of the patterns are incomplete. They merely serve as a reminder for those who forgot or do not know a specific pattern under the given name so they can quickly look it up. For a more detailed description, references are provided.
- The patterns are illustrated rather than rigidly defined. There is no design pattern notation behind the pattern descriptions except than an intuitive understanding about what the role model illustration might communicate to developers regarding the pattern instantiation.
- None of the descriptions is to be taken as a rigid definition. (See the discussion of pattern vs. template in Chapter 3). Role models are more flexible than class model and suggest a wider application [Rie96a], but even they cannot encompass all possible design templates.

In the diagrams, some role types and object relationship descriptions are grayed out. These role types are required in an instantiation of a pattern, but are not considered to be an integral part of the core pattern role model illustration. The primary example of such a grayed-out role model is the role type pair (Client, Object) that simply states that an object provides some domain functionality to a Client.

D.1 Abstract Factory

The Abstract Factory pattern centralizes the creation of objects from a family of products in a Factory object. A Client requests new Product objects from a Factory object. The Factory object ensures consistency among a family of Product objects and hides the details of the object creation process.

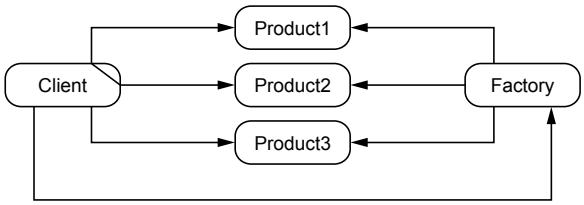

Figure D-1: Role model of the Abstract Factory pattern.

A class-based description of this pattern can be found in [GHJV95].

D.2 Adapter

The Adapter pattern adapts an existing object to a new use-context by means of an intermediate Adapter object. A Client uses the Adapter operations only, and the Adapter implements them in terms of the domain functionality of the Adaptee.

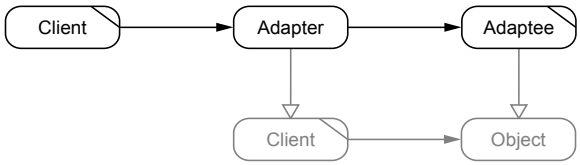

Figure D-2: Role model of the Adapter pattern.

A class-based description of this pattern can be found in [GHJV95].

D.3 Bridge

The Bridge pattern splits the implementation of a domain concept into an Abstraction and an Implementor object so that both can be varied independently. The Abstraction provides the primary domain functionality, and the Implementor provides the implementation primitives all variations of the Abstraction can be implemented by. The Client makes use only of the Abstraction. The Abstraction owns its Implementor, uses its operations for its own implementation, and hides it from the Client.

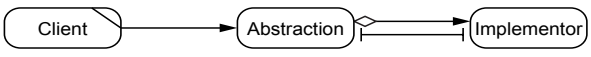

Figure D-3: Role model of the Bridge pattern.

A separate Client (not shown in the figure) configures the Abstraction with its Implementor. A common type of Client is an Abstract Factory that returns a preconfigured Bridge upon Client request.

A class-based description of this pattern can be found in [GHJV95].

D.4 Chain of Responsibility

The Chain of Responsibility pattern determines the target object of a client request dynamically by passing the request along a chain of objects. Each object in the chain may decide whether to execute, drop, or pass on the request. A Predecessor forwards the request to its Successor.

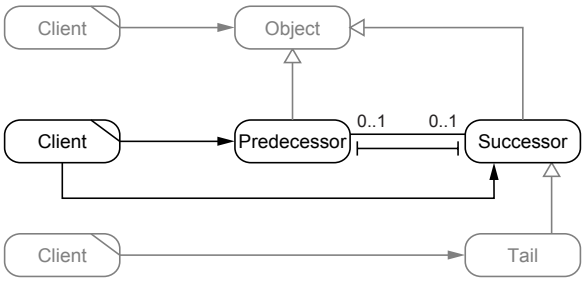

Figure D-4: Role model of the Chain of Responsibility pattern.

Using an object chain this way lets us configure the recipient of the request dynamically. A separate client configures the chain of objects.

A class-based description of this pattern can be found in [GHJV95].

D.5 Class Object

The Class Object pattern provides functionality common to all objects of a class in one Class object. A Class object can be asked for meta-information about any of its Instance objects. In contrast to a Type object, the Class object provides not only operations to inspect its Instances and provide information about it, but also functionality to create and change its Instances.

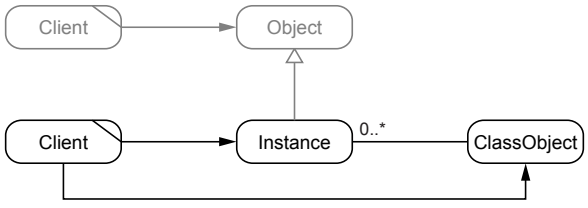

Figure D-5: Role model of the Class Object pattern.

Type Object is one element of the pattern triple Metaobject, Type Object, and Class Object. The distinction between Class Object and Type Object is done pragmatically. Class Objects provide implementation information about its Instances and they can manipulate and create Instances. Type Objects

provide application domain specific information rather than implementation information; their implementations may be heterogeneous, and they cannot manipulate their Instances.

To my knowledge, there is no commonly known class-based description of the pattern. However, any major object-oriented system provides an implementation of this pattern.

D.6 Composite

The Composite pattern determines how to build a hierarchy of objects. Any object in the hierarchy is a Child, or a Parent, or both. A Child may receive a Parent object, and a Parent object may receive or drop some Child objects. The Child and Parent role types serve to configure and maintain the hierarchy. A Client configures a Parent with its Child objects.

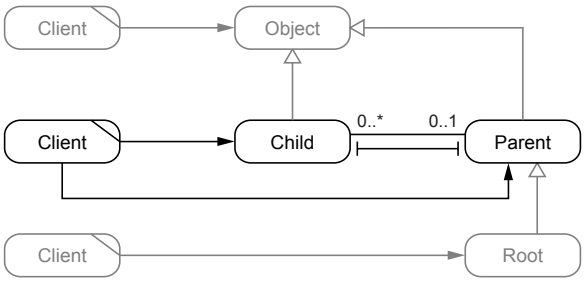

Figure D-6: Role model of the Composite pattern.

A class-based description of this pattern can be found in [GHJV95].

D.7 Decorator

The Decorator pattern lets us transparently add functionality to an existing object through object composition. A Core object is wrapped by a Decorator object. The Client makes use of both the Core and the Decorator without seeing to different objects.

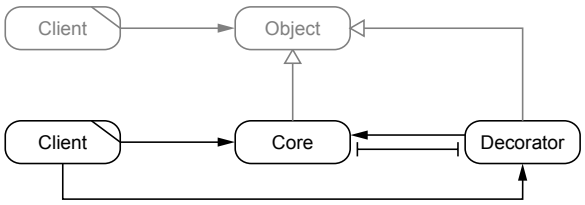

Figure D-7: Role model of the Decorator pattern.

A class-based description of this pattern can be found in [GHJV95].

D.8 Factory Method

The Factory Method pattern puts the creation of an object in method of its own that can be varied independently from the Client using the method. The Factory Method is provided by a Creator object that returns a Product object upon Client request.

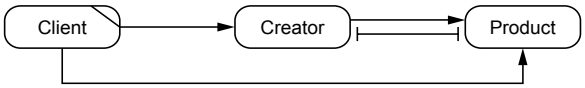

Figure D-8: Role model of the Factory Method pattern.

A class-based description of this pattern can be found in [GHJV95].

D.9 Manager

The Manager pattern puts the management of some Elements into a Manager object so that Element management gets independent of the Client and the Elements. Clients request Elements from the Manager. The Manager owns the Elements. It creates, provides, and deletes them.

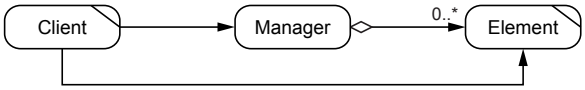

Figure D-9: Role model of the Manager pattern.

In [Som98], Sommerlad describes a class-based variant of this pattern, also called Manager. Sommerlad's Manager requires the set of Elements to be homogeneous, while the definition of Manager here accepts a heterogeneous collection of Elements.

D.10 Mediator

The Mediator pattern centralizes the communication of a set of Colleague objects in one Mediator object. The Colleagues do not communicate with each other directly, but only through the Mediator. This reduces communication complexity from square(n) to n. It facilitates the introduction and removal of a new Colleague object without affecting the other Colleagues.

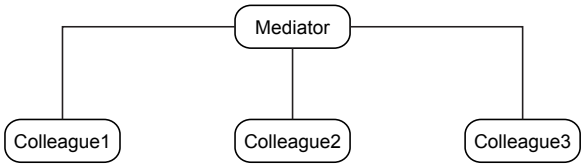

Figure D-10: Role model of the Mediator pattern.

A class-based description of this pattern can be found in [GHJV95].

D.11 Metaobject

The Metaobject pattern separates the domain-specific functionality of an object from the technical procedure of executing this domain functionality. Clients send requests to the Metaobject for execution on a specific BaseObject. The Metaobject defines the procedures for executing incoming requests, and the BaseObject provides the functionality to invoke the domain-specific operations. The Metaobject typically deals with issues like request queuing and synchronization, and the BaseObject provides a dynamic invocation interface for calling the domain-specific operations.

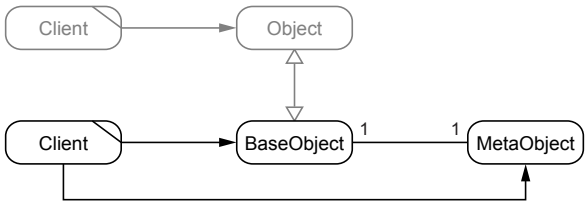

Figure D-11: Role model of the Metaobject pattern.

The configuration of the Metaobject can be complex (see Chapter 6 for an example). Metaobjects are part of a metalevel architecture that represents the overall (conceptual and/or technical) framework for handling metalevel issues.

Metaobject is one of the pattern triple Metaobject, Type Object, and Class Object.

To my knowledge, there is no commonly known class-based description of the pattern. However, every object-oriented system based on an explicit metalevel architecture is likely to feature an instance of this pattern.

D.12 Null Object

The Null Object pattern serves to provide a null implementation of a domain concept. The null implementation provides null behavior, which is the behavior assumed to be executed if no object were present at all. The pattern lets developers set an object reference to the null object rather than to a null reference and avoids cluttering the client code with checks whether the reference is null or not.

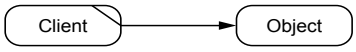

Figure D-12: Role model of the Null Object pattern.

In [Woo98], Woolf describes the Null Object pattern. Because it is only about implementation, the pattern cannot be expressed well using role modeling.

D.13 Object Registry

The Object Registry pattern centralizes access to a set of Element objects in one Registry object. Clients register and unregister Elements at the Registry giving them convenient names for later retrieval. Such a Registry is typically a thread-local or process-local Singleton.

Figure D-13: Role model of the Object Registry pattern.

The Object Registry pattern is to be distinguished from the Value Registry pattern (not documented here). An Object Registry handles objects, and a Value Registry handles values. Clients of an Object Registry have to be aware of possible side-effects, while clients of a Value Registry do not have to do so.

D.14 Observer

The Observer pattern decouples a set of Observers from a Subject. It is used to maintain state dependencies between the Observers and their Subject. In case of a state change, the Subject sends out an event to notify its Observers about the change. The Subject does not rely on any specific type of Observer, but uses a common and minimal Observer protocol only. A Client configures the Subject with its Observers (Client and Observer object may well be the same).

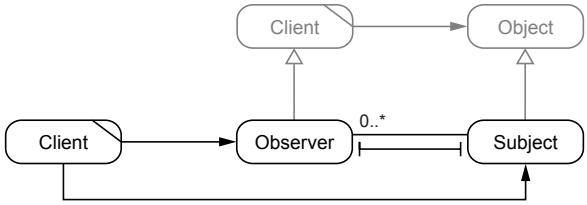

Figure D-14: Role model of the Observer pattern.

In Java, the Observer pattern takes on the form of EventListeners.

A class-based description of this pattern can be found in [GHJV95].

D.15 Product Trader

The Product Trader pattern separates the creation of a Product object from the Client requesting it by putting the creation process into a Trader object. Clients request new Products from the Trader using a specification of the Product (rather than naming its class). The Trader uses the specification to select an element from a set of Elements. Each of the Elements can act as a Creator for the Product.

When asked for a Product, the Trader selects an Element based on the specification provided by the Client. The Trader then acts as a Client of the Element that acts as the Creator for the new Product

object. The Trader/Client object asks the Element/Creator object for a new Product, which is then returned to the Client.

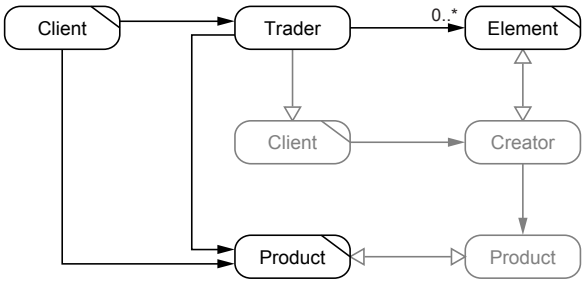

Figure D-15: Role model of the Product Trader pattern.

In [BR98], Bäumer and Riehle present a class-based description of this pattern.

D.16 Property List

The Property List pattern makes a Provider object provide a generic and extensible set of Properties to Clients. A Client asks a Provider object about its Properties. Properties are accessed using a naming scheme, for example simple strings, and generic get and set operations rather than through property-specific operations. The Provider defines which Properties it offers. Every implementation of a Provider may provide its own set of Properties. Properties may even be added and removed at runtime.

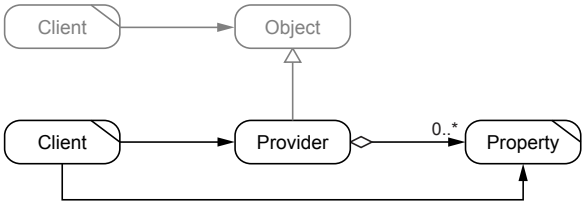

Figure D-16: Role model of the Property List pattern.

In [Rie97a], Riehle describes the pattern in more detail using role modeling.

D.17 Prototype

The Prototype pattern lets Clients create complex Product objects by cloning a Prototype object. Products are copies of the Prototype and reflect its possibly complex object configuration. Clients request a copy of the Prototype, which they receive as a new Product object. Prototypes serve as representatives of one or several complex objects and can be handled generically.

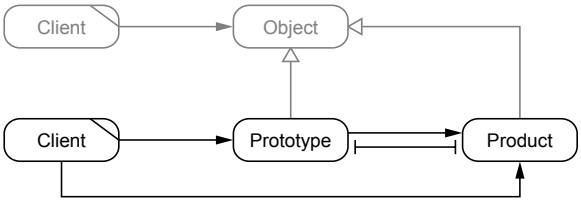

Figure D-17: Role model of the Prototype pattern.

A class-based description of this pattern can be found in [GHJV95].

D.18 Role Object

The Role Object pattern transparently attaches Role objects to a Core object. The Core represents an important domain concept, and the Roles represent some domain-specific extension of the Core concept. Clients make use both of the Roles and the Core. The Core manages its Roles and provides them to a Client upon request. Roles may be retrieved from the Core using a simple naming scheme, for example strings.

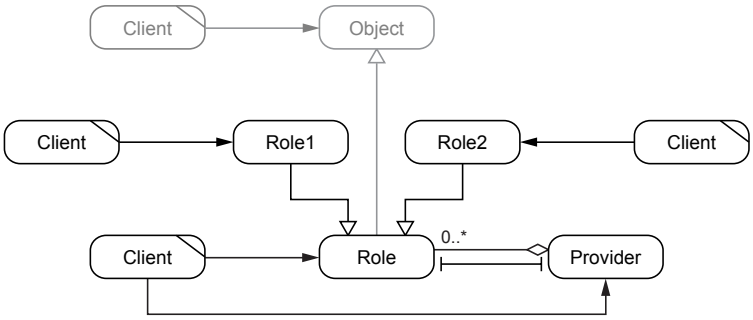

Figure D-18: Role model of the Role Object pattern.

The Core may be configured with Role objects in many different ways, for example during configuration time from configuration data or during execution time through dedicated clients.

In [BRSW00], Bäumer et al. present a class-based description of this pattern.

D.19 Type Object

The Type Object pattern centralizes common information about a set of Instance objects in a Type Object that is shared by all Instances. Clients ask the Type Object of an Instance for information about the Instance. This information may also be directly provided by an Instance, but it will be implemented then by asking the Type Object. Using Type Objects, otherwise redundant information about common properties of all Instances is provided in a single place, the Type Object.

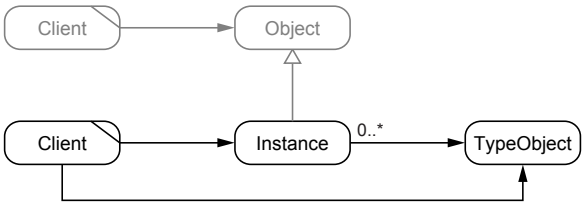

Figure D-19: Role model of the Type Object pattern.

Type Object is one of the pattern triple Metaobject, Type Object, and Class Object. The distinction between Class Object and Type Object is done pragmatically. Class Objects provide implementation information about its Instances and they can manipulate and create Instances. Type Objects provide application domain specific information rather than implementation information; their implementations may be heterogeneous, and they cannot manipulate their Instances.

In [JW98], Johnson and Woolf present a class-based description of this pattern.

D.20 Serializer

The Serializer pattern reads Readable objects from a Reader and writes Writable objects to a Writer. The Reader reads the object information from a specific backend, and the Writer writes the object information to a specific backend. Backends vary with Reader/Writer implementations. The Serializer pattern is used to serialize objects for different purposes like making them persistent, marshalling and unmarshalling them, and debugging them.

Readable and Reader as well as Writable and Writer objects collaborate recursively. A Readable reads all of its attributes from a Reader. For attributes that are Readable object references, the Reader to creates the Readable and then tells it to read its attributes from it, the Reader. Similarly, a Writable writes its attributes to a Writer that in turn tells a Writable attribute to write its attributs on it, the Writer. Primitive value types like integer and string attributes end the recursive descent.

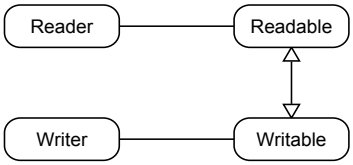

Figure D-20: Role model of the Serializer pattern.

The Serializer pattern can be viewed as the repeated specialized composition of the Visitor pattern.

In [RSB+98], Riehle et al. present a class-based description of this pattern.

D.21 Singleton

The Singleton pattern serves to ensure that there is exactly one instance of an object, the Singleton, in a given operation context, and to provide a central convenient access point to it. Historically, the op-

eration context is the process, but it could be a thread as well. A Client requests the Singleton from a Provider. If necessary, the Provider creates the Singleton on demand.

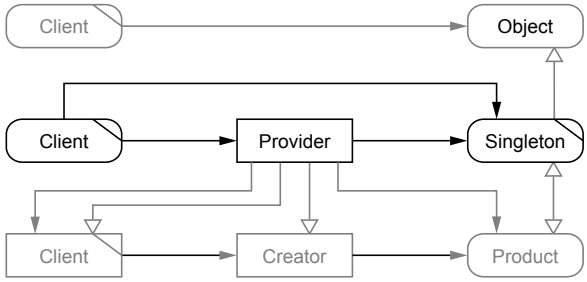

Figure D-21: Role model of the Singleton pattern.

Because the combination of an applied Singleton role model with an object creation role model of Client, Creator, and Product role types ocurrs frequently, the dissertation uses a shortcut for it.

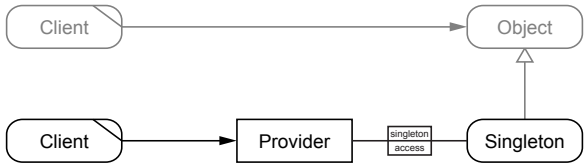

Figure D-22: Shortcut role model of the Singleton pattern.

A class-based description of this pattern can be found in [GHJV95].

D.22 Specification

The Specification pattern provides descriptive elements about an object to a client for use in object selection based on specifications. A Client requests a Specification from a Provider. The Specification describes one or several properties of the Provider. Typically, the Specification can provide a unique key to a client that is computed based on the properties described by the Specification and that distinguishes the Specification from Specifications of other Providers.

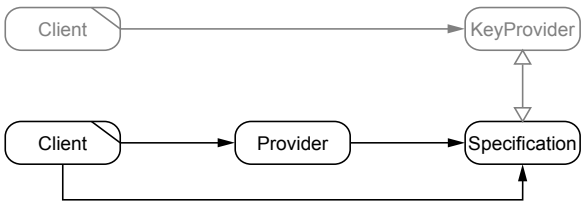

Figure D-23: Role model of the Specification pattern.

In [Rie96c], Riehle describes the use of the Specification pattern in the context of class selection and object creation (also known as trading), and in [EF97], Evans and Fowler describe several patterns that show how to define and compose complex specifications.

D.23 State

The State pattern serves to split a large state space of an object into several distinct parts to ease the object's implementation, to manage the state space more easily, and to extend it more easily. An Object providing domain functionality to a Client acts as the Context for a set of State objects. The Context forwards Client requests to one State object, which implements the requested behavior.

At any given time, exactly one State object is active, representing the subspace of the overall state space of the object the current state vector is in. If state changing operations cause the state vector to leave the current subspace, another State object becomes active, representing the correct subspace. A State object implements the behavior according to the rules of the subspace it represents.

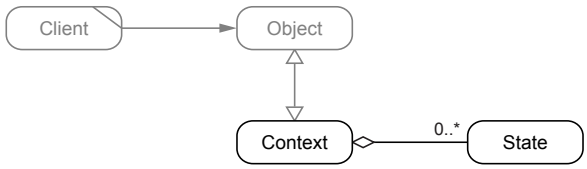

Figure D-24: Role model of the State pattern.

A class-based description of this pattern can be found in [GHJV95].

D.24 Strategy

The Strategy pattern serves to configure a domain object with an instance from a family of algorithms, rather than hard-coding any specific algorithm in the domain object. The Strategy object encapsulates the algorithm, and is set to its Context by a Client. Whenever the domain object has to execute the algorithm it acts as the Context of the Strategy and delegates the task of performing the algorithm to it.

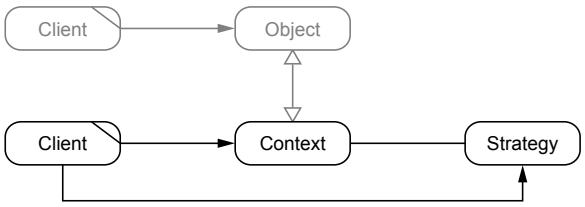

Figure D-25: Role model of the Strategy pattern.

A class-based description of this pattern can be found in [GHJV95].

D.25 Visitor

The Visitor pattern serves to extend an existing object structure with new external algorithms. The different object types from the structure are represented as different Node role types. Common to all objects is the Element role type. A Visitor object represents a new external algorithm. For each of its Node type attributes, the Element dispatches on a Visitor for the particular Node type. The Visitor can then execute the behavior associated with that particular Node type.

Because Node objects can always act as Elements, a Visitor may recursively descent into the object structure (if it is a hierarchy). An Element dispatches on an a Visitor for a given Node type attribute, and the Visitor calls on the Node type attribute to dispatch back to the Visitor on the Node type's attributes. Primitive value types and objects that are not Elements end the recursive descent.

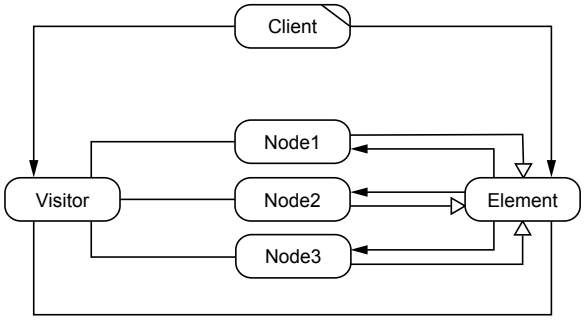

Figure D-26: Role model of the Visitor pattern.

A class-based description of this pattern can be found in [GHJV95].

E

Pointers to Further Material

This appendix provides pointers to information referred to in this work.

The following page provides the most recent pointers to all of the materials listed below:

- Index page: http://www.riehle.org/diss/index.html

At the time of writing this dissertation, the original information can be found at:

- JHotDraw 5.1: http://members.pingnet.ch/gamma/JHD-5.1.zip
- JHotDraw 5.1 tutorial: http://www.eos.dk/jaoo/presentations/index.html

An index to my own publications referenced in Appendix A can be found at:

- Publication index: http://www.riehle.org/papers/index.html

Curriculum Vitae

Name: Dirk Riehle
Nationality: xxxxxxxxx
Date of birth: xxxxxxxxx

xxxx-xxxx: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.
xxxx-xxxx: Matthias-Claudius-Gynmasium (high school), Hamburg, Germany.
1988-1991: University of Hamburg, Germany. Physics studies, Vordiplom.
1988-1995: University of Hamburg, Germany. Computer science studies, Dipl.-Inform.
1995-1996: UBS AG, Corporate Clients IT division, Zürich, Switzerland. Architect and developer.
1996-1999: UBS AG, Ubilab, Zurich, Switzerland. Researcher, architect, and developer.
1997-1999: ETH Zurich, Switzerland. Dissertation, supervised by Prof. Dr. Thomas R. Gross.
1999: Credit Suisse, Data Warehousing, Zurich, Switzerland. Software architect.
1999-current: SKYVA International, Cambridge, MA, and Mannheim, Germany. Software developer.