The Object Constraint Language

Getting your models ready for MDA

Jos B. Warmer
Anneke G. Kleppe

ADDISON-WESLEY

An Imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Library of Congress Cataloging-in-Publication Data

```
Warmer, Jos B.
   The object constraint language : getting your models ready for MDA
/ Jos B. Warmer, Anneke G. Kleppe.
      p.    cm.
    Includes bibliographical references (p.   ) and index.
    ISBN 0-201-37940-6 (alk. paper)
    1. Object-oriented methods (Computer science)  2. UML (Computer
  science)   I. Kleppe, Anneke G.  II. Title.
  QA76.9.O35W37  1999
  005.1'17--dc21                                            98-29492
                                                                 CIP
```

ISBN 0-201-37940-6
Text printed on recycled and acid-free paper.
1 2 3 4 5 6 7 8 9 10-MA-02010099
First printing, May 2003

Contents

List of Figures

List of Tables

Foreword to the first edition

For many years there has been a branch of computer science concerned with using formal logical languages to give precise and unambiguous descriptions of things. As an academic in the 1970s and 1980s I was very interested in such languages, for example Z and Larch. Unravelling the meaning of a statement in one of these languages is sometimes like a complex jigsaw puzzle, but once the unravelling is done the meaning is always crystal-clear and unambiguous. As I moved into the world of object-oriented methods I found a different way of specifying, using diagrams. With diagrams, the meaning is quite obvious, because once you understand how the basic elements of the diagram fit together, the meaning literally stares you in the face.

But there are many subtleties and nuances of meaning that diagrams cannot convey by themselves: uniqueness, derivation, limits, constraints, etc. So it occurred to me that from a modelling perspective a carefully designed combination of diagrammatic and formal languages would offer the best of both worlds. Armed with this realisation I worked during the early 1990s with John Daniels to create Syntropy, an object-oriented modelling language which combined the diagrammatic simplicity and clarity of OMT with the formality of a subset of Z.

Object Constraint Language (OCL) was first developed in 1995 during a business modelling project within IBM in which both Jos Warmer and I were involved, working in IBM's Object Technology Practice. This project was heavily influenced by Syntropy ideas. But unlike in Syntropy there is no use in OCL of unfamiliar mathematical symbols. OCL was very carefully designed to be both formal and simple: its syntax is very straightforward and can be picked up in a few minutes by anybody reasonably familiar with modelling or programming concepts.

During 1996 Jos and I became involved in the Object Management Group's efforts to standardize on a language for object-oriented analysis and design. I led IBM's contribution to this process, and together with ObjecTime Limited we wrote a proposal which emphasised simplicity and precision. Our goal all along was to collaborate with the other submitters to produce an overall standard containing the right elements in the right balance. OCL was a fundamental aspect of this proposal.

The leading proposal was UML (Unified Modeling Language) from Rational Software Corporation and its partners. UML, which combined ideas from its three authors Grady Booch, Ivar Jacobson, and James Rumbaugh, focused primarily on the diagrammatic elements and gave meaning to those elements through English text. As the submissions were combined, OCL was used to give added precision to the definition of UML; in addition OCL became part of the standard and thus available to modellers to express those additional nuances of meaning which just the diagrams cannot represent. This is a very important addition to the standard language of object-oriented modelling.

Jos Warmer and Anneke Kleppe's book is a crucial addition to the object-oriented modeling literature. I've greatly enjoyed working with Jos on the development of OCL over the past few years, and he and Anneke have done a first-class job in this book of explaining OCL and making it accessible to modellers. They have focused on the important aspects and illustrated the concepts with plenty of simple examples. Even for those with no experience of formal methods this book is an excellent place to learn how to add precision and detail to your models.

October 1998, Steve Cook

Steve Cook is the lead architect of IBM's European Object Technology Practice. He has been a pioneer of object-oriented methods and technologies for 20 years. He founded the OOPS specialist group of the British Computer Society in 1985, and the Object Technology series of conferences held in the UK annually since 1993. He was Managing Director of Object Designers Ltd from 1989 until 1994, when he joined IBM. In 1998 he was awarded an Honorary Doctor of Science degree by De Montford University.

Preface

To be done: This is more or less the old preface. It should be revised for the next edition.

In November 1997, the Object Management Group (OMG) set a standard for object-oriented analysis and design facilities. The standard, known as the Unified Modeling Language (UML), includes model diagrams, their semantics, and an interchange format between CASE tools. Within UML, the Object Constraint Language (OCL) is the standard for specifying invariants, preconditions, postconditions, and other kinds of constraints.

The only way we can gain anything from a standard is if everyone uses it. Therefore, any standard should be easy to use, easy to learn, and easy to understand. These objectives were our guidelines during the development of OCL.

OCL can be called a "formal" language, but unlike other currently available formal languages such as Objective-Z or VDM++, OCL is not designed for people who have a strong mathematical background. The users of OCL are the same people as the users of UML: software developers with an interest in object technology. OCL is designed for usability, although it is underpinned by mathematical set theory and logic.

Our objective in writing this book is to offer to practitioners of object technology a convenient way to become acquainted with and make use of this part of the UML standard. By writing this book we intend to make OCL available to everyone who can benefit from it. Using, learning, and communicating with OCL should be easy, and this book is an effort to make it easy.

With this book we emphasize the importance of constraints in object-oriented analysis and design and the importance of a formal, separate language for constraint notation. Please take OCL and use it well, so that the whole object-oriented community will gain from your efforts.

1.1 ACKNOWLEDGMENTS

Although on the cover of any book only the names of the authors appear, a book is always the result of the blood, sweat, and tears of many people. For

their efforts in reviewing this book we would like to thank Balbir Barn, Steve Cook, Wilfried van Hulzen, John Hogg, Jim Odell, and Cor Warmer. Special thanks go to Heidi Kuehn, who did a great job polishing our English.

Acknowledgments for their contributions to the first version of OCL must undoubtedly go to the following:

- The IBM team that developed the first version of OCL: Mark Skipper, Anna Karatza, Aldo Eisma, Steve Cook, and Jos Warmer.
- The joint submission team from IBM and ObjecTime. The ObjecTime team was composed of John Hogg, Bran Selic, and Garth Gullekson, and the IBM team consisted of Steve Cook, Dipayan Gangopadhyay, Mike Meier, Subrata Mitra, and Jos Warmer. On an individual basis, Marc Saaltink, Alan Wills, and Anneke Kleppe also contributed.
- The UML 1.1 team, especially the semi-formal subgroup of the UML core team: Guus Ramackers, Gunnar Overgaard, and Jos Warmer.
- Several people who influenced OCL during this period, most notably Desmond D'Souza, Alan Wills, Steve Cook, John Hogg, and James Rumbaugh.
- The many persons who gave their feedback on the earlier versions of OCL.

The following people are acknowledged for their contribution to the further development of OCL, that concluded in version 2.0 of the OMG standard:

- The members of the OCL 2.0 submission team: Tony Clark, Anders Ivner, Jonas Högström, Martin Gogolla, Mark Richters, Heinrich Hußmann, Steffen Zschaler, and Simon Johnston.
- The participants of the OCL workshops during the UML 2000, and UML 2001 conferences.
- People that made us aware of mistakes in the earlier edition of this book.

We would also like to thank all our teachers, colleagues, clients, and friends who in the past 15 years made us aware of the need for a practical form of formalism in software development. Coming from a theoretical background (mathematics and theoretical computer science), we have always found sound formalisms appealing, but very early in our careers we decided that writing a two-page "proof" for five lines of code is not the right way to improve our software. We have been searching ever since for a way to combine our love for sound and complete formalisms with our sense of practicality. We hope and expect that OCL will turn out to be just that: a practical formalism.

Anneke Kleppe and Jos Warmer

October 2002, Soest, Netherlands

Introduction

The *Object Constraint Language* (OCL) is a modeling language, with which you can build software models. It is defined as a standard "add-on" to the Unified Modeling Language (UML), the Object Management Group (OMG) standard for object-oriented analysis and design. Every expression written in OCL relies on the types, that is, classes, interfaces, etc., that are defined in the UML diagrams. The use of OCL therefore includes the use of at least some aspects of UML.

Expressions written in OCL add vital information to object-oriented models and other object modeling artifacts. This information often cannot be information expressed in a diagram. In UML 1.1, this information was thought to be limited to constraints, where a *constraint* is defined as a restriction on one or more values of (part of) an object-oriented model or system. In UML 2 the understanding is that there is far more additional information that should be included in a model than constraints alone. Defining queries, referencing values, or stating conditions and business rules in a model, is all done by writing expressions. OCL is the standard language in which these expressions can be written in a clear and unambiguous manner.

OCL has evolved from an expression language in the Syntropy method through a business modeling language used within IBM until it was included in the UML in 1997. At that point in time it received its current name. This name is currently well established and therefore it is not expedient to change it to, for instance, *Object Expression Language*, although this name would be more appropriate.

OCL has been used as an expression language for object-oriented modeling during the last six years. Since it was first conceived there have been many changes and additions to the language. Lately this has led to a new version of OCL, version 2.0, which is formally defined in the *Object Constraint Language Specification* [OCL2002].

Recently, the OMG has launched an initiative called the Model Driven Architecture (MDA). The essence of the MDA approach is that models are the basis for software development. To be able to work with this architecture good, solid, consistent, and coherent models are a neccesity. Using the combination of UML and OCL you are able to build such models.

In the many books that have been published on the subject of UML, its expression language has not received the attention it deserves. The aim of this book is to fill this gap and to explain UML's expression language, which supports the task of modeling object-oriented software as much as the UML diagrams.

WHO SHOULD READ THIS BOOK

The book is meant to be a textbook and reference manual for practitioners of object technology who find a need for more precise modeling. This includes persons that want to apply MDA principles. These people will want to use OCL in their analysis and design tasks, most probably within the context of UML but potentially with other graphical object modeling languages.

This book assumes that you have general knowledge of object-oriented modeling, preferably UML. If you lack this knowledge, there are many books on UML that you can choose from.

HOW THIS BOOK SHOULD BE USED

Part 1 of this book explains the Model Driven Architecture and the key role OCL plays in that framework. In this part OCL is explained in a relatively informal way, mostly by example.

Part 2 constitutes a reference guide that describes the OCL language completely.

Appendix ...

TYPEFACE CONVENTIONS

This book uses the following typeface conventions:

- All OCL expressions and context definitions are printed in a `monospaced font`.
- All OCL keywords are printed in a **`monospaced bold font`**.
- At the first introduction or definition of a term, the term is shown in *italics*.
- All references to classes, attributes, and other elements of a UML diagram are shown in *italics*.

INFORMATION ON RELATED SUBJECTS

The text of the UML standard, including OCL, is freely available on the OMG Web site (http://www.omg.org). More information on UML can be found in the following:

- *Unified Modeling Language User Guide*, by Grady Booch, James Rumbaugh, and Ivar Jacobson
- *Unified Modeling Language Reference Manual*, by James Rumbaugh, Grady Booch, and Ivar Jacobson
- *The Objectory Software Development Process*, by Ivar Jacobson, Grady Booch, and James Rumbaugh

OCL is still growing and maturing. Recent information on OCL, as well as a free parser, can be found on the Klasse Objecten Web site:

- http://www.klasse.nl/ocl
- http://www.object-constraint-language.info

Part 1

User Manual

Chapter 1

MDA and the Use of OCL

This chapter explains why it is important to create models that contain as much information about the system as possible, specially when working within within the Model Driven Architecture. Because the Model Driven Architecture itself is fairly new, a short introduction to this framework is given. Most importantly, this chapter states why OCL is a vital and neccesary element in the Model Driven Architecture.

1.1 MODEL DRIVEN ARCHITECTURE

The Model Driven Architecture (MDA) is gradually becoming an important aspect of software development. Many tools are, or least claim to be MDA compliant. But what exactly is the MDA?

The MDA is a framework being built under supervision of the Object Management Group (OMG) that defines how models defined in one language can be transformed into models in other languages. An example of a transformation is the generation of a database schema from a UML model, UML being the source language, and SQL being the target language. Source code is also considered to be a model. Code generation from a UML model is therefore another form of transformation.

This section gives an introduction to MDA, which is by no means complete. You can find more information on MDA on the OMG website and in a number of books, for example in *MDA Explained, Practice and Promise of the Model Driven Architecture,* by the same authors [Kleppe03].

1.1.1 PIMs and PSMs

Key to MDA is the importance of models in the software development process. Within MDA the software development process is driven by the activity of modeling your software system. The MDA process is divided into three steps:

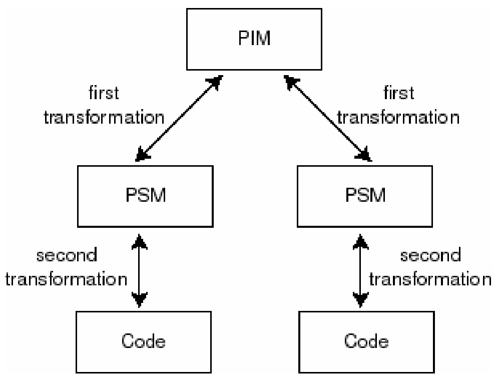

Figure 1-1 *The Relationship between PIM, PSM, and Code.*

1. Build a model with a high level of abstraction, that is independent of any implementation technology. This is called a Platform Independent Model (PIM).
2. Transform the PIM into one or more models that are tailored to specify your system in terms of the implementation constructs that are available in one specific implementation technology, e.g. a database model, an EJB model. These models are called Platform Specific Models (PSMs).
3. Transform the PSMs to code.

Because a PSM fits its technology very closely, the last transformation is straightforward. The complex step is the one in which a PIM is transformed to a PSM. The relationship between PIM, PSM, source code, and the transformations between them is depicted in figure 1-1.

1.1.2 Automation of Transformations

Another key element in the MDA is that the transformations are executed by tools. Many tools have been able to transform a platform specific model to code, there is nothing new to that. What's new, is that the transformation from PIM to PSM is automated as well. This is where the obvious benefits of MDA lie. Anyone that has been around a while in the business of software development knows how much time is spend on tasks that are more or less routine. For example, building a database model from an object-oriented design, or building a COM component model, or an EJB component model from another high level design. The MDA goal is to automate the routine part of software development.

1.1.3 MDA Building Blocks

The MDA framework consists of a number of highly related parts. To understand the framework we must understand every part in itself, and their mutual relationships. So, let's take a closer look at each of the parts of the MDA framework, which are depicted in figure 1-2.

Models

The first and foremost element of the MDA is formed by models, high level models, PIMs, and low level models, PSMs. The PIMs must truely be independent of any implementation technology. The whole idea of MDA is that a PIM can be transformed into more than one PSM, each suited for different target technologies. When the PIM would reflect design decisions made with only one of the target technologies in mind, it could not be transformed into a PSM based on a different target technology.

A PSM, on the other hand, must closely reflect the concepts and constructs used in its technology. In a PSM targeted at databases, for instance, the table, column, and foreign key concepts should be clearly recognisable. The close relationship between the PSM and its technology ensures that the transformation to code will be efficient and effective.

All models, both PSM and PIM, should be consistent, precise, and contain as much information as possible about the system. This is were OCL can be helpful, because UML alone does not provide enough information.

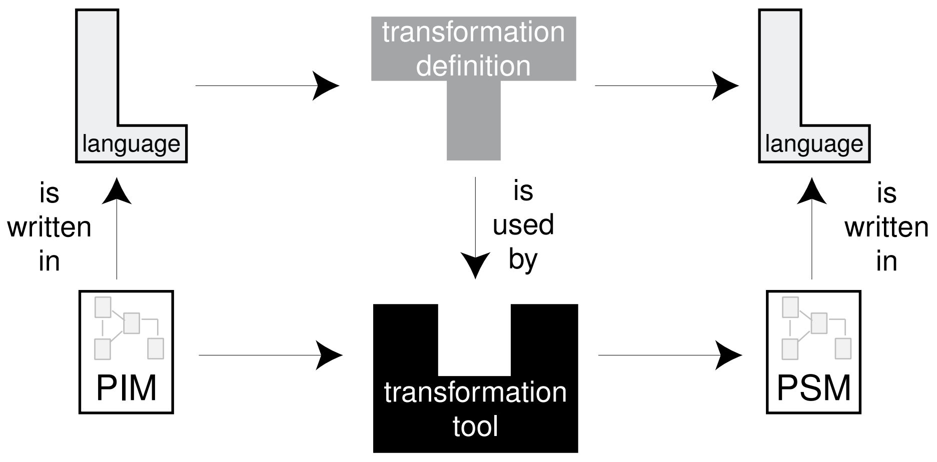

Figure 1-2 *The MDA Framework.*

Modeling Languages

Modeling languages form another element of the MDA framework. Because both PIMs and PSMs are transformed automatically, they should be written in a standard, well-defined modeling language that can be processed by automated tools. Still, the PIMs are written by hand. What a system must do, is (before it is built) only known by humans. Therefore PIMs must be written to be understood and corrected by other humans. This places high demands on the modeling language used for PIMs. It must be understood by both humans and machines.

The PSMs, however, will be generated, and it needs to be understood by experts in that specific technology only. The demands on the languages used forspecifying PSMs are relatively lower than those on the language for PIMs. Currently, there are a number of so called profiles for UML that define a UML-like language for a specific technology, e.g. the EJB profile [EJB01].

Transformation Tools

There is a growing market of transformation tools. These tools implement the central part of the MDA approach, which is that a substantial portion of the software development process can be automated. There are, and have been many tools that implement the PSM to code transformation. Currently, there are only few that implement the execution of the transformation definitions from PIM to PSM. Most of the PIM to PSM tools are combined with a PSM to code component. Preferably the tools should offer the user the flexibility to tune the transformation to their specific needs.

Transformation Definitions

Another vital part of the MDA framework is formed by the definitions of how a PIM is to be transformed to a specific PSM, and how a PSM is to be transformed into code. Transformation definitions are separated from the tools that will execute them in order to reuse them, even with different tools. It is not worthwhile to build a transformation definition to be used once. It is far more effective when a transformation can be executed over and over again on any PIM or PSM written in a specific language.

Some of the transformation definitions will be 'home-made', that is, made by the company that works according to the MDA process. Preferably, transformation definitions would be in the public domain, perhaps even standardized, and tuneable to the individual needs of its users. Currently, some tool vendors have developed their own transformation definitions, which usually cannot be apapted by the users .

1.1.4 MDA Benefits

What benefits would application of MDA bring? Here are some of the advantages of MDA:

1. *Portability*, increasing application re-use and reducing the cost and complexity of application development and management, now and in the future.
2. *Productivity*, by allowing developers, designers and system administrators to use languages and concepts they are comfortable with, and still support seamless communication and integration across the teams.
3. Cross-platform *interoperability*, using rigorous methods to guarantee that standards based on multiple implementation technologies all implement identical business functions.
4. Easier *maintenance and documentation*.

MDA brings us portability and platform independency because the PIM is indeed platform independent and can be used to generate several PSMs for different platforms. A productivity gain can be achieved by the use of tools that fully automate the generation of code from a PSM, and even more when the generation of a PSM from a PIM is automated as well. The promise of cross-platform interoperability can be fulfilled by tools that do not only generate PSMs, but the bridges between them, and possibly to other platforms, as well.

Note that the MDA implies that much of the information about the application must be incorporated in the PIM. It also implies that building a PIM takes less effort than writing code.

1.1.5 The Silver Bullet?

When explaining the MDA to software developers, we often get a sceptical response. "This can never work. You cannot generate a complete working program from a model. You will always need to adjust the code." is a typical reply. Is the MDA just promising another silver bullet?

We believe that the MDA may change the future of software development radically. One argument for this is that although the MDA is still in its infancy, you can today achieve great gains in productivity, portability, interoperability and maintenance effort by applying the MDA using a good transformation tool. Therefore it is, and will be used. A second argument comes from the history of computing.

In the early 1960s our industry was in the middle of a revolution. The use of existing assembly languages was substituted by the use of procedural languages. In those days too there was a lot of secptism, and not without reason. The first compilers were not very good. The Algol programming language, for instance, offered a possibility to give hints to the compiler on how to translate a piece of code. Many programmers were concerned that the generated assembler code would be far less efficient than handwriting the assembler code themselves. Many could not believe that compilers would become good enough to stop worrying about this.

To a certain extent the sceptists were right. You lost efficiency and speed, and you could not program all the assembler tricks in a procedural language. However, the advantages of procedural languages became more and more obvious. Using higher level languages you can write more complex software much faster, and the resulting code is much easier to maintain. At the same time better compiler technology diminished the disadvantages. The generated assembler code became more efficient. Today we accept the fact that we should not program our systems in assembler. Even the opposite, if someone says he is planning to write his new customer relationship management system in assembler, he would be declared insane.

What MDA brings us is another revolution of the same kind. Eventually PIMs can be *compiled* (read: transformed) into PSMs, which are *compiled* into procedural language code (which itself is compiled into assembly or raw machine code). The PIM to PSM *compilers* (read: transformation tools) will not be very efficient for some years to come. Its users will need to give hints on how to transform parts of a model. But eventually the advantage of working on a higher level of abstraction will become clear to everyone in the business.

Concluding we can say that although the MDA is still in its infancy, it shows the potential of changing software development radically. Changes are, we are witnessing the birth of a paradigm shift, and in the near future software development will shift its focus from code to models.

1.1.6 MDA Requirement: Good, Solid Models

A requirement for applying the MDA process, is that the models the process starts off with, are very good, solid models. They should contain as much information about the system as possible, be consistent, and written so that both computers and humans can understand them.

At this moment, the best choice for a modeling language for building PIMs is the UML in combination with the OCL. This is the only option that ensures consistent and precise models, while preserving the readability.

1.2 MODELING MATURITY LEVELS

Within the MDA process the focus of software development will be on producing a good, high level, independent model of the system. Currently many people use the UML or another modeling language during software development. But they all use this standard language in very different ways. To create some order and transparency in working with models we introduce the modeling maturity levels

(MMLs). These levels can be compared to the CMM levels [CMM95]. They give an indication about the role that models play in your software development process, and the direction you need to take to improve this process.

Traditionally there has been a gap between the model and the system. The model is used as plan, as brainstorm material, or as documentation, but the system was the real thing. Often the detailed software code strays a long way from the original model. On every modeling maturity level this gap is made smaller, as shown in figure 1-3.

As one climbs to a higher maturity level, the term programming gets a new meaning. To be clear in the descriptions of each level, we use the word *coding* to mean the final transformation of all knowledge and decisions about the application - in what form soever - to programming language code. Corresponding we speak of the *coder* instead of the *programmer*.

1.2.1 Level 0: No Specification

At the lowest level the specification of the software is in the heads of the developers only. This level is common among non-professional software developers. One simply thinks up the idea of what to develop, and talks about it without ever writing anything down. The characteristics of this level are:

- There are often conflicting views among developers, and between developers and users.
- This manner of working is usable for small applications, larger and more complex applications need some form of design before coding.
- It is impossible to understand the code if the coders leave (and they always do).
- Many choices are made by the coders in an ad-hoc fashion.

1.2.2 Level 1: Textual

At modeling maturity level 1 the specification of the software is written down in one or more natural language documents. These may be more or less formal, using numbering for every requirement or every ‘system function’ or not, large or small, only overview or very detailed, depending on taste. This is the lowest level of professional software development. The characteristics of this level are:

- The specification is ambiguous, because natural language inherently is.
- The coder takes business decisions based on his personal interpretation of the text(s).
- It is impossible (or maybe only very difficult) to keep the specification up-to-date after changing the code.

1.2.3 Level 2: Text with Diagrams

At modeling maturity level 2 the specification of the software is given by one or more natural language documents augmented with several high-level diagrams to explain the overall architecture, and/or some complex details. The characteristics of this level are:

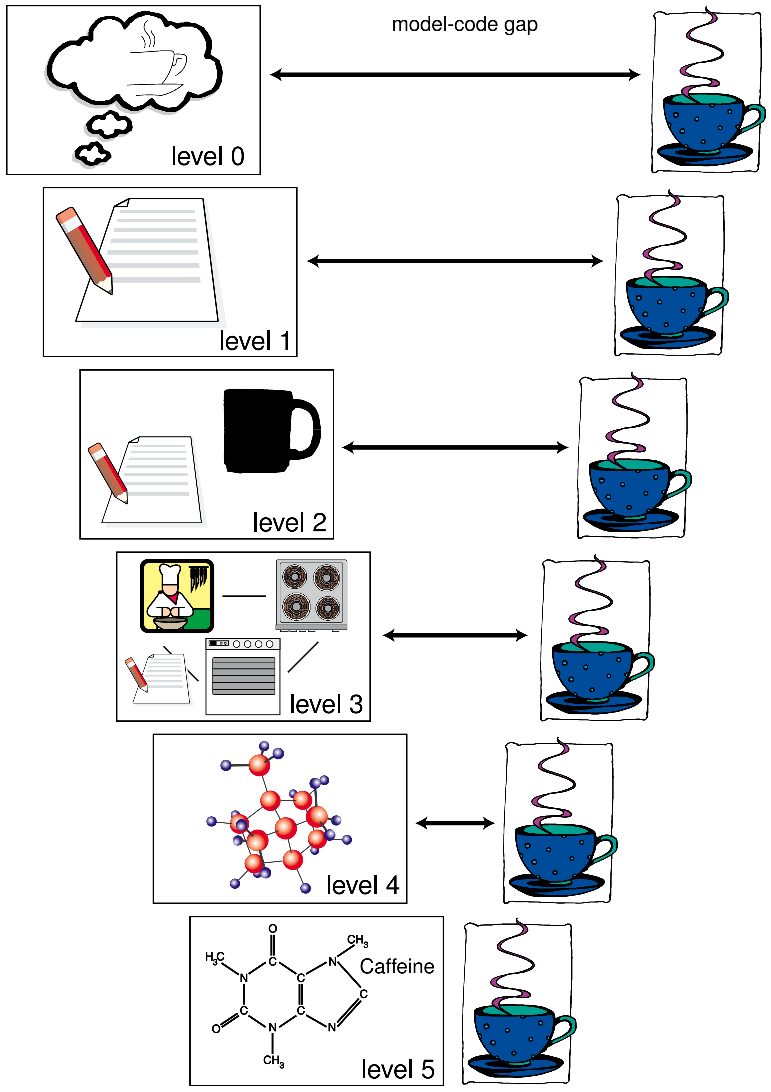

Figure 1-3 *The Modeling Maturity Levels bridging the model-code gap.*

- The text still specifies the system, but it is easier understood because of the diagrams.
- All characteristics of level 1 are still present.

1.2.4 Level 3: Models with Text

A set of models, i.e. either diagrams or text with a very specific and well specified meaning, forms the specification of the software at modeling maturity level 3. Additional natural language text is used to explain the background and motivation of the models, and fills in many details, but the models are the most important part of the design/analysis deliverables. The characteristics of this level are:

- The diagrams or formal texts are real representations of the software.
- The transition of model to code is mostly manual.
- It is still impossible (or maybe only very difficult) to keep the specification up-to-date after changing the code.
- The coder still makes business decisions, but these have less influence on the architecture of the system.

1.2.5 Level 4: Precise Models

A model, meaning one consistent, coherent set of texts and/or diagrams with a very specific and well specified meaning, is specifying the software at modeling maturity level 4. Here too, natural language text is used to add comments that explain the background and motivation of the models. The models at this level are precise enough to have a direct link with the actual code. Yet, they have a different level of abstraction. A model is more than the concepts of some programming language depicted in diagrams. Level 4 is the level at which the Model Driven Architecture is targeted. At this level:

- Coders do not make business decisions anymore.
- Keeping models and code up-to-date is essential and easy.
- Iterative & incremental development are facilitated by the direct transformation from model to code.

1.2.6 Level 5: Models Only

A level 5 model is a complete, consistent, detailed, and precise description of the system. At level 5 the models are good enough to allow complete code-generation. No adjustments need to be made to the resulting code. Software developers can rely on the model-to-code generation in the same way coders today rely on their compilers. The generated code will be invisible to the developer, there is no need to look into it. The language in which the models are written has become the next generation programming language. Certainly some text is still present in the model, but its function is equal to comments in source code.

Note that this level has not been realized yet anywhere in the world. This is future technology, unfortunately. Still it is good to recognize what we our ultimate goal is.

1.3 BUILDING BETTER MODELS

So, to be able to apply the MDA process, models on maturity level 4 are neccessary. This is the first level where a model is more than just paper. At level 4 the models are precise eniugh to have a direct link with the source code. If you want to be able to transform a model from PIM through a PSM to code this precision is neccesary.

But how do you build level 4 models? In section 1.1.6 we already mentioned that the best choice for a modeling language for building PIMs is the UML in combination with the OCL. By specifying your model in a combination of the UML and OCL languages, we claim that you can improve the quality of your models.

1.3.1 Why Combine UML and OCL?

Modeling, specially software modeling, up to now has almost been a synonym for producing diagrams. Most models consist of a number of "bubbles and arrows" pictures and some accompanying text. The information conveyed by such a model has a tendency to be incomplete, informal, imprecise and sometimes even inconsistent.

Many of the flaws in the model are caused by the limitations of the diagrams being used. A diagram can simply not express the statements that should be part of a thorough specification. For instance, in a UML model an association between class *Flight* and class *Person*, as shown in figure 1-4, indicating that a certain group of persons are the passengers on a flight, will have multiplicity many (0..*) on the

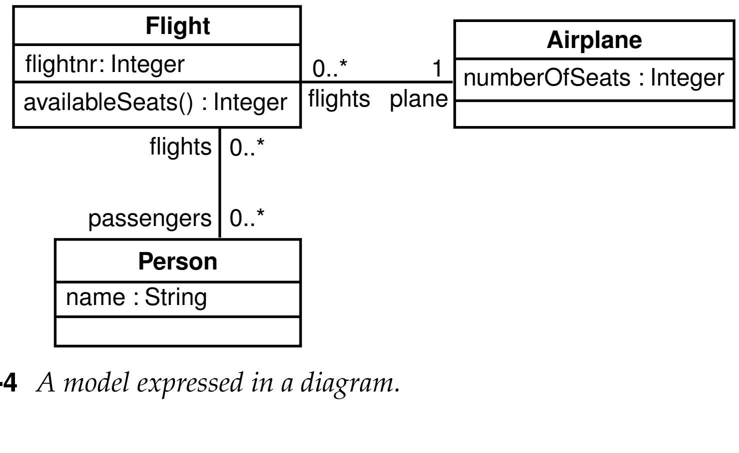

Figure 1-4 *A model expressed in a diagram.*

side of the *Person* class. This means that the number of passengers is unlimited. In reality the number of passengers will be restricted to the number of seats on the airplane that is to perform the flight. It is impossible to express this restriction in the diagram. In this example the correct way to specify the multiplicity is to add to the diagram the following OCL constraint.

```
context Flight
inv: passengers->size() <= plane.numberOfSeats
```

Expressions written in a precise, mathematically-based language like OCL, convey a number of benefits over the use of diagrams to specify a (business or software) system. For example, these expressions cannot be interpreted differently by different people, e.g. an analist and a programmer. They are unambiguous and make the model more precise and more detailed. These expressions can be checked by automated tools to make sure they are correct and consistent with other elements of the model. Code generation becomes much more powerfull.

However, a model written in a language that uses an expression representation alone, is often not easily understood. For example, while source code can be regarded as the ultimate model of the software, most people prefer a diagrammatic model in their encounters with the system. The good thing about "bubbles and arrows" pictures is that their intended meaning is easy to grasp.

The combination of the UML and the OCL offers the best of both worlds to the software developer. A large number of different diagrams, together with expressions written in OCL, can be used to specify models. Note that to obtain a complete model, both the diagrams and OCL expressions are necessary. Without OCL expressions the model would be severely underspecified, without the UML diagrams the OCL expressions would refer to non-existing model elements, as there is no way in OCL to specify classes and associations. Only when we combine the diagrams and the constraints, we are able to completely specify the model.

1.3.2 Value Added by OCL

Still not convinced that using OCL adds value to the use of the UML alone? The diagram in Figure 1-5 shows another example, which contains three classes: *Person*, *House*, and *Mortgage*, and their associations. Any human reader of the model will undoubtedly assume that a number of rules must apply to this model. For example:

1. A person may have a mortgage on a house only if that house is owned by him- or herself; one can not obtain a mortgage on the house of one's neighbour or friend.
2. The start date for any mortgage must be before the end date.
3. The social security number of all persons must be unique.

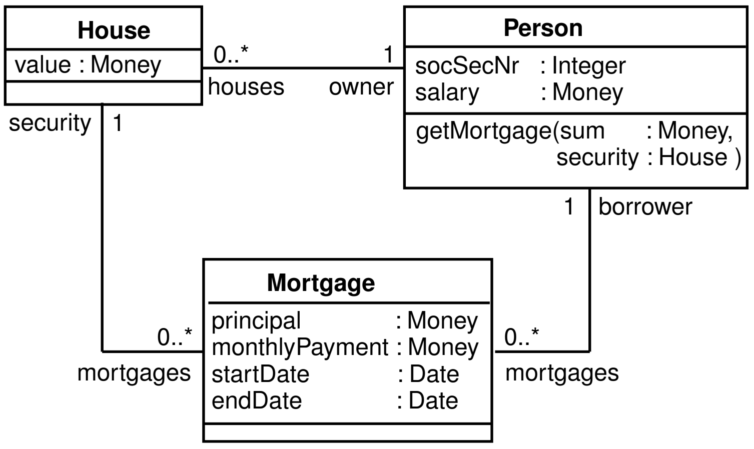

Figure 1-5 *The "mortgage system" expressed in a diagram.*

4. When a person applies for a new mortgage, this will be allowed only when the person's income is sufficient.
5. When a person applies for a new mortgage, this will be allowed only when the counter-value of the house is sufficient.

The diagram does not show this information, nor is there any way in which the diagrams might express these rules. If these rules are not documented, different readers might have different assumptions, and this will lead to incorrect understanding, and incorrect implementation of the system. Writing these rules in English, as we have done above, isn't enough either. By definition English text is ambiguous and very easy to interpret in different ways. The same problem of misunderstanding and incorrect implementation remains.

Only by augmenting the model with the OCL expressions for these rules, a complete and precise description of the "mortgage system" can be obtained. OCL is unambigous and the rules cannot be misunderstood. The rules in OCL are as follows:

```
context Mortgage
inv: security.owner = borrower

context Mortgage
inv: startDate < endDate

context Person
inv: Person::allInstances()->isUnique(socSecNr)

context Person::getMortgage(sum : Money, security : House)
pre: self.mortgages.monthlyPayment->sum() <= 0.30 * self.salary

context Person::getMortgage(sum : Money, security : House)
pre: security.value >= security.mortgages.amount->sum()
```

It is essential to include these rules as OCL expressions in the model for a number of reasons. As stated earlier no misunderstanding occurs when humans read the model. Errors are therefore found in an early stage of the development, when fixing a failure is relatively cheap. The intended meaning of the analist that builds the model, is clear to the programmers that will implement the model.

When the model is not read by humans, but instead it is used as input to some automated system, the use of OCL becomes even more important. Tools can be used for generating simulations and tests, for checking consistency, for generating derived models in other languages using MDA transformations, for generating code, etc. This type of work most people would gladly leave to a computer, if it would and could be done properly.

However, the automation of this work is only possible when the model itself contains all of the information needed. A computerized tool can not interpret English rules. The rules written in OCL include all the necessary information for automated MDA tools. In this way, implementation is faster and more efficient than by hand, and there is a guaranteed consistency between the model in UML/OCL and the generated artifacts. The level of maturity of the software development process as a whole is raised.

1.4 CHARACTERISTICS OF OCL

In the previous section we have seen that even a simple three-class model needs much additional information in OCL to make it complete, consistent and unambiguous. If you only had the UML diagram, many open questions would remain. It is very likely that a system that is built based on the diagram alone will be incorrect. It is clear that OCl is a vital language for building better models. In tghe next section we will take a closer look at some of the characteristics of OCL.

1.4.1 Both Query and Constraint Language

In UML 1.1, OCL was a language to express constraints on the elements given in the diagrams in the model. In the introduction to this book, we already defined a constraint as follows:

> ***A constraint is a restriction on one or more values of (part of) an object-oriented model or system.***

What this means is that although the diagrams in the model state that certain objects or datavalues may be present in the modelled system, these values are valid only if the condition given by the constraint holds. For example, the diagram in figure 1-4 tells us that flights may have any number of passengers, but the constraint restricts the flights to those for which the number of passengers is equal or less than the number of seats on the plane. Flights for which the condition does not hold are not valid in the specified system.

In UML 2, OCL can be used to write not only constraints, but any expression on the elements in the diagram. Every OCL expression indicates a value or object within the system. For example, the expression *1+3* is a valid OCL expression of type *Integer*, that represents the *Integer* value 4. When the value of an expression is of *Boolean* type, it may be used as a constraint. So, the possibilities the language offers, have grown considerably.

OCL expressions can be used anywhere in the model to indicate a value. A value can be a simple value, like an integer, but may also be a reference to an object or a collection of values. An OCL expression can represent e.g. a boolean value used as condition in a statechart or a message in an interaction diagram. An OCL expression can be used to refer to a specific object in an interaction or object diagram. The next expression, for example, defines the body of the operation *availableSeats()* of the class *Flight*.

```
context Flight::availableSeats() : Integer
body: plane.numberOfSeats - passengers->size()
```

Other examples of the use of OCL expressions are the definition of the derivation of a derived attribute or association, or the specification of the initial value of attributes or associations.

Because an OCL expression can indicate any value or collection of values in a system, OCL has the same capabilities as SQL, as proved in [Akehurst01]. This is clearly illustrated by the fact that the body of a query operation can be completely specified by one single OCL expression. But neither SQL nor OQL is a constraint language. OCL is a constraint and query language at the same time.

1.4.2 Mathematical Foundation but No Mathematical Symbols

An outstanding characteristic of OCL is its mathematical foundation. It is based on mathematical set theory and predicate logic, and it has a formal mathematical semantics [Richters01]. The notation however does not use mathematical symbols. Experience with formal or mathematical notations have lead to the following conclusion: The people who can use the notation can express things precisely and unambiguously, but very few people can really use such a notation. Although it seems a good candidate for a precise, unambiguous notation, a mathematical notation is not suitable for a standard language that is to be widely used.

A modeling language needs the rigor and precision of mathematics, but the ease of use of natural language. These are conflicting requirements, so finding the right balance is essential. In OCL this balance is found by using the mathematical concepts without the abracabrada of the mathematical notation. Instead of using mathematical symbols OCL uses plain ascii words that express the same concept.

Especially in the context of MDA where a model needs to be transformed automatically, the value of an unambiguous mathematical foundation of the PIM language is of great value. There can be no doubt as to what an OCL expression means and different tools will understand the expressions in the same way.

The result is a precise, unambiguous language that should be easily read and written by all practitioners of object technology and by their customers, people who are not mathematicians or computer scientists. If you still do not like the syntax of OCL, you may define your own. The OCL specification allows anyone to define his own syntax. The only condition is that your syntax can be mapped to the language structures defined in the standard [BIBREF]. In Appendix C an example is given of a different syntax expressing the same underlying structures. This syntax is directed towards use by business modelers.

1.4.3 Strongly Typed Language

An apparant characteristic of OCL is that it is a typed language. OCL expressions will be used for modeling and specification purposes. Because most models are not executed directly, most OCL expressions will be written while no executable version of the system exists. However, it must be possible to check an OCL expression without having to produce an executable version of the model. As a typed language, OCL expressions can be checked during modeling, before execution. Thus, errors in the model can be removed in an early stage.

Many popular programming languages are typed alnguages as well. Java, Eiffel, C#, delphi, etc. all fall into this category.

1.4.4 Declarative Language

Another distinguishing feature is that OCL is a declarative language. In procedural languages, like programming languages, expressions are descriptions of the actions that must be performed. In a declarative language an expression simply states *what* should be done, but now *how*. To ensure this, OCL expressions have no side effects; that is, the state of a system does not change because of an OCL expression. There are several advantages of declarative languages over procedural languages.

First, the modeler can make decisions at a hign level of abstraction, without going into details of how something should be calculated. For example, the body expression of the operation *availableSeats()* in section 1.4.1 clearly specifies what the operation should calculate. How this should be done is udefined, this will depend on the implementation strategy of the whole system. One such choice is the representation of associations in the code. How do we find all of the passengers to a Flight ? A flight object could contain a collections of references to its passengers. Alternatively, a Flight has no direct reference to its passengers, but needs to search a Passenger table in a database to find its passengers. A third implementation strategy would be to add an additional attribute to *Flight*, containing the number of passengers. Care should be taken to update the value of this attribute whenever a *Passenger* books or cancels a *Flight*. The body-expression in OCL allows for all of these implementations, because it only describes the what, not the how. If a procedural language was used to specify the body of *availableSeats()* it would have forced a specific implementation style, which is undesirable for a PIM-level model.

In making OCL a declarative language, the expressions in a UML model are lifted fully into the realm of pure modeling, without regard for the nitty-gritty details of the implementation and the implementation language. An expression specifies values at a high level of abstraction, while still being one hundred percent precise.

1.5 SUMMARY

OCL is a language that can express additional and neccesary information on the models and other artifacts used in object-oriented modeling, and should be used in conjunction with the UML diagrammatic models.

Much more information can be included in the specification (the model) using the combination of the OCL and the UML, than through the use of UML alone. As we have seen, even in very simple examples, a large number of essential aspects of the system can not be expressed in a UML diagram. This information can only be expressed in the form of OCL expressions.

The level of maturity of the software process is raised by building a UML/OCL combined model. Tools that simulate, generate tests or source code from a model, and tools that support MDA™ need more detailed, and more precise models as input. The quality of the output of these kind of tools depends to large amount on the quality of the model that is used as input.

Tools that generate tests and source code from UML/OCL make the development process more efficient. The time invested in developing the UML/OCL models is gained back during the following stages of development.

OCL is a precise, unambiguous language that should be easily read and written by all practitioners of object technology and by their customers. This means that the language must be understood by people who are not mathematicians or computer scientists. It therefore doesn't use any mathemantical symbols, but maintains the mathematical rigor in its definition.

OCL is a typed language. OCL expressions will be used for modeling and specification purposes. Because most models are not executed while being developed, most OCL expressions will not be executed at the time of writing. However, it must be possible to check an OCL expression without having to produce an executable version of the model. As a typed language, OCL expressions can be checked during modeling, before execution.

OCL is a declarative language. Its expressions have no side effects; that is, the state of a system does not change because of an OCL expression. More importantly, a modeler can specify in OCL exactly what is ment, without restricting the implementation of the system that is being modeled. Implementation choices can be made based on the technical environment and requirements. When the technical choices change, because of a change of platform, of a better insight in the required performence, the OCL expressions will remain unchanged. This allows a UML/OCL model to be really platform independent.

Chapter 2

OCL By Example

The sample system in this chapter provides a short and informal introduction to OCL. A complete and rigorous description of OCL can be found in Part 2. After reading this chapter you will be able to add simple OCL expressions to your own UML models.

2.1 THE "ROYAL AND LOYAL" SYSTEM EXAMPLE

As an example, we have modeled a computer system for a fictional company called Royal and Loyal (R&L). R&L handles loyalty programs for companies that offer their customers various kinds of bonuses. Often, the extras take the form of bonus points or air miles, but other bonuses are possible as well: reduced rates, a larger car for the same price as a standard rental car, extra or better service on an airline, and so on. Anything a company is willing to offer can be a service rendered in a loyalty program. Figure 2-1 shows the UML class model R&L uses for most of its clients.

The central class in the model is *LoyaltyProgram*. A system that administers a single loyalty program will contain only one instance of this class. In the case of R&L many instances of this class will be present in the system. A company that offers its customers a membership in a loyalty program is called a *ProgramPartner*. More than one company can enter into the same program. In that case, customers who enter the loyalty program can profit from services rendered by any of the participating companies.

Every customer of a program partner can enter the loyalty program by filling in a form and obtaining a membership card. The objects of class *Customer* represent the people who have entered the program. The membership card, represented by the class *CustomerCard*, is issued to one person. Card use is not checked, so the card could be used as a single card for a whole family or business. Most loyalty programs allow customers to save bonus points. Each individual program partner decides when and how many bonus points are allotted for a certain pur-

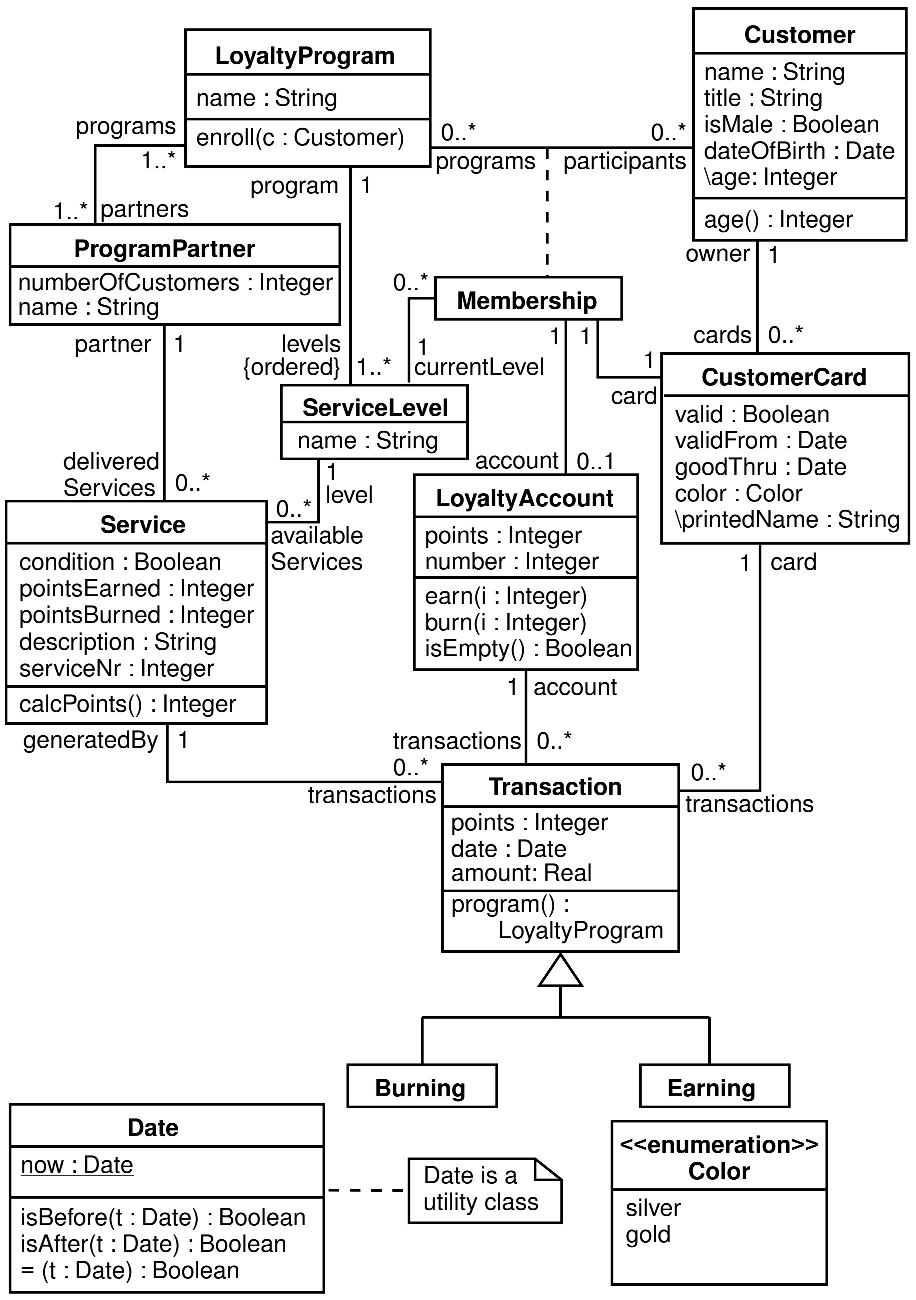

Figure 2-1 *The Royal and Loyal model.*

chase. Saved bonus points can be used to "buy" specific services from one of the program partners. To account for the bonus points that are saved by a customer, every membership can be associated with a *LoyaltyAccount*.

Various transactions on this account are possible. For example, the loyalty program "Silver and Gold" has four program partners: a supermarket, a line of gas stations, a car rental service, and an airline.

- At the supermarket the customer can use bonus points to purchase items. The customer earns 5 bonus points for any regular purchase over the amount of $25.
- The gas stations offer a discount of 5% on every purchase.
- The car rental service offers 20 bonus points for every $100 spent.
- Customers can save bonus points for free flights with the airline company. For every flight that is paid for normally, the airline offers 1 bonus point for each 15 miles of flight.

In this situation, there are two types of transactions. First, there are transactions in which the customer obtains bonus points. In the model (see figure 2-1), these transactions are represented by a subclass of *Transaction* called *Earning*. Second, there are transactions in which the customer spends bonus points. In the model they are represented by instances of the *Burning* subclass of *Transaction*. The gas stations offer simple discounts but do not offer or accept bonus points. Because the turnover generated by the customers needs to be recorded, this is entered as two simultanous transactions on the *LoyaltyAccount*, one *Earning* and one *Burning* for the same number of points.

Customers in the Silver and Gold program who make extensive use of the membership are rewarded with a higher level of service: the gold card. All normal services are available to customers who have a gold card, and they are also offered additional services.

- Every two months the supermarket offers an item that is completely free. The average value of the item is $25.
- The gas stations offer a discount of 10% on every purchase.
- The car rental service offers a larger car for the same price.
- The airline offers its gold card customers a business class trip for the economy class price.

Customers must meet at least one of the following conditions to get a gold card.

- Three sequential years of membership with an average annual turnover of $5,000, or
- One year of membership with a turnover of $15,000, where the turnover is the total turnover with all program partners.

To administer different levels of service, the class *ServiceLevel* is introduced in the model. A service level is defined by the loyalty program and used for each membership.

R&L advertises the program and its conditions. It manages all customer data and transactions on the loyalty accounts. For this purpose, the program partners must inform R&L of all transactions on loyalty program membership cards. Each year, R&L sends new membership cards to all customers. When appropriate, R&L upgrades a membership card to a gold card. In this case, R&L sends the customer a new gold card along with information on the additional services offered, and R&L invalidates the old membership card.

The customer can withdraw from the program by sending a withdrawal form to R&L. Any remaining bonus points are canceled and the card is invalidated. R&L can invalidate a membership when the customer has not used the membership card for a certain period. For the Silver and Gold program, this period is one year.

We could tell you more about R&L, but the preceding description is sufficient for our purposes. The diagram in figure 2-1 outlines the system model. Now it's time to add the neccessary details by adding some expressions and stating a number of useful and indispensable constraints.

2.2 ADDING EXTRA INFORMATION

The diagram in Figure 2-1 does not express all relevant information on the R&L system. This and the following sections will gives examples of additional information that cannot be expressed in the diagram, but should be given in OCL expressions.

2.2.1 Initial Values and Derivation Rules

A very basic addition to the diagram shown in Figure 2-1 is to include rules that state initial values for attributes and association ends. Initial value rules can be expressed very simply. First, you indicate the class that holds the attribute or association end. This class is called the *context*. Then you write the expression that states your initial value rule. For instance, a loyalty account will always be initialized with zero points, and a customer card will always be valid at the moment it is issued.

```
context LoyaltyAccount
init: points = 0

context CustomerCard
init: valid = true
```

Another small but indispensible part of the model is the specification of how the value of derived elements is to be determined. A model may contain derived attributes and derived associations. For both a so-called *derivation rule* can be specified. Again the context of the expression is the class that holds the attribute or association end. The expression after the context states the derivation rule. For instance, the derived attribute *printedName* of *CustomerCard* is determined based on the name and title of the owner of the card.

```
context CustomerCard
derive: printedName = owner.title.concat(‘ ‘).concat(owner.name)
```

In this example, the printedName is the concatenation of the title of the Customer that owns the card, with its name, with a space between both strings (e.g. "Mr. Johnson").

2.2.2 Query Operations

Query operations are operations that do not change the state of the system, they simply return a value or set of values. The definition of the result of query operations cannot be given in a diagram. In OCL this can be defined by writing a *body expression*. As context the operation name, parameters and return type (its signature) is given. For instance, suppose the class *LoyaltyProgram* has a query operation *getServices*, which returns all services offered by all program partners in this program.

```
context LoyaltyProgram::getServices(): Set(Service)
body: partners.deliveredServices
```

In this example, the association end *deliveredServices* of *ProgramPartner* that holds a set of *Services*, is used. For all instances of *ProgramPartner* that are associated with the *LoyaltyProgram* instance of which the operation *getServices* is called, these sets of Services are collected and combined into one set. This set is the result of the query operation.

In the body expression the parameters of the operation may be used. For instance, suppose there is need for a more refined version of the *getServices* operation. This operation takes as parameter a program partner object and returns the services delivered by the parameter object if it is a partner in this program. In this case the refined operation can be specified as follows.

```
context LoyaltyProgram::getServices(pp: ProgramPartner)
                                      : Set(Service)
body: if partners->includes(pp)
      then pp.deliveredServices
      else Set{}
      endif
```

The result of this query operation is the set of *Services* held by the parameter *pp*, or an empty set if the parameter instance of *ProgramPartner* is not a partner of the *LoyaltyProgram* for which the query operation is called.

2.2.3 Defining New Attributes And Operations

Although most elements in the model are introduced in the UML diagrams, attributes and operations can be added to the model using an OCL expression. The context is the class to which the attribute or operation is added.

An attribute that has been defined in this manner is always a derived attribute. The expression that defines the attribute includes the name and type of the attribute, and the derivation rule. For instance, we might want to introduce an attribute called *turnover* in class *LoyaltyAccount*, which would sum the amount attributes of the transactions on the account. This attribute can be defined by the following expression. The part after the equal sign states the derivation rule for the new attribute.

```
context LoyaltyAccount
def: turnover : Real = transactions.amount->sum()
```

The derivation rule results in a single real number that is calcaluted by summing up the value of the amount attribute in all transactions associated with the *LoyaltyAccount* instance that holds the newly defined attribute.

An operation that has been defined in an OCL expression is always a query operation. In this case the expression after the equal sign states the body expression. For example, we might want to introduce the operation *getServicesByLevel* in the class *LoyaltyProgram*. This query operation returns the set of all delivered services for a certain level in a loyalty program.

```
context LoyaltyProgram
def: getServicesByLevel(levelName: String): Set(Service)
   = levels->select( name = levelName ).availableServices->asSet()
```

The result of the body expression is calculted from a selection of the levels associated with the instance of *LoyaltyProgram* for which the operation *getServicesByLevel* is called. It returns only the services available from the *ServiceLevel* which name is equal to the parameter *levelName*. Why the *asSet* operation is used, is explained in section 2.4.

2.3 ADDING INVARIANTS

More information can be added to the model in the form of invariants. An *invariant* is a constraint that should be true for an object during its complete lifetime.

Invariants often represent rules that should hold for the real life objects after which the software objects are modeled.

2.3.1 Invariants on Attributes

A reasonable rule for every loyalty program of R&L would be to demand that every customer who enters a loyalty program be of age. In the model, this means that the attribute *age* of every customer object must be equal to or greater than 18. This can be written as an invariant:

```
context Customer
inv: age >= 18
```

Invariants on one or more attributes of a class can be expressed in a very simple manner. The class to which the invariant refers is the context of the invariant. It is followed by a boolean expression that states your invariant. All attributes of the context class may be used in this invariant.

2.3.2 The Type of the Attribute is a Class

When the attribute is not of a standard type, such as *Boolean* or *Integer*, but its type is a class itself, you can use the attributes or query operations defined on that class to write the invariant, using a dot notation. For example, a simple but useful invariant on the date attribute of *CustomerCard* states that *validFrom* should be earlier than *goodThru*:

```
context CustomerCard
inv: validFrom.isBefore(goodThru)
```

This invariant uses the operation *isBefore* in the *Date* class that checks whether the date in the parameter is later than the date object the operation is called on, and that results in a boolean value. Note that you may only use operations that do not change the value of any attributes, only the so-called query operations are allowed.

2.3.3 Invariants on Associated Objects

Invariants may also state rules on associated objects. The rule that every customer should be of age could also be stated as follows:

```
context CustomerCard
inv: owner.age >= 18
```

Stating an invariant on associated objects is done by using the *rolename* on the association to refer to the object on the other end. Or, if the rolename is not present, you can use the name of the class. Giving rolenames to each association end is preferred. Using rolenames to go to associated instances is called *navigation*.

Finding the right context for an invariant can sometimes be challenging. Usually the right context is the context for which the invariant can be expressed in the simplest way.

2.3.4 Using Association Classes

Association classes may also be used in OCL expressions, but there are some special rules. Association classes never have a rolename, therefore the name of the association class is used to refer to an instance of that class. For instance, the next invariant uses the association class Membership, to state that the service level of each membership must be a service level known to the loyalty program for which the invariant holds.

```
context LoyaltyProgram
inv: levels->includesAll(Membership.currentLevel)
```

From an association class you may navigate to the instances of both classes at the end of the association.

```
context Membership
inv: participants.cards->includes(self.card)
```

All classes that are associated normally to the association class may be used ...

```
context Membership
def: oper getCurrentLevelName() : String
     body: currentLevel.name
```

To be done: Add more explanation.

2.3.5 Using Enumerations

In an UML model enumeration types may be defined. An enumeration type may be used for instance as an attribute type in a UML class model. The values of an enumeration type are indicated in an OCL expression by the name of the enumeration type, followed by two colons, followed by the valuename. An example can be found in the *CustomerCard* class, where the attribute *color* can have two values, either *silver* or *gold*. The following invariant states that the color of this card must match the service level of the membership.

```
context Membership
inv: currentLevel.name = 'Silver' implies
                                   card.color = Color::silver
     and
     currentLevel.name = 'Gold' implies card.color = Color::gold
```

2.4 WORKING WITH COLLECTIONS OF OBJECTS

Often the multiplicity of an association is greater than 1, thereby linking one object to a collection of objects of the associated class. To deal with such a collection, OCL provides a wide range of collection operations, and distinguishes between different types of collections.

2.4.1 Using Collections Operations

Whenever navigating the link in an expression results in a collection of objects, you can use one of the *collection operations*. To indicate the use of one of the predefined collection operations you write an arrow between the rolename and the operation. When you use an operation defined in the model, you write a dot.

The *size* operation

For R&L it would be reasonable to demand that a loyalty program must have at least one service it can offer to its customers. Using the dot notation we can navigate from the context of a loyalty program through its program partners to the services they deliver. This results in a collection of *Service* instances. With the arrow notation we can indicate the use of the predefined operation *size*. The rule would be stated as follows:

```
context LoyaltyProgram
inv: partners.deliveredServices->size() >= 1
```

The *select* operation

Another invariant on the R&L model is that the number of valid cards for every customer must be equal to the number of programs the customer participates in. This constraint can be stated using the *select* operation on sets. The *select* takes an OCL expression as parameter. The result of the *select* is a subset of the set on which it is applied, where the parameter expression is true for all elements of the subset. In the following example, the result of the *select* is the subset of *cards*, where the attribute *valid* is true.

```
context Customer
inv: programs->size() = cards->select( valid = true )->size()
```

The *forAll* and *isEmpty* operations

Also relevant for the R&L model is that, when none of the services offered in a *LoyaltyProgram* credits or debits the *LoyaltyAccount* instances, these instances are useless and should not be present. We use the *forAll* operation on the collection of all services to state that all services comply with this condition. The *forAll* operation, like the *select*, takes an expression as parameter. Its outcome is boolean: true if the expression evaluates to true for all elements in the collection, and otherwise false. The following invariant states that when the *LoyaltyProgram* does not have the possibility for earning or burning points, the members of the *LoyaltyProgram* do not have *LoyaltyAccount*s; that is, the collection of *LoyaltyAccount*s associated with the *Membership*s must be empty.

```
context LoyaltyProgram
inv: partners.deliveredServices->forAll(
            pointsEarned = 0 and pointsBurned = 0 )
        implies membership.account->isEmpty()
```

Note that the fact that we define a constraint for a class already implies that the condition holds for all instances of that class. There is no need to write:

```
context LoyaltyProgram
inv: forAll( partners... )
```

In fact, this is an incorrect expression. The *forAll* operation is used in those cases where we already have a subset of all instances of a class, and we want to check on the elements of that subset. In the above example all services delivered by the partners of a certain loyaltyprogram is a subset of all instances of class *Service*. This subset is checked to include only those services for which the *pointsEarned* and *pointsBurned* are equal to zero. (See section 3.10.3 for more information on this topic.)

The above example also introduces two logical operations: *and* and *implies*. The *and* operation is the normal *and* operation on booleans. The *implies* operation states that when the first part is true, the second part must also be true; when the first part is not true, it does not matter whether the second part is true, the whole expression is true.

The *collect* operation

A collection operation that is used very often is the *collect* operation. It is used when, for instance, you want to take the set of all values for a certain attribute of all objects in a collection. In fact, a large number of the examples in this chapter uses this operation, because the dot notation is an abbreviation for applying the *collect* operation. Take for instance the following expression in the context of *LoyaltyProgram*.

```
partners.numberOfCustomers
```

Another way to write it is:

```
partners->collect( numberOfCustomers )
```

It means that for each element in the collection of partners of a loyalty program, the value of the attribute *numberOfCustomers* is added to a new collection, in this case, containing *Integer* values.

The collect operation may also be used to build up a new collection from association ends.

```
partners.deliveredServices
```

Another way to write this is:

```
partners->collect( deliveredServices )
```

To be done: More explaining text

More collection operations

Here are some more collection operations (the complete list can be found in Chapter 8):

- *notEmpty*, which is true when the collection has at least one element
- *includes(object)*, which is true when *object* is an element of the collection
- *union(collection of objects)*, which results in a collection of objects that holds the elements in both sets
- *intersection(collection of objects)*, which results in a collection of objects that holds all elements that are in both collections

2.4.2 Sets, Bags, Ordered Sets and Sequences

When working with collections of objects, you should be aware of the difference between a *Set*, a *Bag*, an *OrderedSet* and a *Sequence*. In a Set, each element may occur only once. In a Bag, elements may be present more than once. A Sequence is a bag in which the elements are ordered. An OrderedSet is a set in which the elements are ordered.

Navigations Resulting in Sets and Bags

To understand why these differences are important, take a look at the attribute *numberOfCustomers* of class *ProgramPartner*. We want to state that this attribute holds the number of customers who participate in one or more loyalty programs offered by this program partner. In OCL, this would be expressed as

```
context ProgramPartner
inv: numberOfCustomers = programs.participants->size()
```

But there is a problem with this expression. A customer can participate in more than one loyalty program. In other words, an object of class *Customer* could be repeated in the collection *program.participants*. So this collection is a bag and not a set. In the preceding expression, these customers are counted twice, and that is not what we intended.

In OCL, the rule is that when you navigate through more than one association with multiplicity greater than 1, you end up with a bag. That is, when you go from A to more than one B to more than one C, the result is a bag of Cs. When you navigate just one such association you get a set. There are standard operations that transform one of the types into any of the other types. Using one of these operations we can correct the previous invariant:

```
context ProgramPartner
inv: numberOfCustomers = program.participants->asSet()->size()
```

When you navigate through an association with multiplicity greater than 1 on the end you are navigating to, and next through an association with multiplicity greater than 1 on the end you are navigating from, you also end up with a bag. The expression 'transactions.service' from the context of *CustomerCard* denotes a bag of instances of *Service*. Every service may be associated with more than one transaction, so when you take the services of a set of transactions, some services might be present more than once.

Navigations Resulting in OrderedSets and Sequences

When you navigate an association marked *{ordered}*, the resulting collection is an *OrderedSet,* and following the rules explained above, when you navigation more than one association and one of them is marked *{ordered},* you end up with a *Sequence*. Several standard operations deal with the order of an ordered set or sequence: *first, last,* and *at(index).* The only ordered association in the R&L model lies between *LoyaltyProgram* and *ServiceLevel*. From the context of *LoyaltyProgram,* the expression *serviceLevel* results in an ordered set. We can state that the first element of this ordered set must be named *Silver* as follows:

```
context LoyaltyProgram
inv: levels->first().name = 'Silver'
```

2.5 ADDING PRECONDITIONS AND POSTCONDITIONS

Pre- and postconditions are constraints that specify the applicability and effect of an operation without stating an algorithm or implementation. Adding them to the model results in a more complete specification of the system.

2.5.1 Simple Pre- and Postconditions

The context of pre- and postconditions is given by the name of the class that holds the operation and the operation signature (its name, parmeters and returntype). In the R&L example, the class *LoyaltyAccount* has an operation *isEmpty.*[1] When the number of points on the account is zero, the operation returns the value *true*. The postcondition states this more precisely, it tells us that the operation returns the outcome of the boolean expression *points = 0*. The return value of the operation is indicated by the keyword *result*.

```
context LoyaltyAccount::isEmpty(): Boolean
pre : -- none
post: result = (points = 0)
```

There is no precondition for this operation, so we include a comment, "none" where the precondition should be placed, indicated by the double dash. To include a precondition, even if it is an empty one, is optional. Rather, it is a matter of style. We prefer to state even empty preconditions, because we feel this is more clear. If, for instance, this example were part of a list of operations with their pre- and postconditions and the precondition for the *isEmpty* operation was the only one missing, the reader might misinterpret its meaning and think that the precondition was mistakingly forgotten.

2.5.2 Previous Values in Postconditions

In a postcondition, the expression can refer to values at two moments in time:

- the value at the start of the operation
- the value upon completion of the operation

The normal value of an attribute, association end, or query operation in a postcondition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, one has to postfix the property name with the keyword '*@pre*', as in the following example:

```
context Customer::birthdayHappens()
pre: -- none
```

[1] Note that this operation is different from the *isEmpty* operation defined on collections.

```
post: age = age@pre + 1
```

The term *age* refers to the value of the attribute after the execution of the operation. The term *age@pre* refers to the value of the attribute *age* before the execution of the operation.

When you need the pre-value of a query operation, the '*@pre*' is postfixed to the operationname, before the parameters.

```
context Company::hireEmployee(p : Person)
post: employees = employees@pre->including(p) and
    stockprice() = stockprice@pre() + 10
```

The '*@pre*' postfix is allowed only in OCL expressions that are part of a Postcondition.

To be done: Find other example.

2.5.3 Messaging in Postconditions

Another thing that is only allowed in postconditions is to specify that communication has taken place. This can be done using the hasSent ('^') operator. For example:

```
context Subject::hasChanged()
post:  observer^update(12, 14)
```

The *observer^update(12, 14)* results in true if an update message with arguments 12 and 14 was sent to *observer* object during the execution of the operation. *Update()* is either an operation that is defined in the class of *observer*, or it is a signal specified elsewhere in the UML model. The argument(s) of the message expression (12 and 14 in this example) must conform to the parameters of the operation/signal definition.

2.6 TAKING INHERITANCE INTO ACCOUNT

The advantage of using inheritance is that an object using the superclass interface need not know about the subclasses. But sometimes you explicitly want to use the subclasses. In the R&L example, the program partners want to limit the number of bonus points they give away; they have set a maximum of 10,000 points to be earned using services of one partner. The following invariant sums up all the points of all transactions for a partner. But it does not specify our intent because it does not differentiate between burning and earning transactions.

```
context ProgramPartner
inv: deliveredServices.transactions.points->sum() < 10,000
```

To determine the subclass to which an element of this collection of transactions belongs, we use the standard operation *oclIsTypeOf*, which takes a class, datatype, component or interface name as parameter. To retrieve from the collection all instances of this subclass, we use the *select* operation. We use the *collect* operation to retrieve from the collection of burning transactions a collection of points. These are the points that are summed by the operation *sum* and compared with the given maximum. So the correct invariant would be:

```
context ProgramPartner
inv: deliveredServices.transactions
        ->select( oclIsTypeOf( Earning ) ).points->sum() < 10,000
```

2.7 COMMENTS

In any model comments are neccessary to ease human understanding. This holds for an UML/OCL model too. It is good practice to accompany every OCL expression with a comment. A line comment is indicated by '--'. After this marker everything on the same line is considered to be comment. Comments that span more than one line can be enclosed between '/*' and '*/'.

For example, the previous invariant could have been accompanied by the following comments.

```
/* the following invariant states that the maximum number of points
   that may be earned by all services of a program partner is equal
   to 10,000
*/
context ProgramPartner
inv: deliveredServices.transactions -- all transactions
       ->select( oclIsTypeOf( Earning ) ) -- select earning ones
           .points->sum() -- add all points
               < 10,000 -- assert sum smaller than 10,000
```

2.8 LET EXPRESSIONS

To be done: Decide whether it is neccessary to explain the let expression in this chapter.

2.9 SUMMARY

In this chapter, we have shown how to write OCl expressions by example. The OCL description in this chapter is neither complete, nor precisely specified. Part 2 gives the complete specification of the OCL language.

To be done: Write better summary.

Chapter 3

Building Models with OCL

In this chapter we will show how a UML model can be augmented by OCL expressions, thus resulting in a model that is rich enough for automated tools to be used as input.

3.1 WHAT IS A MODEL?

Before we discuss how to build a model, we need to be sure what is meant by the word *model*. The word is used in many places and often has a different meaning. For instance, the figure depicting the R&L system in the previous chapter is often called a class *model*. A statechart is sometimes called a state *model*. Are these models two separate unrelated items, or should we consider them to be part of the same thing?

3.1.1 Definitions

In our view the answer to this question should be that both are views on the same model. We will use the word *model* to refer to a consistent, coherent set of model elements that have features and restrictions. For example, attributes and operations are features of classes, and derivation rules and invariants are restrictions on attributes and classes respectively. because most models are built using tools, we refer to the storage of all model elements as the *model repository.*

To indicate a certain view on that repository of model elements we use the word *diagram*. So the picture in figure 2-1 is a *diagram* showing part of the R&L *model*. Model elements can be shown in one, in many, or in no diagram at all. When an element is present in the model repository, it is part of the model. It need not be shown in any diagram. Nor need all of the features of a model element be shown when the element is present on a diagram. For instance, a class can be shown on two diagrams. On the first all attributes are shown, but no operations. On the second no attributes are shown, but all operations are.

Summarizing, the following are the definitions that will be used in this book.

A model is a consistent, coherent set of model elements that have features and restrictions.

A model repository is the storage of all model elements in an automated tool.

A diagram is a view on the model elements in a model.

3.1.2 Model is Consistent, Coherent Entirety

Any model must be an integrated, consistent, coherent entirety. It must be crystal clear how entities shown on one diagram relate to entities in other visible parts of the model. In the previous example it must be clear that in both class diagrams the same class is depicted. Likewise, it should be clear that the objects in an interaction diagram are instances of classes present in the model. These classes may be visible in one of the class diagrams. Another example of the relation between model elements shown in different diagrams is that a statechart and all its states and transitions always belongs to a class. This class must be present in the model and may be shown in other diagrams. The attributes and operations of this class shown in the class diagram may also be shown in the statechart, e.g. in a guard to a transition.

OCL expressions are often not shown in any diagram, but still they are part of the model. They are present in the underlying repository. Automated tools will use the OCL expressions in a model as well as any other information in the model. Therefore we must establish how the UML diagrams and OCL expressions connect. The relation between OCL expressions and entities shown on the diagrams must be clear. The link between an entity in a UML diagram and an OCL expression is called the *context definition* of an OCL expression.

3.1.3 The Context of an OCL Expression

The *context definition* of an OCL expression states for which model entity the OCL expression is defined. Usually this is a class, interface, datatype, or component. (For sake of convience, we will use the word *type* in the remainder of this book to indicate either class, interface, datatype, or component[1].) Sometimes the model entity is an operation, and rarely it is an instance. It is always a specific element of the model that is usually defined in a UML diagram. This element is called the *context* of the expression.

Next to the context, it is important to know the c*ontextual type* of an expression. The contextual type is the type that is, or contains the context. It is important because OCL expressions are evaluated for a single object, which is always an instance of the contextual type. To distinguish between the context and the

[1] In terms of the UML standard, this is called a *Classifier*, but we prefer to use the more intuitive word *type* in this book.

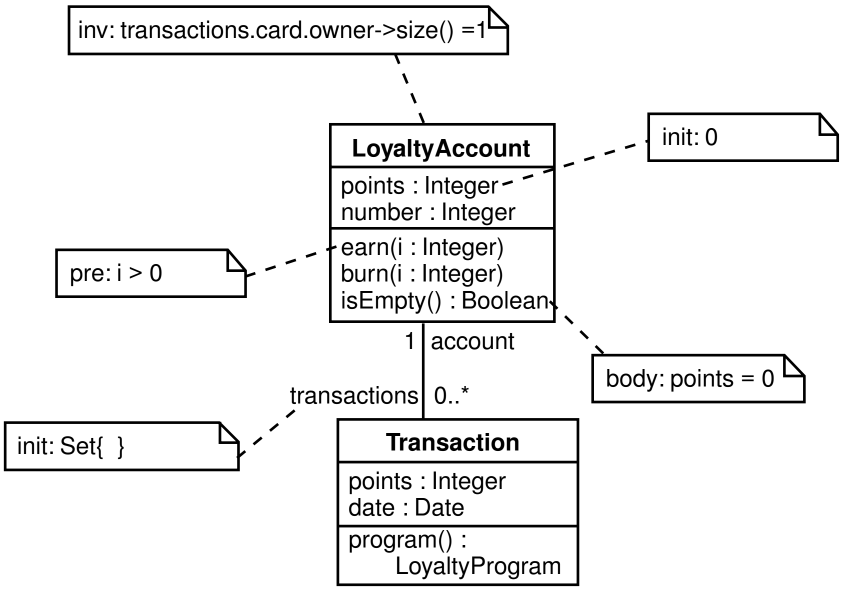

Figure 3-1 *OCL expressions and their context.*

instance for which the expression is evaluated, the latter is called the *contextual instance*. Sometimes it is necessary to refer explicitly to the contextual instance. The keyword *self* is used for this purpose [see section 6.1.2].

For example, the contextual type for all expressions in Figure 3-1 is the class *LoyaltyAccount*. The precondition (`pre: i>0`) has as context the operation *earn.* When it is evaluated, the contextual instance is the instance of *LoyaltyAccount* of which the operation has been called. The initial value (`init: 0`) has as context the attribute *points*. The contextual instance will be the instance of *LoyaltyAccount* that is newly created.

3.2 USE UML DIAGRAMS AS BASE

When starting a new model, the first step is to build a number of diagrams. The most important diagram in the model is the class diagram. Because any model is structured around classes or components, this is where all information in the model connects. The use of OCL strongly relies on the types (classes, datatypes, etc.) defined in a UML class diagram. This diagram should always be built first. Other diagrams can be built according to need and taste.

The second step is to identify where the need for explicitly stating extra information lies. Some of the obvious situations that call for additional information are:

- There are some elements that might be underspecified, for instance when an attribute has been defined as *derived*, but no derivation rule is given.
- There are business rules that must be incorporated in the model. This will give rise to a number of invariants, as shown in [Eriksson00].
- There is a need for a more precise definition of the interfaces in the system. The introduction of pre- and postconditions to operations is called for.
- Finally, some aspects of the diagrams might be ambiguous without stating invariants.

Below a number of situations are identified where adding expressions to the diagrams is neccessary to obtain a complete and unambigous model.

3.3 COMPLETING CLASS DIAGRAMS

The model depicted in a class diagram can be augmented with extra information in a large number of places. Most of the examples given in this section will augment the class diagram for the R&L system in Figure 2-1 in Chapter 2.

3.3.1 Derivation Rules

Often in a model derived attributes and associations are defined. A derived element does not stand alone. The value of a derived element must always be determined from other (base) values in the model. Leaving the way to derive the element value unspecified, results in an incomplete model. Using OCL the derivation can be expressed in a derivation rule. In the following example the value of a derived element *usedServices* is defined to be all services that have generated transactions on the account.

```
context LoyaltyAccount::usedServices : Set( Service )
derive: transactions.service->asSet()
```

Note that the question whether *usedServices* is an attribute or an association end should be answered from the UML diagram(s). In this case it was not shown in the diagram in Chapter 2, so we can not tell. Often derived attributes are not shown in the diagrams at all, but are defined by an OCL attribute definition (see section 3.9.1).

3.3.2 Initial Values

In the model information on the initial value of an attribute or association role can be given by an OCL expression. In the next examples the initial value for the attribute *points* is *0*, and for the association end *transactions* it is an empty set.

```
context LoyaltyAccount::points : Integer
init: 0

context LoyaltyAccount::transactions : Set(Transaction)
init: Set{}
```

Note the difference between an initial value and a derivation rule. A derivation rule in fact states an invariant: the derived element should always have the same value as the rule expresses. Whereas an initial value must hold only at the moment when the contextual instance is created. After that moment the attribute may have a different value at any point in time.

3.3.3 Body of Query Operations

The class diagram can introduce a number of query operations. Query operations are operations that have no side-effects. Execution of a query operation results in a value or set of values, without any alterations in the state of the system. Query operations can be introduced in the class diagram, but can only be fully defined by specifying the result of the operation. Using OCL the result can be given in a single expression, called a *body expression*. In fact, OCL is a full query language, comparable to SQL, as shown in [Akehurst01]. The use of body expressions is an illustration thereof.

The next example states that the operation *getCustomerName* will always result in the name of the owner of the card associated with the loyalty account.

```
context LoyaltyAccount::getCustomerName() : String
body: membership.card.owner.name
```

3.3.4 Invariants

Another way to augment a class diagram is by stating an invariant. The concept invariant is defined as follows.

> ***An** invariant **is a boolean expression that states a condition that must always be met by all instances of the type for which it is defined.***

An invariant is described using an boolean expression that evaluates to true if the invariant is met. The invariant must be true upon completion of the constructor, and completion of every public operation but not necessarily during the execution of operations. Incorporating an invariant in a model means that any system made according to the model is faulty when the invariant is broken. How to react when an invariant is broken is explained in section 4.6.6.

In the following example, all cards that generate transactions on the loyalty account must have the same owner.

```
context LoyaltyAccount
inv oneOwner: transactions.card.owner->asSet()->size() = 1
```

An invariant may be named, which can be useful for reference in an accompaning text. The above invariant is named *oneOwner.*

3.3.5 Preconditions and Postconditions

Pre- and postconditions to operations are yet another way to complete the model depicted in a class diagram. Because pre- and postconditions do not specify how the body of an operation should be implemented, they are an effective way to state a precise definition of the interfaces in the system. The concepts are defined as follows:

> ***A precondition is a boolean expression that must be true at the moment that the operation starts its execution.***
>
> ***A postcondition is a boolean expression that must be true at the moment that the operation ends its execution.***

Note that in contrast to invariants, which must always be true, pre- and postconditions need be true only at a certain point in time: before, and after execution of an operation, respectively.

We can use OCL expressions to specify either the pre- and postconditions of operations on all UML types, or the pre- and postconditions of use cases. The first option is explained in this section, the second option is explained in Section 3.8.1. In the following example a customer is enrolled in a loyalty program only when its *name* attribute is not empty. The postcondition of the *enroll* operation ensures that the new customer is included in the list of customers belonging to the loyalty program.

```
context LoyaltyProgram::enroll(newCustomer : Customer)
pre : newCustomer.name <> ‘’
post: customers = customers@pre->including( newCustomer )
```

Note that a even a simple postcondition like `myVar = 10` does not specify any statement in the body of the operation. The variable `myVar` may become equal to 10 in a large number of ways. Here are some example implementations in Java.

```
// use another variable
otherVar = 10;
```

```
// other statements
myVar = otherVar;

// use a calculation
myVar = 100/10;

// use another object
myVar = someElement;  // where the value of someElement is 10
```

The principle behind the use of pre- and postconditions is often referred to as the *design by contract* principle. Design by contract can be used within the context of any object-oriented development method. The following paragraphs describe the principle and its advantages.

Design by Contract

The definition of *contract* in the design by contract principle is derived from the legal notion of a contract: a univocal lawful agreement between two parties in which both parties accept obligations, and on which both parties can ground their rights. In object-oriented terms, a contract is a means to establish the responsibilities of an object clearly and unambiguously. An object is responsible for executing services (the obligations) if and only if certain stipulations (the rights) are fulfilled. A contract is an exact specification of the interface of an object. All objects that are willing to use the services offered are called *clients* or *consumers*. The object that is offering the services is called the *supplier*.

Although the notion of contract is derived from law practice, it is not completely analogous when employed in object technology. A contract is offered by a supplier independently of the presence of any client. But when a client uses the services offered in the contract, the client is bound to the conditions in the contract.

A contract describes the services that are provided by an object. For each service, it specifically describes two things:

- The conditions under which the service will be provided.
- A specification of the result of the service that is provided, given that the conditions are fulfilled.

An example of a contract can be found at most mailing boxes in the Netherlands:

> ***A letter posted before 18:00 will be delivered on the next working day to any address in the Netherlands.***

A contract for an express service is another example:

> ***For the price of two euros, a letter with a maximum weight of 80 grams will be delivered anywhere in the Netherlands within 4 hours after pickup.***

Table 3-1 shows the rights and obligations of both parties in the express delivery service example. Note that the rights of one party can be directly mapped to the obligations of the other party.

A contract can become much more complicated, for example, when it concerns the purchase of a house. The important thing is that the rights and obligations in a contract are unambiguous. In software terms we call this a *formal specification*. Both parties benefit from a clear contract.

- The supplier knows the exact conditions under which its services can be used. If the client does not live up to its obligations, the supplier is not responsible for the consequences. This means that the supplier can assume that the conditions are *always* met.
- The client knows the exact conditions under which it may or may not use the offered services. If the client takes care that the conditions are met, the correct execution of the service is guaranteed.

The interface that is offered by an object consists of a number of operations that can be performed by the object. For each operation, a contract can be envisioned. The rights of the object that offers the contract are specified by the preconditions. The obligations are specified by the postconditions.

If either party fails to meet the conditions in the contract, the contract is broken. When this happens, it is clear which party broke the contract: either the client did not meet the specified conditions, or the supplier did not execute the service correctly. Failure of a pre- or postcondition—that is, the condition is not true when it should be—means that the contract is broken. In Eiffel, the only language that implements the design by contract principle, an exception is raised when a pre- or postcondition fails. In this way the exception mechanism is an integral part of the design by contract principle.

Table 3-1 *Rights and obligations in a contract.*

Party	Obligations	Rights
Customer	Paying two euros	Letter delivered within 4 hours
	Supply letter with weight less than 80 grams	
	Specify delivery address within Netherlands	

Table 3-1 *Rights and obligations in a contract.*

Express service company	Deliver letter within 4 hours	Delivery addresses are within the Netherlands Receive two euros All letters weigh less than 80 grams

3.3.6 Messaging in Postconditions

An aspect of postconditions that is very usefull for defining interfaces, is the fact that a postcondition may state that a certain operation has been called. For example, in the Eclipse framework every file is contained in a project. Several builders, that are responsible for keeping associated files up to date, can be linked with the project. An example of a builder is the java compiler, that compiles a .java file in a .class file. When a file is saved, all builders are informed by calling their *incrementalBuild* operation. There is no need to uncover how the framework is taking care of this, but there is a need to tell Eclipse developers about this feature. A simple postcondition to the save operation of the class *File* will suffice:

```
context File::save()
post: self.project.builders->forAll( b : Builder |
                                     b^incrementalBuild() )
```

In general, when there is a need to specify dynamic aspects of a model element without revealing the actual implementation, the postcondition of the operations should contain message information. Note again that a postcondition does not specify any statement in the body of the operation. The message may have been sent in a large number of ways. The postcondition merely states that it is sent. Here are some example implementations in Java.

```
// get all builders and loop over the collection
Builders[] builders = getProject().getBuilders();
for(int i=0; i<builders.size; i++) {
    builders[i].incrementalBuild();
}

// let some other object take care of calling the builders
someObject.takeCareOfBuilders( getProject() );

// let the project take care of calling the builders
getProject().callBuilders();
```

3.3.7 Cycles in Class Models

In many class models, there are *cycles* in the sense that you can start at an instance, navigate through various associations, and come back to an instance of the same class. In the class diagram this is shown as a cycle of classes and associations. Quite often these cycles are the source of ambiguities.

The example used in section 1.3.2, reprinted in Figure 3-2, shows a class diagram with a cycle. In this model a *Person* owns a *House*, which is paid for by taking a *Mortgage*. The *Mortgage* takes as security the *House* that is owned by the *Person*. Although the model looks correct at first glance, it contains a flaw. Or better stated, it is imprecise and can give rise to ambiguities. The model allows a person to have a mortgage that takes as security a house that is owned by another person. This is clearly not the intention of the model. We can easily state the necessary invariant as:

```
context Person
inv: self.mortgage.security.owner = self
```

In general, cycles in a class model should be checked carefully, and any constraints on these cycles should be stated as an invariant on one of the classes in the cycle. Specially when the multiplicities in a cycle are higher than one, we need to carefully write the invariant or invariants.

3.3.8 Defining Derived Classes

In database systems the concept view exists. In a UML/OCL model a similar convenience concept exists. Here it is called a *derived class*. A derived class is a class

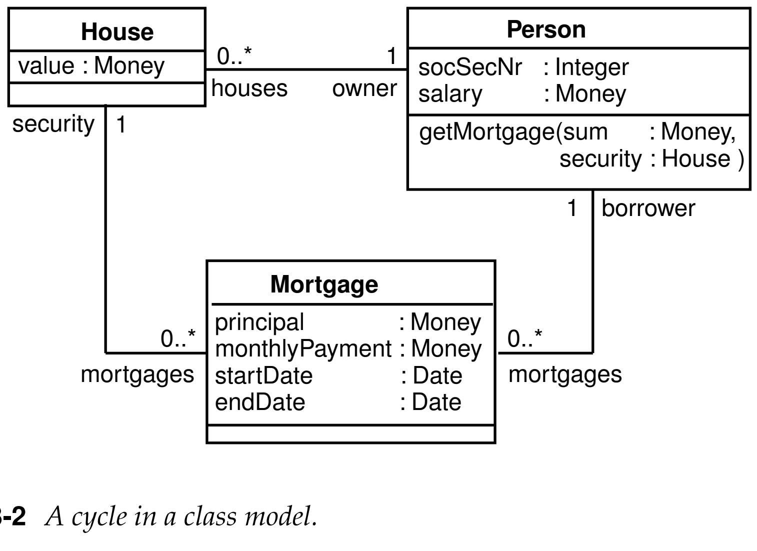

Figure 3-2 *A cycle in a class model.*

whose features can be derived completely from already existing (base) classes and other derived classes. The concept of derived classes has been introduced in [Blaha98], and formalized in [Balsters03].

For instance, in the R&L system it might be useful to define a derived class that holds a transaction report for a customer, as in figure 3-3. The attributes, association ends, and query operations of this class can be defined by the following expressions.

```
context TransactionReportLine::partnerName : String
derive: transaction.generatedBy.partner.name

context TransactionReportLine::serviceDesc : String
derive: transaction.generatedBy.description

context TransactionReportLine::points : Integer
derive: transaction.points
```

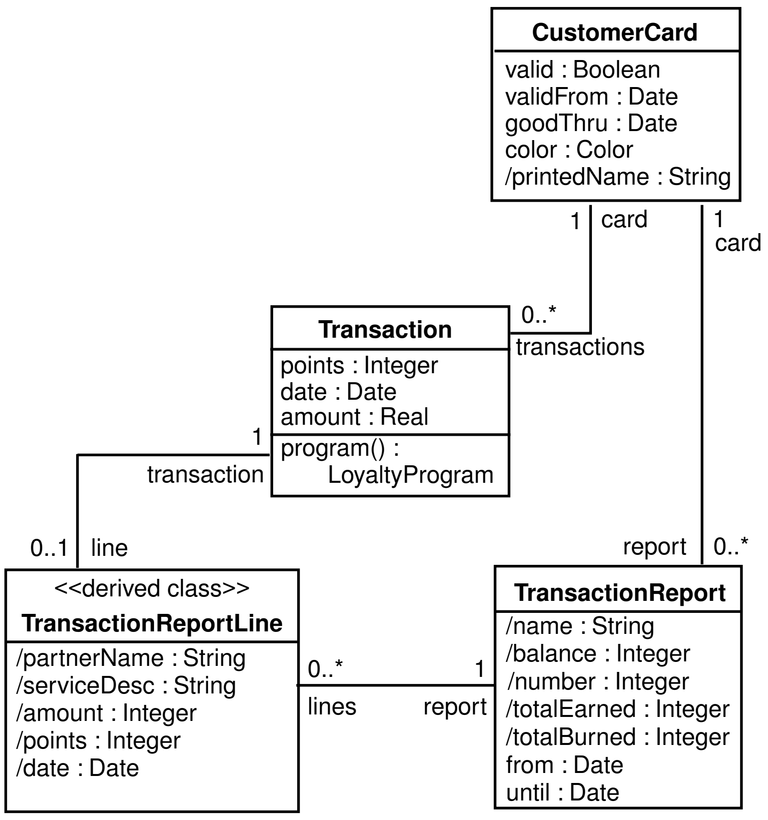

Figure 3-3 *A derived class.*

```
context TransactionReportLine::amount : Integer
derive: transaction.amount

context TransactionReportLine::date : Date
derive: transaction.date
```

The class *TransactionReport* is not completely derived. Its *from* and *until* attributes are normal attributes without derivation rules. The other attributes can all be derived.

```
context TransactionReport::name : String
derive: card.owner.name

context TransactionReport::balance : Integer
derive: card.Membership.account.points

context TransactionReport::number : Integer
derive: card.Membership.account.number

context TransactionReport::totalEarned : Integer
derive: lines.transaction->select( oclIsTypeOf( Earning ) )
            .points->sum()

context TransactionReport::totalBurned : Date
derive: lines.transaction->select( oclIsTypeOf( Burning ) )
            .points->sum()
```

To complete the definition of this class some invariants are needed. The *dates* invariant states that the transactions in this report should be transaction between the *from* and *until* dates. The *cycle* invariant states that the transactions in the lines of the report should indeed be transactions for this customer card.

```
context TransactionReport
inv dates: lines.date->forAll( d | d.isBefore( until ) and
                                   d.isAfter( from ) )

context TransactionReport
inv cycle: card.transactions->includesAll( lines.transaction )
```

3.3.9 Dynamic Multiplicity

Associations in a class diagram can sometimes be inprecise specifications of the system. This is the case when the multiplicity of the association is not fixed, but should be determined based on another value in the system. This is called *dynamic multiplicity.* An example has already been given in section 1.3.1, where the multiplicity of the association between *Flight* and *Passenger* was given by the *numberOfSeats* on the *Airplane*.

3.3.10 Optional Multiplicity

An *optional multiplicity* of an association in a class diagram is often only a partial specification of what is really intended. Sometimes the optionality is free; that is, in all circumstances there can be either one, or no associated object. Quite often, an optional association is not really free. Whether an associated object can or must be present depends on the state of the objects involved. For example, in Figure 3-2, the optional association is not completely free. If a Person has a Mortgage, he or she must also own a House. This constraint can be specified by the following invariant:

```
context Person
inv: mortgages->notEmpty() implies houses->notEmpty()
```

In general, when an optional multiplicity is present in the class diagram, we must use OCL invariants to describe precisely the circumstances under which the optional association may be empty or not empty.

3.3.11 Or Constraints

The class diagram can contain an *or constraint* between two associations, as shown in figure 3-4. The meaning of this constraint is that only one of the potential associations can be instantiated at one time for any single object. This is shown as a dashed line connecting two or more associations (all of which must have at least one class in common) with the string *{or}* labeling the dashed line. The multiplicity of the associations must be optional, otherwise they cannot be empty.

Note that the visual representation of an *or* constraint is ambiguous in certain situations. This is the case when two associations between the same two classes have multiplicity optional at both ends of both associations. The visual *or* con-

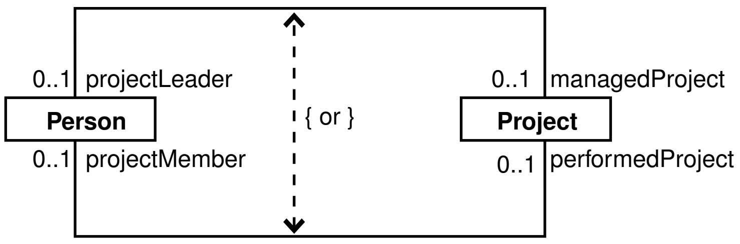

Figure 3-4 *Or constraint.*

straint can now be read in two directions. In one direction it can mean that one person has either a *managedProject* or a *performedProject*, but not both. In the other direction it can mean that one project has either a *projectLeader* or a *projectMember*, but not both. Therefore, two different interpretations are possible.

Although the choice of interpretation may seem obvious in this case, you can imagine the consequences if someone takes the wrong interpretation. Specifying the visual or constraint as an OCL constraint resolves the ambiguity problem. When you go for the first interpretation, you should augment the diagram with the following invariant:

```
context Person
inv: managedProject->isEmpty or performedProject->isEmpty
```

The invariant stating the second interpretation is:

```
context Project
inv: projectLeader->isEmpty or projectMember->isEmpty
```

This example shows the importance of formal constraints and illustrates how they ensure that a model is unambiguous.

3.4 COMPLETING INTERACTION DIAGRAMS

OCL expressions may also be used to complete UML interaction diagrams. In this section we explain how interaction diagrams that are used as instance diagrams can be modeled. When an interaction diagram is used for role modeling, it is not an instance diagram, but alike to a class diagram. It is not showing instances, but classes, interfaces, etc. For the sake of adding OCL expressions it can be treated as a class diagram.

The example that will be used in this section is the diagram in Figure 3-5. In the R&L system new transactions are introduced through an operation of *LoyaltyProgram* called *addTransaction*. This operation takes five parameters:

1. the number of the account the transaction was performed against, called *accNr* of type *Integer*,
2. the name of the program partner that delivered the service for which the transaction was performed, called *pName* of type *String*,
3. the identification of the service, called *servId* of type *Integer*,
4. the amount that paid for the service, called *amnt* of type *Real*, and
5. the date on which the transaction did take place, called *d* of type *Date*.

The *addTransaction* operation implements the following algorithm. First the correct service is selected. This service calculates the points earned or burned by the transaction and creates a transaction object of the right type. This object is added

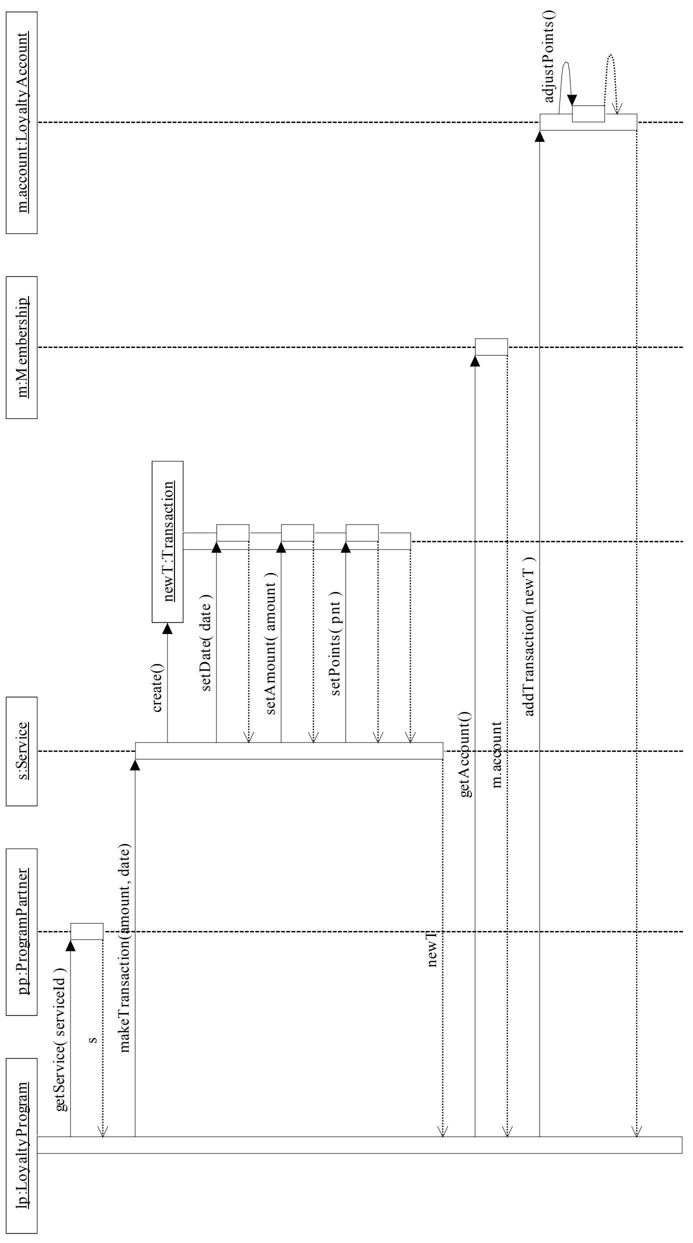

Figure 3-5 *OCL expressions in an interaction diagram.*

to the service's association end *transactions*. Next the correct account is selected. The newly created transaction is added to the transactions associated with this account, meanwhile adjusting the balance of the account. Finally the correct card is selected and the new transaction is added to the transactions associated with this card. The algorithm is depicted in the sequence diagram.

As an elaborate example of an OCL postcondition, the *addTransaction* operation is also specified using a postcondition in the following OCL expression.

To be done: addTransaction as described here is the operation on LoyatltyProgram, which is the whole sequence diagram. The addTransaction() on LoyaltyAccouint should be renamed to avoid confusion.

```
context LoyaltyProgram::addTransaction( accNr: Integer,
                                        pName: String,
                                        servId: Integer,
                                        amnt: Real,
                                        d: Date )
post: let acc : LoyaltyAccount =
             Membership.account->select( a | a.number = accNr ),
          newT : Transaction =
              partners-> select(p | p.name = pName)
                 .deliveredServices
                    ->select(s | s.serviceNr = servId)
                      .transactions
                         ->select( date = d and amount = amnt ),
          card : CustomerCard =
              Membership->select( m |
                           m.account.number = accNr ).card
      in acc.points = acc.points@pre + newT.points and
         newT.oclIsNew() and
         amnt =  0 implies newT.oclIsTypeOf( Burning )        and
         amnt >  0 implies newT.oclIsTypeOf( Earning )        and
         acc.transactions - acc.transaction@pre = Set{ newT } and
       card.transactions - card.transaction@pre = Set{ newT }
```

3.4.1 Instances

In an interaction diagram instances are shown. Because of our requirement that a model must be internally consistent, each instance shown must be of a type declared in another diagram in the model. Furthermore, although there are no rules in the UML that state this explicitly, the target of a message in an interaction diagram must be known and visible to the source of the message. This might be indicated in the diagram using the name by which the target is known to the source. Formally this has no meaning at all, but the human reader will probably find it useful. Another way to express the same fact, is to use your own names for the instances, and accompany the diagram with OCL expressions that states the relation between the instances.

In figure 3-5 we can see that the instance of *LoyaltyAccount* is named *m.account*, where *m* references the *Membership* instance in the diagram. To state the relationship between the other instances in the diagram the following expressions can be used:

```
lp.partners->includes(pp) and pp.name = pName
pp.deliveredServices->includes( s ) and s.serviceNr = servId
lp.Membership->includes( m ) and m.account.number = accNr
```

3.4.2 Conditions

A message in a sequence or collaboration diagram can have an attached condition that specifies in what circumstances the message is sent. This condition can be written as an OCL expression.

Because an interaction diagram is an instance diagram, the context of the OCL expression is an instance, not a type. This is a rare case, therefore extra attention has to be paid to determine what is the contextual type, and what is the contextual instance. Well, the contextual instance is the instance that sends the message, the source of the message. The contextual type is the type of this instance.

An example can be found in Figure 3-6 that extends the sequence diagram in figure 3-5. The service object *s* creates a transaction of the right type, depending on the *amnt* parameter of the *addTransactions* operation. The conditions are written as valid OCL expressions, given the fact that the diagram specifies an operation, and therefore the parameters of the operation can be used.

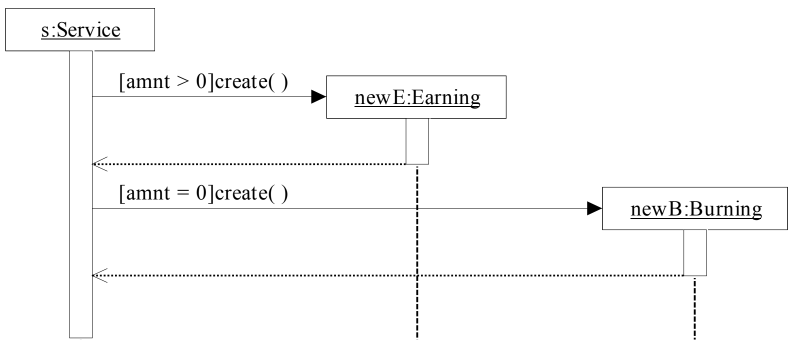

Figure 3-6 *Extended interaction diagram.*

3.4.3 Actual Parameter Values

A message in an interaction diagram is not the definition of an operation or signal, but instead represents an operation call or the sending of a signal. Both operations and signals can take parameters. The parameters to a message are not the formal parameters but actual values. For instance, when an operation is defined, its name, and the name and type of every parameter is given, as in *set(i: Integer)*. When an operation is called, its parameters are substituted by actual values, as in *set(235)*.

To specify the actual value of a parameter to a message, you may use an OCL expression. In the example in figure 3-5 the value to be substituted for the parameters of operation *addTransaction* is given by *newT*, a reference to the newly created transaction instance.

Note that a message to an object in an interaction diagram must conform to an operation in the type of the target object, or to a signal that has been elsewhere defined. Conformance here means that the name of the operation is the same, and the parameters in the message are of the types indicated for the parameters of the operation. Likewise, the result message in an interaction diagram (a dotted arrow in the sequence diagram, or an assignment in the collaboration diagram) must be of the type indicated as resulttype of the called operation.

3.5 COMPLETING STATECHARTS

In UML statecharts OCL expressions may be used in a number of ways. In all cases, the contextual type is the class to which statechart belongs, and the contextual instance is the instance for which a transition in the statechart fires.

The example in this section is a simple process control system in a factory that produces bottles with certain liquid contents. The classes used are depicted in Figure 3-7. The process goes as follows. A line object takes a new bottle object from its stock, and triggers a filler object to fill this bottle. The bottle monitors its contents and messages the filler when it is full. The filler then triggers the line to move the bottle to the capper, which caps the bottle. Figure 3-8 contains the statechart for class *Filler*, and Figure 3-9 contains the statechart for class *Bottle*.

3.5.1 Guards

OCL expressions can be used to express guards in statecharts. A guard is a condition on a transition in a statechart. A guard is often included in the statechart diagram itself. In that case it is written between square brackets ('[' and ']') after the event that triggers the transition. In Figure 3-8 you can find a guard on the transi-

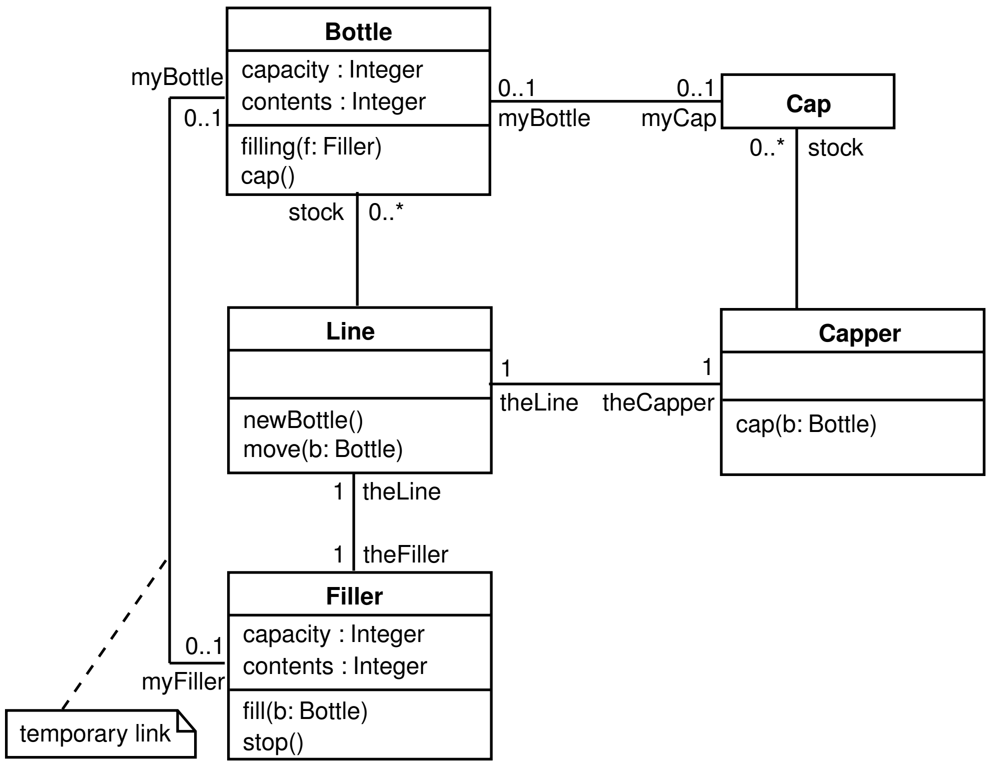

Figure 3-7 *Class diagram for the bottle filling system.*

tion from *stopped* to *filling*. The *Filler* object will only change state if it contains enough liquid.

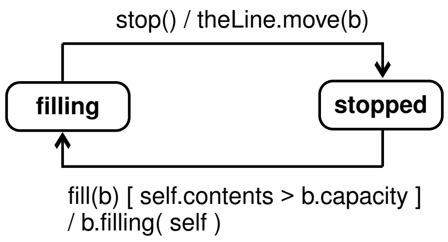

Figure 3-8 *Filler statechart.*

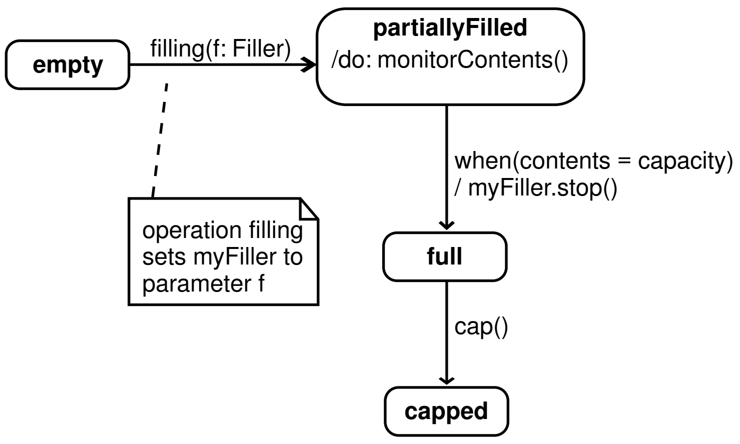

Figure 3-9 *Bottle statechart with change event.*

3.5.2 Target of Actions

An action in a statechart represents either an operation call, or the sending of an event. Actions may be coupled to transitions or states in a statechart. When coupled to a transition, the action is executed if and when the transaction fires, and is targeted at a specific object, or set of objects. When coupled to a state, the action is executed when the state is entered, or when a state is left, or when an indicated event occurs. In that case usually the target is the self object.

To identify the target of an action, whether coupled to a transition or a state, an OCL expression can be used. For example, the action *theLine.move(b)* in Figure 3-8 has as target the *Line* object linked to the contextual instance of *Filler,* and the action *myFiller.stop()* in Figure 3-9 has as target the *Filler* object linked to the contextual instance of *Bottle*.

3.5.3 Actual Parameter Values

Because actions represent operation calls or the sending of events, and both can have parameters, actions in statecharts can have parameters. Like the parameters to messages in an interaction diagram, these are actual values, not the formal parameters of the corresponding operation or send event. The value of such a parameter can be specified using an OCL expression, in which case the context of the expression is the transition or state to which the action is coupled. In Figure 3-8 the both actions have an actual value as parameter.

3.5.4 Change Events

A *change event* is an event that is generated when one or more attributes or associations change value. The event occurs whenever the value indicated by the expression changes from false to true. A change event is denoted by the keyword *when*, parameterized with an OCL expression. For example, in the state transition diagram of the *Bottle* class in Figure 3-9, a change event is attached to the transition from state *partiallyFilled* to state *full*. The transition takes place as soon as the condition *contents = capacity* becomes true. No externally generated event is needed.

Note that a change event is different from a guard condition. A guard condition is evaluated once when its event fires. If the guard is false, the transition does not occur, and when no other transition for the same event is specified, the event is lost. When a change event occurs, a guard can still block any transition that would otherwise be triggered by that change.

3.5.5 Restrictions on States

Usually there are restrictions on values of links and attributes when an object is in a certain state. These should be specified explicitly. For instance, for the class Bottle the following invariants should hold.

```
context Bottle
inv: (self.oclInState(capped) or self.oclInState(full))
                                        implies contents = capacity
inv: (self.oclInState(empty) implies contents = 0
inv: self.oclInState(capped) implies myCap->notEmpty()
inv: self.oclInState(partiallyFilled) implies myFiller->notEmpty()
```

3.6 COMPLETING ACTIVITY DIAGRAMS

The way in which expressions may be used in UML activity diagrams, is explained in this section. The model elements that may be used in an expression in an activity diagram, are all attributes, all query operations, all states, and all associations of the contextual type. What the contextual type is, depends on the position of the expression in the diagram.

TBD.

3.7 COMPLETING COMPONENT DIAGRAMS

TBD.

3.8 COMPLETING USE CASES

In UML usecases pre- and postconditions may be used. These may also be written using OCL. Because use cases are an informal way of stating requirements, and OCL is a formal language, some adjustments need to be made when the pre- and postconditions of use cases are to be defined using OCL expressions.

3.8.1 Pre- and Postconditions

Although use cases can be considered to be operations defined on the complete system, we cannot identify the complete system as a type, because it is neither a class, an interface, a datatype, or component. Therefore, the pre- and postconditions of a use case have no contextual type, or contextual instance. As a consequence the keyword *self* can not be used.

Another consequence is that it is not clear which model elements may be referenced in the expression. Normally, all elements held by the contextual type may be used. Here, there is no contextual type. Therefore, the model elements that may be used must be explicitly stated. This can be done by formalizing the types mentioned in the use case, e.g. Customer and Order, in an accompanying class diagram, and adding to the use case template a section called (for instance) 'concerns' with a list of variable declarations, e.g. *newCustomer : Customer, newOrder : Order.*

Yet another consequence is that we cannot write a context definition, as there is no contextual type to be referenced. This is not a problem. The OCL expressions may be directly included in the use case at the position indicated by the use case template.

As an example a use case is given for the R&L system. It tells how customer cards are upgraded and how unvalid cards are removed from the system.

```
Use case title: Check customer cards for loyalty program
Summary: On a regulas basis the cards of all participants in a loy-
alty program are checked. Those that have been unvalidated are re-
moved, and those have been used often are upgraded to the next
service level.
Primary actor: user
Uses: - the class diagram for the Royal & Loyal system as shown in
        figure 2-1
      - lp of type LoyaltyProgram, which is the loyalty program for
        which the cards are checked
      - upgradeLimit of type Integer, which indicates how many
        points must have been earned before a card is upgraded
      - fromDate of type Date, which gives the date from which the
        calculate points to see if an upgrade is required
Precondition: none
Main succes scenario:
```

```
1. The actor starts the use case and gives the upgradeLimit. This
value must be larger than 0.
2. The system selects all cards that have been unvalidated or have
a goodThru date that lies before today, and shows them to the user.
3. The actor checks the unvalidated cards, and marks any card that
must not be removed by giving a new goodThru date.
4. The system removes all unvalidated cards, except the ones that
have received a new goodThru date.
5. The actor gives the fromDate. This date must be before today.
5. The system calculates for all remaining cards the number of
points earned in the period starting with fromDate until today, and
shows the cards for which this number is higher than the upgrade-
Limit.
6. The actor checks the cards to be upgraded, and marks any card
that must not be upgraded.
7. The system upgrades the selected cards.
Extensions:
   -- indicate what is to be done when the upgradeLimit and fromDate
    -- do not uphold their conditions
Postcondition:
-- all goodThru dates are in the future
lp.participants.cards.goodThru->forAll( d | d.isAfter( Date::now )
and
-- all cards that have earned enough points and have not been marked
-- otherwise are upgraded
let upgradedCards : Set( CustomerCard ) =
           lp.participants.cards->select( c |
                  c.getTotalPoints( fromDate ) >= upgradeLimit
                  and c.markedNoUpgrade = false )
in
    upgradedCards->forAll( c |
           lp.levels->indexOf( c.Membership.currentLevel ) =
           lp.levels->indexOf( c.Membership.currentLevel@pre ) + 1
```

3.9 MODELING STYLES

Apart from mending apparant flaws in the model, OCL expressions can also be used to express the same information in a different manner. There are various styles of modeling.This section explains some of the differences in style.

3.9.1 Definitions of Attributes or Operations

Attributes or operations may be defined in the class diagram by adding them to a type, but they may also be defined by an OCL expression. In that case the new attribute or operation need not be shown in the diagrams. In the following example two attributes, *wellUsedCards* and *loyalToCompanies*, and one operation, *cards-ForProgram*, are defined.

```
context Customer
def: wellUsedCards : Set( CustomerCard )
        = cards->select( transaction.points->sum() > 10,000 )
def: loyalToCompanies : Bag( ProgramPartner )
        = programs.partners
def: cardsForProgram(p: LoyaltyProgram) : Set(Cards)
        = p.membership.card
```

The expression following the equal sign in an attribute definition indicates how the value of the attribute must be calculated. It is a derivation rule (see Section 3.3.1). For example, the newly defined attribute *loyalToCompanies* is always equal to the collection of program partners that are associated with the programs the customer is enrolled in. Whereas, the attribute *wellUsedCards* is always equal to the set of cards for which the total of the points earned with the transactions carried out with that card is larger than 10,000.

The expression following the equal sign in an operation definition states the result of the operation. It is a body expression (see Section 3.3.3). Note that operations defined by an OCL expression are query operations. They can not have side-effects. In the above example, the operation *cardsForProgram* will always result in the set of cards issued for the loyalty program *p* given as parameter to the operation.

3.9.2 The Subset Constraint

The class diagram may contain *subset constraints*, as shown in Figure 3-10. The meaning of this constraint is that the set of links for one association is a subset of the set of links for the other association. In Figure 3-10, *flightAttendants* is a subset of *crew*.The singleton set *pilot* is also a subset of *crew*.

By showing all subset constraints in the diagram, it may become cluttered and difficult to read. In that case you may choose to specify the subset constraints

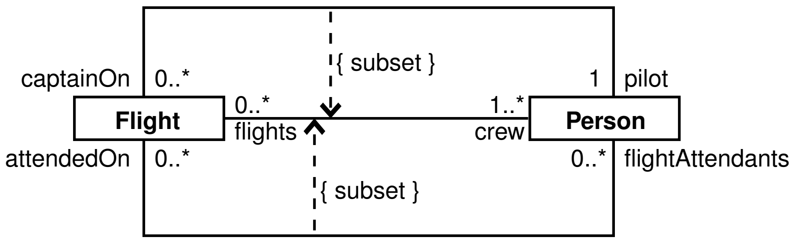

Figure 3-10 *A subset constraint.*

using OCL expressions. The two subset constraints shown in Figure 3-10 are identical to the following invariants on *Flight* and *Person*:

```
context Flight
inv: self.crew->includes(self.pilot)
inv: self.crew->includesAll(self.flightAttendants)

context Person
inv: self.flights->includes(self.captainOn)
inv: self.flights->includesAll(self.attendedOn)
```

3.9.3 Adding Inheritance Versus Invariants

During modeling, we often encounter situations in which we add detail to the class model to specify the real-world situation precisely. For example, suppose we have a simple class diagram of a guitar, in which a guitar has a number of guitar strings. Furthermore, there are two types of guitars: electric and classic. We also have two types of guitar strings. Each guitar type has its own kind of guitar strings.

The association between *Guitar* and *GuitarString* specifies that a *Guitar* has *GuitarString*'s. The association between *ClassicGuitar* and *PlasticString* is a redefinition of the same association, which constrains a *ClassicGuitar* to have *PlasticString*'s only. The generalization between the associations shows that the lower association is a specialization of the upper, more general association. The association between *ElectricGuitar* and *MetalString* is also a redefinition of the top association.This situation is depicted in figure 3-11.

As you can see, the model becomes comparatively complex. The model can be simplified using invariants. The two specializations of the upper association can be captured in two invariants on the two types of guitars, as in the class model

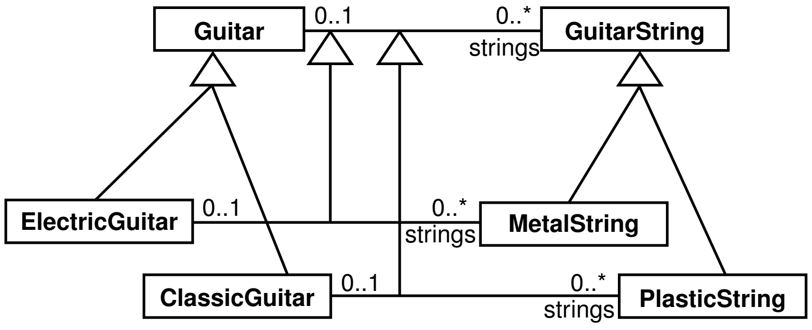

Figure 3-11 *Model with specializations.*

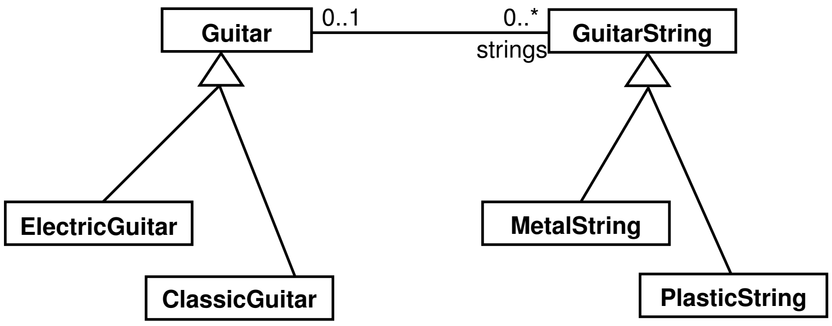

Figure 3-12 *Model without specialized associations.*

shown in figure 3-12. The visual class model becomes more readable, while the level of detail is retained. The invariants are:

```
context ClassicGuitar
inv: strings->forAll(oclType = PlasticString)

context ElectricGuitar
inv: strings->forAll(oclType = MetalString)
```

The preceding model can be simplified even more by removing the subclasses of *Guitar*, *GuitarString*, or both. If the main reason for having the subclasses is to distinguish between different kinds of strings for different guitars, this simplification makes sense. Depending on the situation, this simplification may result in either figure 3-13 or figure 3-14.

Here are the invariants for the class model in figure 3-13:

```
context ClassicGuitar
inv: strings->forAll(type = StringType::plastic)
```

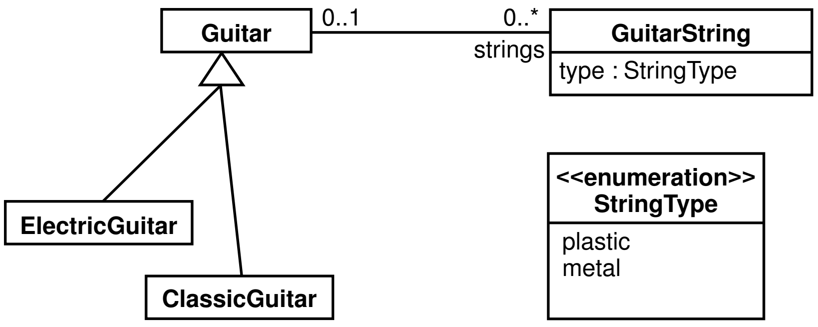

Figure 3-13 *Model without some subclasses.*

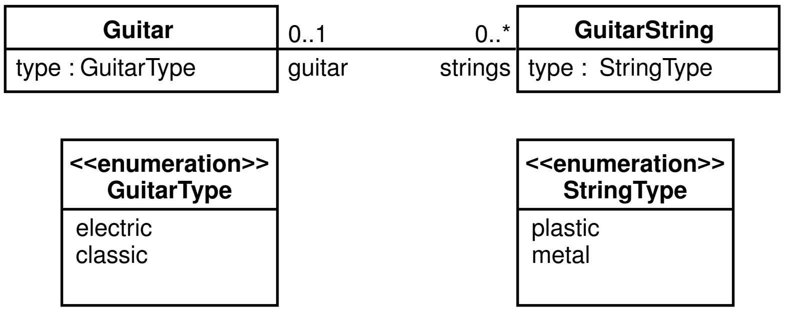

Figure 3-14 *Class model without subclasses.*

```
context ElectricGuitar
inv: strings->forAll(type = StringType::metal )
```

Here are the invariants for the class model in figure 3-14:

```
context Guitar
inv: type = GuitarType::classic implies
                  strings->forAll(type = StringType::plastic)

context Guitar
inv: type = GuitarType::electric implies
                  strings->forAll(type = StringType::metal )
```

The deciding question in the trade-off that must be made in these circumstances is which of the solutions will be best in your situation. Figure 3-14 keeps the model simple, because there is no need for subclasses. If there are no attributes or behavior specific to the subclasses, this is a good solution. In figure 3-13, we have a more elaborate model that is suitable when there are attributes or operations specific to the different subclasses. In figure 3-11, we have an elaborate model with probably too much detail. This option is desirable if you want to show all details in the visual diagram. In general, showing your model in graphical or visual form is best for giving a good overview, whereas a textual form is good for adding detail. The art is to find the right balance, and that depends on the intended use of the model and the intended audience.

3.10 TIPS AND HINTS

This section gives some tips and hints on how to write meaningfull OCL expressions.

3.10.1 Avoid Complex Navigation Expressions

Using OCL, we can write long and complex expressions that navigate through the complete object model. We could write all invariants on a class model starting from only one context, but that does not mean that it is good practice to do so.

Any navigation that traverses the whole class model creates a coupling between the objects involved. One of the essential notions within object orientation is encapsulation. Using a long navigation makes details of distant objects known to the object where we started the navigation. If possible, we would like to limit the object's knowledge to only its direct surroundings, which are the properties of the type, as described in Section 8.1.

Another argument against complex navigation expressions is that writing, reading, and understanding invariants becomes very difficult. It is hard to find the appropriate invariants for a specific class, and maintaining the invariants when the model changes becomes a nightmare.

Consider the following expression, which specifies that a *Membership* does not have a *loyaltyAccount* if you cannot earn points in the program.

```
context Membership
inv: programs.partners.deliveredServices->
   forAll(pointsEarned = 0) implies loyaltyAccount->isEmpty
```

Instead of navigating such a long way, we might want to split this. First, we define a new attribute *isSaving* for *LoyaltyProgram*. This attribute is true if points can be earned in the program.

```
context LoyaltyProgram
def: isSaving : Boolean =
          partners.deliveredServices->forAll(pointsEarned = 0)
```

The invariant for *Membership* can use the new attribute rather than navigate through the model. The new invariant looks much simpler:

```
context Membership
inv: program.isSaving implies loyaltyAccount->isEmpty
```

3.10.2 Choose Context Object Wisely

By definition, invariants apply to a type, so it is important to attach an invariant to the right type. There are no strict rules that can be applied in all circumstances, but some guidelines will help.

- If the invariant restricts the value of an attribute of one class, the class containing the attribute is a clear candidate.

- If the invariant restricts the value of attributes of more than one class, the classes containing any of the attributes are candidates.
- If a class can be appointed the responsibility for maintaining the constraint, that class should be the context. (This guideline uses the notion of responsibility-driven design [Wirfs-Brock90].)
- Any invariant should navigate through the smallest possible number of associations.

Sometimes it is a good exercise to describe the same invariant using different classes as context. The constraint that is the easiest to read and write is the best one to use. Attaching an invariant to the wrong context makes it more difficult to specify and more difficult to maintain.

As an example, let's write an invariant in several ways. The invariant is written for the model in figure 3-15 and states the following: Two persons who are married to each other are not allowed to work at the same company. This can be expressed as follows, taking *Person* as the contextual object:

```
context Person
inv: wife.employers->intersection(self.employers)->isEmpty
     and
     husband.employers->intersection(self.employers)->isEmpty
```

This constraint states that there is no company in the set of employers of the wife or husband of the person that is also in the set of employers of the person. The constraint can also be written in the context of *Company,* and that creates a simpler expression:

```
context Company
inv: employees.wife->intersection(self.employees)->isEmpty
```

In this example the object that is responsible for maintaining the requirement will probably be the *Company.* Therefore, *Company* is the best candidate context for attaching the invariant.

3.10.3 Avoid *allInstances*

The *allInstances* operation is a predefined operation on any modeling element that results in the set of all instances of the modeling element and all its subtypes in

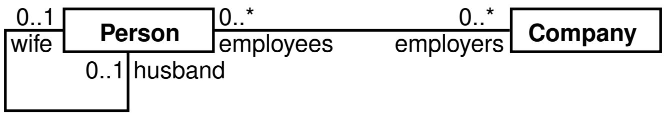

Figure 3-15 *Persons working for Companies.*

the system. An invariant that is attached to a class always applies to all instances of the class. Therefore, you can often use a simple expression as invariant instead of using the *allInstances* predefined operation. For example, the following two invariants on *Person* are equivalent, but the first is preferred.

```
context Person
inv: parents->size <= 2

context Person
inv: Person.allInstances->forAll(p | p.parents->size <= 2)
```

The use of *allInstances* is discouraged, because it makes the invariant more complex. As you can see from the example, it hides the actual invariant. Another, more important, reason is that in most systems, apart from database systems, it is difficult to find all instances of a class. Unless there is an explicit tracking device that keeps record of all instances of a certain class as they are create and deleted, there is no way to find them. Thus there is no way to implement the invariant using a programming language equivalent of the *allInstances* operation.

In database systems, the *allInstances* operation can be used for types that represent a database table. In that case the operation will result in the set of objects representing all record in the table.

3.10.4 Split *and* Constraints

Constraints are used during modeling, and they should be as easy to read and write as possible. People tend to write long constraints. For example, all invariants on a class can be expressed in one large invariant, or all preconditions on an operation can be written as one constraint. In general, it is much better to split a complicated constraint into several separate constraints. It is possible to split an invariant at many boolean *and* operations. For example, we can write an invariant for *ProgramPartner* as

```
context LoyaltyProgram
inv: partners.deliveredServices->forAll(pointsEarned = 0)
     and
     membership.card->forAll(goodThru = Date.fromYMD(2000,1,1))
     and
     customer->forAll(age() > 55)
```

This invariant is completely valid and useful, but we can rewrite it as three separate invariants, making it easier to read.

```
context LoyaltyProgram
inv: partners.deliveredServices->forAll(pointsEarned = 0)
```

```
context LoyaltyProgram
inv: membership.card->forAll(goodThru = Date::fromYMD(2000,1,1))

context LoyaltyProgram
inv: customer->forAll(age() > 55)
```

The advantages of this splitting approach are considerable.

- Each invariant becomes less complex and therefore easier to read and write.
- When you are discussing whether an invariant is appropriate, the discussion can be more focused and precise on the invariant concerned, instead of having to deal with the large invariant as a whole.
- When you are checking and finding broken constraints in an implementation, it can point more precisely to the part that is broken. In general, the simpler the invariant, the more localized the problem.
- The same arguments hold for pre- and postconditions. When a precondition is broken during execution, the problem can be pinpointed much more effectively when you are using small separate constraints.
- Maintaining simpler invariants is easier. If you need to change one condition, then you need change only one small invariant.

3.10.5 Use the *collect* Shorthand

The shorthand for the *collect* operation on collections, as defined in Section 9.3.11, has been developed to streamline the process of reading navigations through the class model. You can read from left to right without the distracting *collect* operations. We recommend that you use this shorthand whenever possible.

For example:

```
context Person
inv: self.parents.brothers.children->notEmpty()
```

This is much easier to read than

```
context Person
inv: self.parents->collect(brothers)
                               ->collect(children)->notEmpty()
```

Both invariants are identical, but the first one is easier to understand.

3.10.6 Make Tuples Explicit

To be done: Write it!!!

3.10.7 Always Name Association Ends

To be done: Write it!!!

3.11 SUMMARY

In this chapter ... TBD

Chapter 4

Implementing OCL

This chapter describes how OCL expressions can be implemented in code. Although in the examples the Java language is used, the principles explained can be applied using any object-oriented programming language as target language. The goal of this chapter is to show how OCl epxressions map to code. This process can be done manually, but also by OCL tools. Several tools are available that can translate OCL to code. See [REF TO TOOLS].

4.1 IMPLEMENTATION PROCESS

UML and OCL are not programming languages but specification languages, i.e. a UML/OCL model specifies how a system should be structured and what it should do, not how it should be implemented. These models will usually not be directly compiled and executed. There is, however, a strong connection between the implementation of a system and the model that specifies it. Certainly within the context of MDA the connection between the model and its implementation(s) should be stated clearly.

In this chapter we will explain how to generate Java code from a combined UML/OCL model. Our focus will, of course, be on how to build code from OCL expressions. But, as the UML class diagram must be built before writing OCL expressions, the first step in implementing the model is to define the implementation of the model elements defined in the class diagram(s). In the next step, the OCL expressions can be translated using the implementation for the model elements. For instance, when an attribute has been implemented by a private class member and a get and a set operation for that member, the implementation of an OCL expression referencing this attribute must use the corresponding get operation.

Furthermore, the code fragments translated from the OCL expressions must be used according to their function in the model (invariant, attribute definition, derivation rule, etc.). The code fragments must be placed in the implementation in such a manner that they fulfil their purpose.

The order of steps in the implementation process is:

1. Define the implementation of the UML model elements.
2. Define the implementation of the OCL standard library.
3. Define the implementation of the OCL expressions.
4. Place the code fragments implementing the OCL expressions correctly in the code for the model elements.
5. For invariants, and pre- and postconditions, decide when to check them, and what to do when the check fails.

Because OCL is a declarative language, it doesn't specify how to calculate the value of an expression. The translation from OCL to code that we show in this chapter therefore by definition includes a number of implementation choices. These are by no means the only ones or the best ones. In other circunstance, you might need to make other choices.

The rest of this chapter is structured according to this order of steps.

4.2 IMPLEMENTING UML MODEL ELEMENTS

About the implementation of user defined types we can be short: use the manner of implementation you prefer. The only thing you need to take care of is maintaining the connection with the implementation of the OCL expressions. For instance, if an association is implementation by two class members, each placed in one of the classes at both sides of the association, the implementation of an OCL expression that refers to the association must refer to the right class member.

In this chapter we will assume that the following rules are used to implement user defined types. In our R&L example there are no components or datatypes, therefore no rule for implementing them is given.

1. Every user defined class is implemented in one Java class.
   - Every operation on the class in the model is implemented by one operation in the Java class.
   - Every attribute (private, protected, or public) of the class in the model is implemented by a private class member and a get and set operation (private, protected, or public respectively) in the Java class. The get operation always has the form *getAttributename*(). The set operation always has the form *setAttributeName(AttributeType newValue).*
   - Every association end (private, protected, or public) of the class in the model is implemented by a private class member and a get and set operation (private, protected, or public) in the Java class. The name of the class member is the name of the role at that end. When the multiplicity of the association end

is more than one, the type of the class member is either the Java implementation of the OCL *Set* or *OrderedSet*, depending on the ordering of the association. In that case an *add* and *remove* operation are added to the implementation of the contextual type, which can add or remove an element from the class member.

- Every state defined in a statechart for that class is implemented by a boolean class member. Extra invariants are added to ensure that an instance of the type can may be in only one state at a time. When a state has substates, the invariants are adjusted to accommodate them. The class member representing the parent state and one of the class members representing the substates may be true at the same time.
- Every event defined in a statechart for that class is implemented by an operation. This operation implements the reaction of the instance on the event. The reaction must take into account the state the instance is in, any conditions on transitions that are triggered by the event, and any actions connected to the transitions or to the begin and end state of the transitions. In other words, we assume that a statechart has a protocol interpretation.

2. Every enumeration type is implemented by one Java class, holding static public class members for each value.
3. Every interface is implemented by one Java interface.

As an example the implementation of some classes from the Royal&Loyal system is given in REF.

4.3 IMPLEMENTING THE OCL STANDARD LIBRARY

Preferably the OCL standard library would be implemented by a library in the target programming language. Unfortunately, this is not possible for most languages, because the implementations of many of the standard operations that loop over a collection should take a piece of code as parameter. Only few programming languages support this. For instance, it would be very straightforward if we could translate the next OCL expression into one single Java expression.

```
context Customer
inv: cards->select( valid = true )->size() > 1
```

The Java expression would need to take the code that represents the *valid = true* part as parameter to the select operation.

```
// the following is incorrect Java
getCards().select( {valid == true} ).size() > 1
```

When you work in Smalltalk or any other language that supports this, you are encourage to take advantage of this aspect of the language to define your own standard library. In the following we will use the Java language and this influences the design choices we make.

4.3.1 OCL Basic Types

The mapping of the predefined basic types and model types is, although not straightforward, easy because its problems are well-known. The predefined basic OCL types, their literals and operations, must be mapped to basic types in the target language. For instance, the Java programming language offers the types *float* and *double*, whereas OCL has only one *Real* type. A choice needs to be made how the *Real* type is to be implemented by Java code, either by *float* or by *double*. Because the basic types of most languages are very similar, this mapping will not constitute many problems. Table 4-1 shows the mapping that we will use

After defining this mapping we need to map all operations from these OCL types toi operations in Java. Since Java does not provide all operations, we will need to define a special library that will hold our own operations. This can be done by defdining one library class in which we define each required operations as a static Java method.

Sometimes mapping the operations defined on every OCL type, like *oclIsTypeOf* and *oclIsKindOf,* are slightly more complicated. Usually, these operations can be mapped on similar constructs defined in the target language on the root class. In Java the keyword *instanceof* can be used to implement *oclIsKindOf,* and the *getClass* method of the root class *Object* can be used to implement *oclIsTypeOf.*

OCL type	Java type
Integer	int
Real	float or double
String	String
Boolean	boolean
OclType	Class
OclAny	Object

Table 4-1 *Mapping of basic types from OCL to Java.*

4.3.2 OCL Tuples

Because tuples are types that are defined on the fly, as it were, there are no explicit type definitions for tuples in the model. In the Java language too, there is no need for explicit tuple types. You may easily use any implementation of the *Map* interface from java.util.

As an alternative, you can choose to define a separate Java class for each tuple type. This is safer, because it ensures proper typechecking in the Java code.

4.3.3 OCL Collection Types

OCL collection types must be mapped to the collections in one of the libraries of the target language, or when the target language does not provide collections, you have to built your own. Java provides a large number of different collection types, for instance, *Set*, *Tree, List*. Chose one for each OCL collection type. It is usually best to stick to this choice for every mapping that needs to be made. That is, always use the same Java class for implementing an OCL *Set*, another for implementing an OCL *Bag*, etc. Because there are no direct counterparts for Bag and OrderedSet you need to choose a closely matching type. Table REF shows a possible mapping.

OCL collections have a large number of predefined operations. These operations come in two flavours, the ones that loop over the collection (*select*, *exists*, *collect*) and the ones that don't (*union*, *size*). Any operation in the first category is called an *iterator*.

Simple operations

Operations in the latter category must be mapped to operations on the target language collection types. When they are not present, you have to make a choice. You can define your own classes to represent the OCL collection types by inheriting from a standard collection type. The remaining OCL operations that could not

OCL Collection type	Java type	Concrete type
Set	Set	
Sequence	List	
Bag	List	
OrderedSet	List	

Table 4-2 *Mapping of collection types from OCL to Java.*

be mapped directly can be implemented on these classes. The disadvantage is that these collection classes need to be used everywhere in the Java code, because the OCL expressions will use collection typed fields in the Java code.

Another option is to use the standard Java collection classes and add the required operations as static Java methods in a separate library class. You can now use ordinary Java collections anywhere, and you will only refer to the special static methods when you evaluate an OCL expression.

```
public class OclCollections {
  static notEmpty(Collection c) {
    (c == null) || (c.length == 0);
  }
}

OCL Expression
     someCollection->notEmpty()

Java code Expression
     OclCollections.notEmpty(someCollection)
```

4.3.4 Collection Iterators

The iterators are usually more difficult to implement. The collections in the Java library do not provide fitting counterparts for these operations. Fitting counterparts for the operations that loop over collections cannot be found in many languages. In that case every occurence of such an operation must be implemented by a special code fragment.

All collection iterators loop over a collection. To implement this you need to use the looping mechanisms of the target language. In Java looping is usually implemented using the Iterator class. For instance, the OCL expression `source->select(a=true)`, where *source* is a Set, can be implemented by the following piece of Java code:

```
OCL expression:
     source->select(a=true)

Java code:
     Iterator it = source.iterator();
     Set result = new HashSet();
     while( it.hasNext() ){
         ElementType elem = (ElementType) it.next();
         if ( elem.a == true ){
             result.add(elem);
         }
     }
     return result;
```

There are only two places in this code fragment where the OCL expression can be recognized. These are shown in boldface. The first is a reference to the collection *source*, the second is the test *a* == *true* from the body of the select operation.

When the test is more complex then in this example, it will be more difficult to recognize. For instance, to get all program partners for which all services have no points to be earned from the context of a LoyaltyProgram we can use the following OCL query:

```
self.partners->select(deliveredServices->forAll(pointEarned = 0))
```

This results in the following piece of Java code :

```
Iterator it = self.getPartners().iterator();
Set selectResult = new HashSet();
while( it.hasNext() ){
    ProgramPartner p = (ProgramPartner) it.next();
    Iterator services = p.getDeliveredServices().iterator();
    boolean forAllresult = true;
    while( services.hasNext() ){
    Service s = (Service) services.next();
        forAllResult = forAllResult && (s.getPointsEarned() == 0);
    }
    if ( forAllResult ){
        selectResult.add(p);
    }
}
return result;
```

Example implementation of collect.

```
implementation of
    // source->collect(attr)
    //
    Iterator it = source.iterator();
    Set result = new HashSet();
    while( it.hasNext() ){
        ElementType elem = (ElementType) it.next();
        AttrType attr = (AttrType) elem.getAttr();
        if ( !(attr instanceof ClassImplementingOclCollection) ) {
            result.add( elem.getAttr() );
        } else {
            // ........
        }
    }
    return result;
```

Each loop operations (*select*, *exists*, *forAll*, etc.) will have its own template, which needs to be filled with the details of the OCL parts. These templates can be optimized. For example the *forAll* template, as used in the nested example above can be written to stop the loop as soon as the result is found to be false. This, of course means that the Java code will become even lengthier and more difficult to read.

The collection examples in this section shows that such expressions in OCL are more comprehensive than the corresponding Java code. As a result it it easier to write, but , even more important, it is much easier to read and understand the OCL.

4.4 IMPLEMENTING OCL EXPRESSIONS

Once you have decided on the way you want to implement the model elements and the OCL standard library, implementation of OCL expressions is rather straightforward. There are some issues that need attention.

4.4.1 Evaluation Order

Evaluation order is not (and does not need to be) defined in OCL, it is perfectly legal to write a constraint and have part of that constraint be *Undefined* (see section 10.5). For example, consider the following invariant in the R&L model.

```
context Membership
inv: loyaltyAccount.points >= 0 or loyaltyAccount->isEmpty()
```

If there is no *loyaltyAccount*, the *loyaltyAccount.points > 0* evaluates to *Undefined* and the *loyaltyAccount->isEmpty()* evaluates to true. In OCL, the result of the full invariant is true and is well defined. However, if we generate code directly from the invariant, and the execution order is left to right, the *loyaltyAccount.points* code will try to reference a nonexisting object (*loyaltyAccount*). Usually this leads to a runtime error (e.g. java.lang.nullPointerException), so care must be taken to avoid such situations.

4.4.2 No Side Effects

Another issue is that an OCL expression should always be treated as an atomic expression. No changes of value of any object in the system can take place during evaluation of the expression. In purely sequential applications this might not be a problem, but in a parallel multi-user environment it must be addressed. The values referenced in a single OCL expression should be visible and reachable from the thread that executes the implementation of the expression.

4.4.3 Getting Attribute and AssociationEnd Values

The code for getting an attribute value depends on the way that attributes from the UML model are mapped to Java. In the mapping described in section 4.2 the value of an attribute name *attribute* can be obtained thtough the operation *getAttribute()*. Therefore any attribute reference in OCL needs to be mapped to the corresponding *get*-operation.

```
OCL expression:
     self.attribute

Java code:
     this.getAttribute();
```

The Java code to get the value of an association-end depends on the mapping of the UML association in the same way. The solution in section 4.2 maps an association-end to a Java private attribute with corresponding *get*-operation. This means that the OCL navigation of an association will become an operationcall in Java:

```
OCL expression:
     self.associationEnd

Java code:
     this.getAssociationEnd();
```

4.4.4 Let Expressions

A let expression can be implemented by defining a local variable in the scope of the OCL expression that uses the let variable.

4.4.5 Treating Instances as Collections

In OCL, an object can be used as a collection as in instance->size(). This is a special case. The standard way to do this in Java

```
OCL expression
     instance->size()

Java expression
     Set coll = new HashSet();
     coll.add( instance );
     return coll.size();
```

4.5 PLACING CODE FRAGMENTS IN CONTEXT

The code fragments that implement OCL expressions must be placed in the code that implements the system according to their role in the model. For most expressions this is simple. An attribute definition belongs in the class that is mentioned in the context definition. An initial value of an attribute must be placed where the attribute is created. For other expressions, like guards, invariants and pre- and postconditions this is slightly more complex.

4.5.1 Derivation rules and Initial Values

Derivation rules can be implemented in two diffrent ways. The choice depends on the complexity and usage of the derivation rule.

The most straightforward approach is to implement an attribute with a derivation rule as a query operation, where the derivation rule serves as the body of the operation. Each time the attribute value is requested the derivation will be calculated.

If the evaluation of the rule is expensive and the attribute is frequently used, the above strategy might not be optimal. As an alternative, the value of the attribute can be stored as an attribute. However, the object that ciontains the attribute needs to be notified of changes in the objects that the derivation depends upon. The Java listener pattern can be used to setup such a notification mechanism. Each time that the value of the attribute is requested, you can check whether it needs to be recalculated.

In any case, a derived attribute should not have a public *set*-operation, because no other object shouold be able to change its value.

Initial values and parameter values are implemented in a straight forward manner in the constructor.

4.5.2 Body of Query Operations

If the body of a query operation is given as an OC expression, the Java for that expression will become the body of the operation.

4.5.3 Invariants, Pre- and Postconditions

Implement as separate operation that can be called when needed.

The best way to implement an invariant is to write a separate operation with a boolean result that implements the check. This operation can be called whenever it is appropriate to check the invariant. Example: to be done.

4.5.4 Attribute and Operation Definitions

An attribute definition can be implemented as a normal attribute with a derivation rule. An operation definition can be implemented as a normal query operation with a body expression.

4.5.5 Guards and Change Events in Statecharts

Depends on how statecharts and transitions are implemented. When transition is implemented by operation (transition is actually an operation call where the object responds to), the guard must be part of the precondition, as is the start state. The end state is part of the postcondition.

To be done: An example of a state diagram with template of code + code for OCL expressions.

4.5.6 Code in Interaction Diagrams

Interaction diagrams are not complete specifications, but examples of interactions between instances. Interaction diagrams can be used to provide a template for the operations taht are visible in the diagrams, but they cannot be used to generate the complete Java code.

The OCL expressions used for conditions, target objects of messages and parameetrs of massages all become part of the operation templates.

To be done: An example of an interaction diagram with template of code + code for OCl expressions.

Actual Parameter Values

Target to Actions

Conditions

4.6 CONSIDERATIONS FOR CONSTRAINTS

4.6.1 Asserting Invariants

When coding invariants, you must decide when to check them. Invariants are defined as being true at any moment in time, but they cannot be checked continuously. One obvious solution is to check the invariants on an object immediately after any value in the object has changed. For example, when the value of an attribute changes, you would check all invariants that refer to the attribute. Of course, this might include invariants on objects other than the object that is being changed. If this approach is too inefficient, you need to find a solution that balances complete checking with runtime efficiency.

Een invariant altijd checken na uitvoeren van een publieke operatie, of op aanroep. De invariant en pre/post operaties hebben als result type Boolean.

4.6.2 Asserting Pre- and Postconditions

Pre- and postconditions can best be implemented in the operation for which they are defined. Some languages provide an assert-mechanism that can be used. Assert aan- en uitzetten. including Guards and Events and Messaging

Example:

```
context LoyaltyProgram::enroll(c : Customer)
pre : not customer->includes(c)
post: customer = customer@pre->including(c)

Void enroll(Customer c) {
assert( not customer.includes(c) );
old_customer = customer;
...
< body of operation >
...
assert( customer = old_customer.including(c)
}
```

Preconditions could be checked each time the operation is called. Depending on the complexity of the precondition and the performance requirements, the cost of this technique might be prohibitive. In the Eiffel community, preconditions are often checked during development, testing, and debugging but not checked (or only partially checked) when the application is operational.

Postconditions are naturally checked at the end of the execution of an operation. The same arguments hold here as with preconditions. Practice has shown, however, that precondition checking is much more important than postcondition checking. Therefore, if the possibility for checking in an operational system is limited, preconditions are the best candidates to check.

4.6.3 Using Guards and Events in Pre- and Postconditions

The information in a state transition diagram of a class can be rewritten in the form of pre- and postconditions on operations of that class. This technique can be useful when you are implementing the information in the model to an implementation. The basic assumption of this technique is that events often correspond to operation calls. The state transition diagram tells us when these calls are permitted, which guards must be true, and in what state the object must be to perform this operation. This information on state and guards can be used as preconditions for the operation.

From the state transition diagram you can conclude the effects of an event: the object may enter a different state. This state information can also be included in postconditions.

As an example, consider the state transition diagram for the *Bottle* class in figure 4-1. Only two events are defined: *fill(amount: Integer)* and *cap*. The *fill* event will undoubtedly be implemented by an operation in the implementation code. The precondition for this operation can be read from the state transition diagram: the only state in which the *fill* operation will not perform is *filled*. The postcondition is somewhat more complex. Either the object is in the state *partiallyFilled* and the new *contents* is less than the *capacity*, or the object is in the state *filled* and the new *contents* is equal to or more than the *capacity*. In OCL, we can express the information from the state transition diagram as follows:

```
context Bottle::fill(amount : Integer)
pre: not filled and not capped
post: (partiallyFilled and
                contents@pre + amount < capacity )
      or
      (filled and contents@pre + amount >= capacity )
```

Note that all state names are transformed into attributes of the *Boolean* type. We could have transformed the state names into one attribute of an *Enumeration* type, *state: enum{empty, partiallyFilled, filled, capped}*, as in figure 4-1. In that case, the OCL expressions would be

```
context Bottle::fill(amount : Integer)
pre : not state = #filled and not state = #capped
post: (state = #partiallyFilled and
                contents@pre + amount < capacity )
      or
      (state = #filled and
                contents@pre + amount >= capacity )
```

The pre- and postconditions for the *cap* operation are simple:

```
context Bottle::cap()
pre : state = #filled
post: state = #capped
```

4.6.4 Using Messaging in Postconditions

To be done: Probably leave out

4.6.5 Checking Constraints in the Deployed System

Another question to answer is whether you want constraints to be checked during deployment. during system development checking is very useful. When the system is deployed checking might take to much processor time, making the system slower than neccessary.

4.6.6 What to Do When a Constraint Fails

What is often a matter of discussion in the interpretation of constraints is how one should react when a constraint is broken, in other words what action has to be taken when the restrictions laid on the objects are no longer met. There are three different approaches to broken constraints.

Some people argue that breaking a constraint should throw some kind of exception, or at least that such behavior should be an option. In Eiffel, assertions are used as a debugging tool and as an exception facility; this means that when the assertion (or in our terms the constraint) is broken, an exception is thrown. In Java, which has recently incorporated the concept of assertions (version 1.4), this approach is used as well.

Others argue that the breaking of a constraint is a trigger for an operation to be executed. For instance, in Soma, the analysis and design method defined by Ian Graham, rules can be triggers to actions that the system must undertake. We call both of these kinds of constraints *operational constraints*.

Other possibilities are:

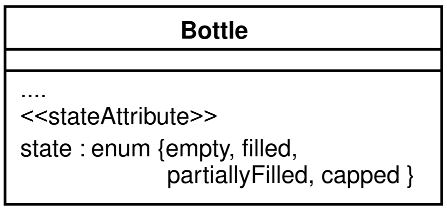

Figure 4-1 *The state attribute.*

- Dump a program trace.
- When in a transaction environment, roll back the transaction.
- Print a message to the operator or user.

From the point of view of extending OCL most of these approaches are simple. We will discuss the first possibility in some detail.

If we choose the first option we must add to OCL a syntax for associating a constraint with an action along with a syntax for defining the action. Such a syntax could, for example, look like this:

```
when <OCL-constraint> broken do <action-part> endwhen
```

Because the purpose of the action part is probably to try to repair the broken constraint, the action part will have side effects. Being free of side effects is at the heart of the OCL definition, so it is impossible to extend OCL with such an action part. Therefore, the action part will have to be written in another language. Of course, a language that is in line with the OCL principles could use a similar syntax and become an OCL look-alike. The semantics of such a language with side effects will need much attention.

4.7 SUMMARY

This chapter has discussed ... TBD.

Chapter 5

Using OCL for MDA

OCL is a small, yet extremely key ingredient for the MDA. Without a precise modeling language like OCL, consistent and coherent platform independent models cannot be made. Besides that there are two other aspects where OCL is important for the MDA. This chapter explains how and where OCL fits in with the MDA.

5.1 RELATION OF OCL TO MDA

OCL relates strongly to the MDA framework. In fact, it is a key element in the MDA. In section 1.1.3 we explained the building blocks of the MDA framework: models, languages, transformation definitions, and transformation tools. As shown in figure 5-1, OCL is very helpful in creating at least three of the building blocks:

1. models, because only with a precise specification language models can be built on maturity level 4.
2. transformations definitions, because a formal and precise language is needed to write transformation definitions that can be used by automated tools.
3. languages, because languages will need to be understood within the MDA framework. This is only possible if the language definition is formal and precise.

How to build better models using the OCL has been the subject of this book so far. In this chapter we will focus on the second and third aspect in which the OCL is useful: the definition of transformations and modeling languages. For this we must get to know the metalevel of modeling. In the remainder of this chapter will we introduce metamodels and metamodeling, explain the UML and OCL metamodels, see how the OCL has aided in the development of these metamodels, and give an example of a transformation definition written in OCL.

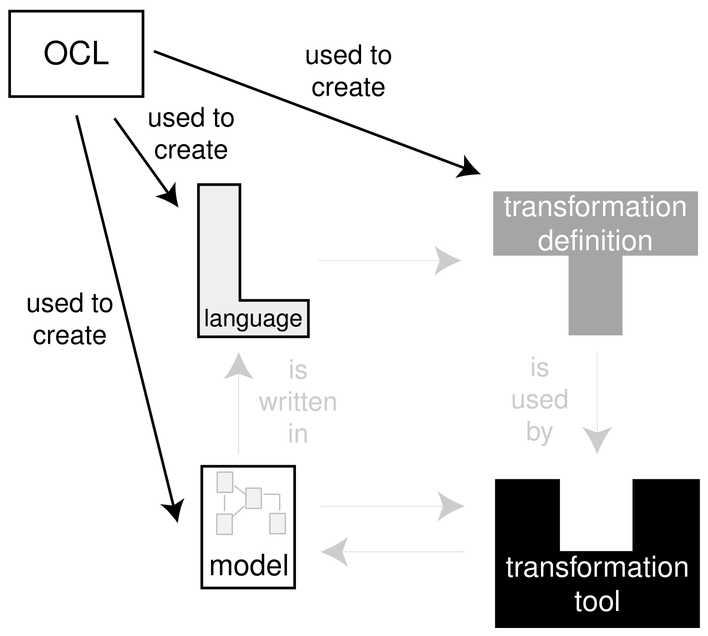

Figure 5-1 *Use of OCL in the MDA framework.*

5.2 METAMODELS

The *metamodel* of a language, also known as the abstract syntax, is a description of all the concepts that can be used in that language. For instance, the concept *attribute* is part of the UML language, the concepts *constructor*, *method*, and *field* are part of the Java language, the concepts *table*, *column*, and *foreign key* are part of the SQL language. These concepts are sometimes called *metaclasses* or *metatypes*. The set of all metaclasses of a language and the relationships between them constitute the metamodel of that language.

Every element of an ordinary model is an instance of a concept in the modeling language used, in other words, every model element is an instance of a metaclass. For instance, a class in an UML model, called *Car*, is an instance of the metaclass *Class* from the UML metamodel. An attribute of class *Car*, called *isValuable* of type *Boolean*, is an instance of the metaclass *Attribute* from the UML metamodel. In the model there is a relation between *Car* and *isValuable*. In the metamodel this is reflected by the relationship between metaclass *Class* and metaclass *Attribute*. Actually, all modelers are familiar with this instance-of relationship: an object named *The Object Constraint Language* is an instance of the class called *Book*. In a

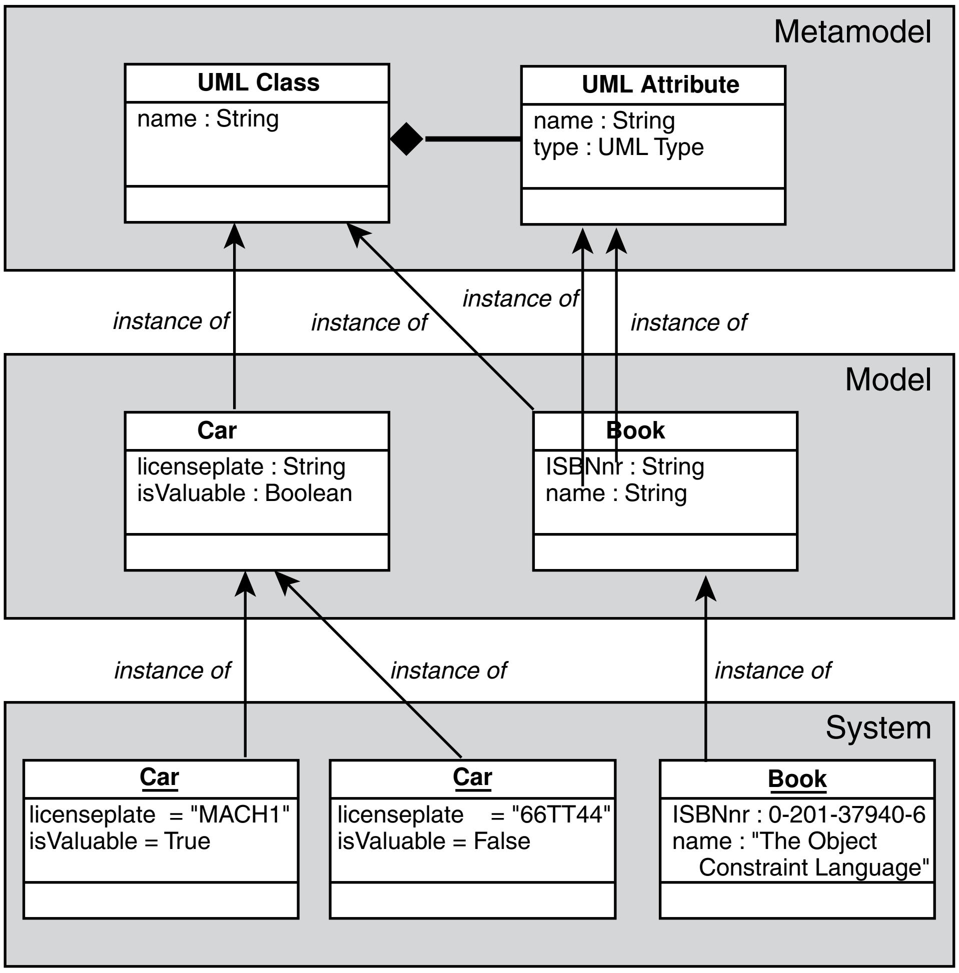

Figure 5-2 *Relation between system, model, and metamodel.*

metamodel this relationship is brought one level higher: the class called *Book* is an instance of the metaclass called *Class*. Figure 5-2 shows both instance-of relationships.

As a class defines its objects, a metaclass defines its instances, the model elements. The metaclass *Attribute* from the UML metamodel defines that an attribute should have a name and a type, e.g. Boolean. The metaclass *Class* defines that each class should have a name, and that it could have attributes, operations, etc. Each of these related elements must itself be defined within the metamodel, that is, the metaclasses *Attribute*, and *Operation* should exist.

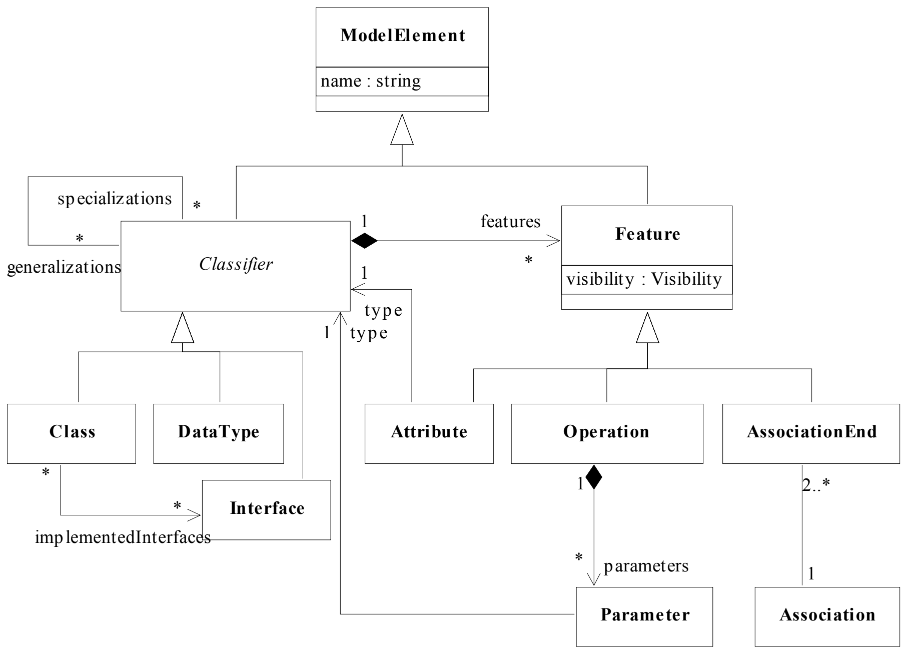

Figure 5-3 *UML metamodel (simplified)*

A modeler can use in his/her model only elements that are defined by the metamodel of the language the modeler uses. In UML you can use classes, attributes, associations, states, actions, etc., because in the metamodel of UML there are elements that define what a class, attribute, association, etc. is. If the metaclass *Interface* was not included in the UML metamodel, a modeler could not define an interface in an UML model.

5.3 THE OCL AND UML METAMODELS

The OCL and UML metamodels are defined in the OMG standards, which are complete books on their own. In this section we can only describe simplified versions of both. We will explain the relationship between them, which will clarify the context of an OCL expression in terms of metaclasses.

5.3.1 The UML Metamodel

A simplyfied version of part of the UML metamodel is shown in figure 5-3. It contains the concepts that can be used in a class diagram. Everything in a model is a

ModelElement, so this is the superclass of all other metaclasses. Then there are types, which in the UML metamodel are called *Classifiers*. *Classifier* is an abstract superclass of the types one may encounter in a model: *Class*, *DataType*, and *Interface* (actually *Component* should be in this list too). Every classifier has *Features*, either *Attributes*, *Operations*, or *AssociationEnds*. Every feature has a type. For attributes this is simply the type of the attribute. For operation this represent the return type. For association ends the type refers to the class the holder of the association end is associated with. To define associations two or more *AssociationEnds* are coupled into one *Association*.

For instance, in the R&L model the class *LoyaltyProgram* has an association with the class *ProgramPartner*. Both ends of the association have multiplicity 1..*. The end at *LoyaltyProgram* is called *programs*, the end at *ProgramPartner* is called *partners*. In terms of the metamodel this mean that there are two instances of metaclass *Class*, called *LoyaltyProgram* and *ProgramPartner*. The *LoyaltyProgram* instance holds an *AssociationEnd* instance called *partners*. The type of this *AssociationEnd* is the instance of metaclass *Class* called *ProgramPartner*.

5.3.2 The OCL Metamodel

A simplyfied version of part of the OCL metamodel is shown in figure 5-4. It defines the various types of expressions that can be used. Superclass to all expressions is the metaclass *OclExpression*. The subclass *ModelPropertyCallExp* represent an expression that references a value within the model, an attribute, operation, or association end. The metaclass *ModelPropertyCallExp* has three subtypes to indicate the kind of feature of the source that has been called: *AttributeCallExp*, *OperationCallExp*, and *AssociationEndCallExp*. Because properties are always called from some object, a *ModelPropertyCallExp* has a source which is another *OclExpression*.

For instance, in the following expression *monkey* is the *OclExpression* that is the source of the *pealsBanana()* part. The *pealsBanana()* part is an instance of *ModelPropertyCallExp*. In fact, it is an instance of its subclass *OperationCallExp*.

```
monkey.pealsBanana()
```

Expressions that loop over collections are instances of the metaclass LoopExpression. Each loop expression has a source that is a collection or an instance treated as a collection. This source can be written as an OCL expression. The body of a loop expression is the part that indicates which elements of the collection should be considered. For instance, in the following expression *monkey.siblings* is an OCL expression that represents the source of the *select* loop expression. The part between brackets, *eyes.colour = Colour::blue*, is the body of the loop expression.

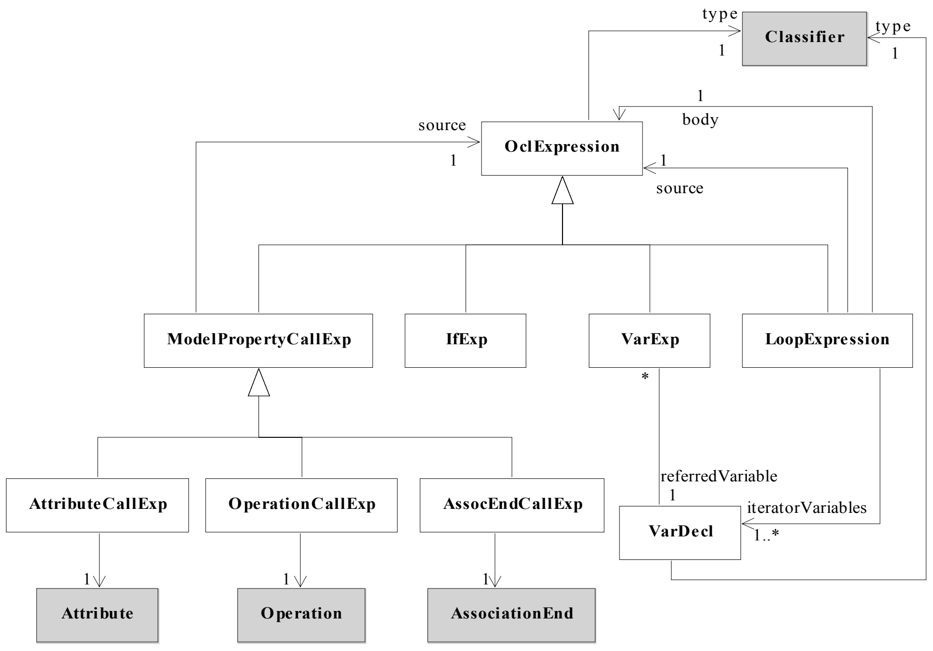

Figure 5-4 *OCL metamodel (simplified) showing relation to UML metamodel*

```
monkey.siblings->select( eyes.colour = Colour::blue )
```

Although not shown in figure 5-4, the OCL metamodel adds a number of datatypes to the UML metamodel, like the set, ordered set, bag and sequence types. Each is a subclass of *Classifier,* in fact they are subclasses of the metaclass *DataType*.

5.3.3 The Relation Between the UML and OCL Metamodels

The relation between the UML and OCL metamodels is twofold. First, an OCL expression may reference an element from the model. This element is an instance of a UML metaclass. This relation is shown in figure 5-4. The metaclasses from the UML metamodel are shown gray. The associations between the OCL metaclasses and the UML metaclasses define the relationship between the UML and OCL metamodel.

For instance, every OCL expression results in a value. The type of this value is an instance of the UML metaclass *Classifier*. This relation is represented in the metamodels by the fact that there is an association between the metaclasses

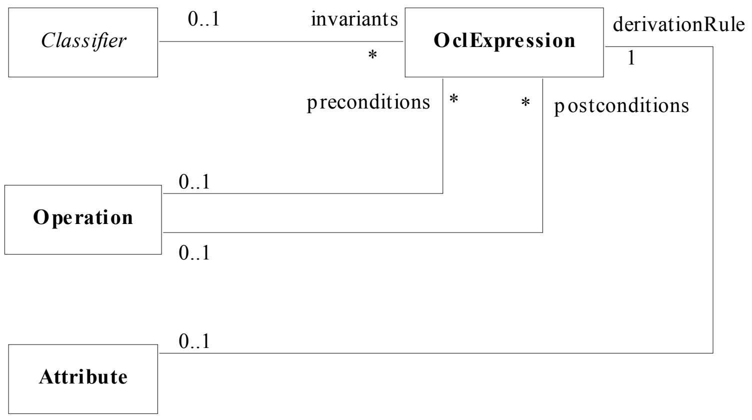

Figure 5-5 *OCL context in terms of the metamodels*

OclExpression and *Classifier*. Another example is an OCL expression that references an attribute. The expression is an instance of *AttributeCallExp*, and the attribute is an instance of *Attribute*. Their relation is represented by the association between both metaclasses.

The second relationship between the UML and OCL metamodels is shown in figure 5-5. Various elements from the UML metamodel may be adorned with information in the form of an OCL expression. For instance, invariants are *OclExpression* instances that are linked to a *Classifier* instance. This relationship represents the relation between an OCL expression and its context.

For instance, an attribute A may have a derivation rule. In terms of the metamodels the attribute is an instance of the metaclass *Attribute*, the derivation rule is an instance of the metaclass *OclExpression*. The fact that the rule describes the derivation for attribute A, is represented in the metamodel by the association between the metaclasses *Attribute* and *OclExpression*.

5.4 USING OCL TO DEFINE LANGUAGES

One of the conclusions that may be drawn from the previous sections is that metamodels are simply models. The only difference is that they reside on another level of abstraction. As OCL is useful in building good models, it is useful in building good metamodels. When you build a metamodel, you are actually defining a language. Thus OCL is helpful in defining languages.

In fact the UML was defined using OCL. Approximately twohunderd invariants and onehunderd operation definitions were written to define the metamodel of UML. In the standard they are called well-formedness rules. The following expression is an example that states that an interface may not have any attributes:

```
context Interface
inv: features->select(f |f.oclIsKindOf( Attribute ) )->isEmpty()
```

Even OCL itself was defined using OCL expressions. In the OCL metamodel we can find another example. This one tells us that the source of a loop expression must be a collection, and that the type of the iterator varable must be equal to the type of the elements of the source collection.

```
context LoopExpression
inv: source.oclIsKindOf( Collection )
inv: iteratorVariable.type = source.elementType
```

Within the MDA framework new languages will need to be defined, but also there is a need for throrough specifications of existing languages. Most programming languages, for instance, are not formally defined. The only formal part in their specification is usually the grammar that is written in a BNF format. For the rest its users rely on the compilers, and the textbooks.

New languages can also be defined as so-called *profiles* to UML. This means that there is not a completely new metamodel for the language. Instead the metamodel of UML is used. Extra rules and a mapping of the language concepts to the syntax used are given.

Existing languages can also be fitted in the MDA framework using UML profiles. For instance, there is a Java profile [EJB01]. When you use this profile and draw a diagram that looks like a UML class diagram, you have actually created a number of Java classes.

5.5 USING OCL TO DEFINE TRANSFORMATIONS

A transformation definition describes how a model written in one language can be transformed into a model written in another language. Such a description is generic when it is independent of the actual models. Instead this description must make use of the concepts defined in both languages, in other words it is built using the metaclasses in the metamodels of both languages. A transformation definition relates metaclasses in the source language to metaclasses in the target language.

OCL is useful for defining transformations. An OCL expression is a representation of an element in the model, in this case in the metamodel. An OCL expression can therefore precisely indicate which element or elements in the source metamodel are used in a certain transformation. The same holds for the elements in the target metamodel. For instance, when a UML class is being transformed into a

Java class, not all attributes in the model need to be transformed into class members, just the ones that are owned by the UML class that is being transformed. In OCL this can be expressed precisely. From the context of the UML class the attributes to be transformed are exactly identified by the following expression.

```
self.features->select( f | f.isOclType( Attribute ) )
```

Because transformations are to be executed by automated tools, transformation definitions will need to be written in a precise and unambiguous manner. At the time of writing this book no standard language for writing transformation definitions existed. In our opinion such a language should be built on the assets of OCL. In the next section we use a language that is an extension of OCL to write an example transformation definition. More information on transformation definitions can be found in [Kleppe03].

5.5.1 Example Transformation Definition

This section shows the definition of the transformation of a public attribute to a private attribute and a get and set operation. Of course this is only a very simple example, yet it shows how transformations can be defined using OCL expressions. The source and target languages are both UML. The transformation will be executed according to the following rules:

- for each class named *className* in the PIM there is a class named *className* in the PSM.
- for each public attribute named *attributeName : Type* of class *className* in the PIM the following attributes and operations are part of class of class *className* in the target model.
  - a private attribute with the same name: *attributeName : Type*
  - a public operation named with the attribute name preceded with 'get' and the attribute type as return type: *getAttributeName() : Type*
  - a public operation named with the attribute name preceded with 'set' and with the attribute as parameter and no return value: *setAttributeName(att : Type)*

The rules above must be written in a manner that can be understood by an automated tool, therefore we need to formalize them. Thereto we use a language that is an extension of OCL. In it each transformation rule is named, and its source and target language are specified. In a rule a condition may be specified under which the elements of the source language metamodel can or cannot be transformed. Likewise, a condition may be specified that must hold for the generated elements. Finally the actual transformation is defined by stating the rule to be used to transform a metamodel element of the source language in a metamodel element of the target language. Because there may be conditions in the applied rule we use the keyword **try** to indicate that the rule will be applied only in the case the source and target conditions hold. The **<~>** symbol represents the transformation relation.

```
Transformation ClassToClass (UML, UML) {
  source c1: UML::Class;
  target c2: UML::Class;
  source condition -- none
  target condition -- none
  mapping
         try PublicToPrivateAttribute on
                        c1.features <~> c2.features;
         -- everything else remains the same
}
Transformation PublicToPrivateAttribute (UML, UML) {
  source sourceAttribute : UML::Attribute;
  target targetAttribute : UML::Attribute;
         getter          : UML::Operation;
         setter          : UML::Operation;
  source condition
         sourceAttribute.visibility = VisibilityKind::public;
  target condition
        targetAttribute.visibility = VisibilityKind::private
         and -- define the set operation
         setter.name = 'set'.concat(targetAttribute.name)
         and
         setter.parameters->exists( p |
                     p.name = 'new'.concat(targetAttribute.name)
                       and
                       p.type = targetAttribute.type )
         and
         setter.type = OclVoid
         and -- define the get operation
         getter.name = 'get'.concat(targetAttribute.name)
         and
        getter.parameters->isEmpty()
         and
         getter.returntype = targetAttribute.type;
  mapping
         try StringToString on
                    sourceAttribute.name <~> targetAttribute.name;
         try ClassifierToClassifier on
                    sourceAttribute.type <~> targetAttribute.type;
}
-- somewhere the rules StringToString and ClassifierToClassifier
-- need to be defined
```

5.6 SUMMARY

To be done: write summary

Part 2

Reference Manual

Chapter 6

The Context of OCL Expressions

This chapter describes the relation between the UML and OCL parts in a combined model.

6.1 A COMBINED MODEL

OCL relies on the types (classes, datatypes, etc.) defined in a UML model, thus the use of OCL includes use of (at least some aspects of) UML. Any model in which OCL plays a part, consists of some UML diagrams, and a series of OCL expressions. Often only the class diagram is used, but other diagrams may be included in the specification.

A model must be an integrated, consistent entirety. In a model it must be crystal clear how entities used in one diagram relate to entities in other diagrams. The same holds for the relation between expressions and entities in the diagrams. There are two ways in which this relation can be viewed. First, expressions linked to specific entities may have only specific functions. For instance, an expression defining a new attribute, may only be attached to a class, interface, or datatype. Second, the UML model entity to which an expression is linked, defines which other model entities are visible, and can be referenced. For instance, in an expression attached to a class, all attributes, associations, and query operations of that class may be used.

The link between an entity in a UML diagram and an OCL expression is called the *context definition* of an OCL expression.

6.1.1 The Context of an OCL Expression

The context definition states for which model entity the OCL expression is defined. Usually this is a class, interface, datatype, or component. Sometimes it is an operation, and rarely it is an instance. It is always a specific element defined in a UML diagram. This element is called the *context* of the expression.

OCL expressions can be incorporated in the model directly in the diagrams, but they may also be given in a separate text file. In both cases there is a context definition. In the diagram the context definition is shown by a dotted line that links the model element and the OCL expression. In the example in figure 6-1 five expressions and their contexts are shown.

When the OCL expression is given in a separate text file, the context definition is given in a textual format. It is denoted by the keyword *context* followed by the name of the type, as in the next example invariant.

```
context Customer
inv: name = 'Edward'
```

Next to the context, it is important to know the contextual type of an expression. The *contextual type* is the type of the object for which the expression will be evaluated. With type we mean either a class, an interface, a datatype or a component (in terms of the UML standard, a *Classifier*). When the context itself is a type, the context is equal to the contextual type. When the context is an operation, attribute, or association end, the contextual type is the type for which that feature has been defined. When the OCL expression is connected to an instance in a diagram, the contextual type is the type of that instance.

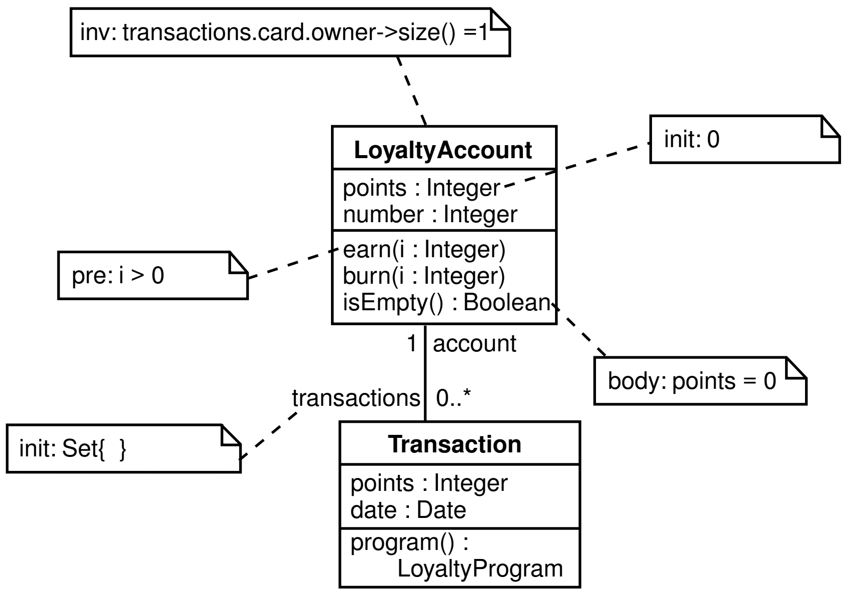

Figure 6-1 *OCL expressions and their context.*

OCL expressions are evaluated for a single object. This is always an instance of the contextual type. To distinguish between the context and the instance for which the expression is evaluated, the latter is called the *contextual instance*.

OCL expressions can have many different functions, depending on the context of the expression. For instance, when the context is an attribute, the expression may represent an initial value or a derivation rule, but when the context is a class, the expression will never represent an initial value (an initial instance). The remaining sections in this chapter describe in which ways an OCL expression may be used when connected to various contexts.

6.1.2 The *self* Keyword

Sometimes it is necessary to refer explicitly to the contextual instance. The keyword *self* is used for this purpose. Whenever the reference to the contextual instance is obvious, the use of the keyword *self* is optional. The previous invariant thus can be written as follows:

```
context Customer
inv: self.name = 'Edward'
```

In the R&L model from chapter 2 there is an example of an invariant in which the reference to *self* is not optional:

```
context Membership
inv: participants.cards.membership.includes( self )
```

6.1.3 More than one Expression to a Context

Often more than one invariant, or pre- or postcondition, or other type of expression applies to the same context. These can be combined to follow one context definition statement. Because all invariants need to be true for an instance of the class, the invariants are conceptually connected by the boolean *and* operation. The same holds for sets of pre- and postconditions. So the following two invariants have exactly the same meaning.

```
context Customer
inv: self.name = 'Edward'
inv: self.title = 'Mr.'

context Customer
inv: self.name = 'Edward' and self.title = 'Mr.'
```

The following two sets of pre- and postconditions have the same meaning as well.

```
context LoyaltyProgram::addService(p: Partner,
                                    l: Level,
                                    s: Service)
pre: partners->includes( p )
pre: ServiceLevel->includes( l )
post: partners.deliveredServices->includes( s )
post: ServiceLevel.availableServices->includes( s )

context LoyaltyProgram::addService(p: Partner,
                                    l: Level,
                                    s: Service)
pre: partners->includes( p ) and ServiceLevel->includes( l )
post: partners.deliveredServices->includes( s ) and
               ServiceLevel.availableServices->includes( s )
```

6.2 CLASSES AND OTHER TYPES

This section explains which function expressions may be have when the context is a type, i.e. a class, interface, datatype, or component,. The model elements that may be used in this case are all attributes, all query operations, all states, and all associations of the type.

6.2.1 Invariants

The first way in which an expression with a type as context can be used is the invariant. An invariant is described using an boolean expression that evaluates to true if the invariant is met. The invariant must be true upon completion of the constructor, and completion of every public operation but not necessarily during the execution of operations. Incorporating an invariant in a model means that any system made according to the model is faulty when the invariant is broken. How to react when an invariant is broken is explained in section 4.6.6.

To indicate that the expression is intended to be an invariant, the context declaration is followed by the keyword *inv,* an optional name, and a colon, as in the following example.

```
context Customer
inv namingInvariant: self.name = 'Edward'
```

The meaning of an OCL expression used as invariant is that for all instances of the contextual type, the expression must evaluate to true. Thus, in our (stupid) example, all instances of class *Customer* would have to be named Edward.

An invariant may be named, which can be useful for reference in an accompaning text. The above invariant is named *namingInvariant*.

6.2.2 Definitions of Attributes or Operations

Attributes or operations may be defined by an OCL expression. The meaning of an attribute or operation definition is that every instance of the contextual type holds an attribute or operation that conforms to the given definition.

The context of an attribute or operation definition is always the type that must hold the new element. To indicate that the expression is intended to be a definition, the context declaration is followed by the keyword *def* and a colon, as in the following examples.

```
context Customer
def: shortName : String = 'Eddie'
def: newOper(par: T) : String = par.assoc To be done!
```

In the case of an attribute definition, the name and type of the attribute must be given. The expression following the equal sign is also mandatory. This expression indicates how the value of the attrbitue must be calculated. It is a derivation rule (see section 6.3.1).

All operations defined by an OCL expression are considered to be query operations. The name, parameters including their types, and the returntype, if any, of the operation must be given. The expression following the equal sign is also mandatory, and states the result of the operation (see section 6.4.2).

6.3 ATTRIBUTES AND ASSOCIATION ENDS

This section explains which functions expressions may be have when the context is an attribute or the role of one end of an association. The model elements that may be used in this case are all attributes, all query operations, all states, and all associations of the contextual type.

6.3.1 Derivation Rules

An expression whose context is an attribute or association role may be used as a *derivation rule*. The meaning of a derivation rule is that the value of the context element should always be equal to the value given by the evaluation of the derivation rule. In the case the context is an attribute, the contextual type is the type that holds the attribute. In the case the context is an association end, the contextual type is the type at the opposite end of the association. For example, in the R&L model the context of the association end named *participants* is the class *LoyaltyProgram*.

To indicate that the expression is intended to be a derivation rule, the context declaration includes the name of the attribute or the association end, and is followed by the keyword *derive*, and a colon, as in the following examples.

```
context LoyaltyAccount::totalPointsEarned : Integer
derive: transactions->select( oclIsTypeOf( Earning ) )
           .points->sum()

context CustomerCard::myLevel : ServiceLevel
derive: membership.currentLevel
```

6.3.2 Initial Values

The initial value of an attribute or association role can also be given by an OCL expression. The meaning of an initial value is that the attribute or association end will have this value at the moment that the contextual instance is created. The context declaration is followed by the keyword *init*, the name of the attribute, and the expression that gives the initial value, as in the next example.

To be done: example

Note the difference between an initial value and a derivation rule. A derivation rule in fact states an invariant: the derived element should always have the same value as the rule expresses. Whereas, an initial value must hold only at the moment when the contextual instance is created. After that moment the attribute may have a different value at any point in time.

6.4 OPERATIONS

This section explains the different functions which expressions may have for operations in the model. The model elements that may be used in an expression whose context is an operation, are all attributes, all query operations, all states, and all associations of the contextual type, plus the parameters of the operation.

6.4.1 Preconditions and Postconditions

The first two ways in which expressions may be used for operations, are pre- and postconditions, two forms of constraints. A precondition is a boolean expression that must be true at the moment that the operation starts its execution. A postcondition is a boolean expression that must be true at the moment that the operation ends its execution. The meaning of a precondition is that the expression must evaluated to true, otherwise the operation will not be executed. The meaning of a postcondition is that the expression must evaluated to true, otherwise the operation has not executed correctly.

The context is denoted by the keyword *context* followed by the name of the type to which the operation belongs, and a double colon, followed by the complete operation signature, that is, the name of the operation, all parameters, their types, and the return type of the operation. Usually the complete operation signature is defined in the UML class diagram.

Following this context definition are lines—labeled with the keywords *pre*: and *post*:—that contain the actual pre- and postconditions respectively. The general case looks like this:

```
context Type1::operation(arg : Type2) : ReturnType
pre : -- some expression using the param arg and features of the
      -- contextual type
post: -- some expression using the param arg, features of the
      -- contextual type, the @pre keyword, and messaging
      -- expressions
```

The contextual instance is always an instance of the type for which the operation has been defined.

Note that in contrast to invariants, which must always be true, pre- and postconditions need be true only at a certain point in time: before, and after execution of an operation, respectively.

6.4.2 Body of Query Operations

Query operations can be fully defined by specifying the result of the operation in a single expression. By definition query operations have no side-effects. Executions of a query operation results in a value or set of values, nothing more. The context is indicated in the same manner as for pre- and postconditions. Instead of the keywords *pre* or *post*, the keyword *body* is used, followed by the body expression.

```
context CustomerCard::getTransactions(from : Date, until: Date )
                                    : Set(Transaction)
body: transactions->select( date.isAfter( from ) and
                            date.isBefore( until ) )
```

6.5 EXPRESSIONS IN INTERACTION DIAGRAMS

The way in which expressions may be used in UML interaction diagrams is explained in this section. The model elements that may be used in an expression in an interaction diagram, are all attributes, all query operations, all states, and all associations of the contextual type. What the contextual type is, depends on the position of the expression in the diagram.

6.5.1 Conditions

A message in a sequence or collaboration diagram can have an attached condition that specifies in what circumstances the message is sent. This condition can be written as an OCL expression. In this case the ocl expression is linked to an instance, not to a type. The contextual instance is the instance that sends the message, the source. The context is the message. The contextual type is the type of the contextual instance.

6.5.2 Actual Parameter Values

Messages in collaboration and sequence diagrams can take parameters. Because a message represents an operation call, these parameters are actual values. You can specify the actual value of such a parameter using an OCL expression. The context of the expression is the message. The contextual instance is the instance that sends the message. The contextual type is the type of the contextual instance.

Note that a message to an object in an interaction diagram must conform to an operation in the type of the target object, or to a signal that has been elsewhere defined. Likewise, the result message in an interaction diagram must conform to the type indicated as resulttype of the called operation.

6.6 EXPRESSIONS IN STATECHARTS

In UML statecharts expressions may be used in different ways. The model elements that may be used in an expression in a statechart, are all attributes, all query operations, all states, and all associations of the contextual type. What the contextual type is, depends on the position of the expression in the diagram.

6.6.1 Guards

OCL expressions can be used to express guards in statecharts. The context is the transition for which the guard is defined. The contextual type is the class to which statechart belongs. A guard is included in the statechart diagram itself. It is written between square brackets ('[' and ']') before the event coupled to a transition.

6.6.2 Target to Actions

An action represents either an operation call or the sending of event. An action is targeted at a specific object, or set of objects. An OCL expression can be used to identify the target of an action.

Actions may be linked to either transitions or states in a statechart. The context of the expression is the transition or state to which the action is linked. The contextual type is the type for which the statechart has been defined. The contextual instance is the instance of the contextual type for which a transition fires.

6.6.3 Actual Parameter Values

Actions in statechart diagrams can take parameters. Like the parameters to messages in an interaction diagram, these are actual values. You can specify the value of such a parameter using an OCL expression. The context of the expression is the action. The contextual type is the type for which the statechart has been defined. The contextual instance is the instance of the contextual type for which a transition fires. Note that the parameter values must conform to the operation called, or the signal sent.

6.6.4 Change Events

A *change event* is an event that is generated when one or more attributes or associations change value according to an expression. The event occurs whenever the value of the expression changes from false to true. A change event is denoted in the statechart by the keyword *when*, followed by the expression.

The condition of the change event can be written in OCL. The context of the expression is the transition. The contextual type is the type for which the statechart has been defined. The contextual instance is the instance of the contextual type for which a transition fires.

6.7 ACTIVITY DIAGRAMS

The various functions of expressions in UML activity diagrams, are explained in this section. The model elements that may be used in an expression in an activity diagram, are all attributes, all query operations, all states, and all associations of the contextual type. What the contextual type is, depends on the position of the expression in the diagram.

To be done: TBD.

6.8 COMPONENT DIAGRAMS

To be done: TBD.

6.9 USE CASES

In UML usecases pre- and postconditions may be used. These may also be written using OCL. Because use cases are an informal way of stating requirements, and OCL is a formal language, some adjustments need to be made when the pre- and postconditions of use cases are to be defined using OCL expressions.

6.9.1 Pre- and Postconditions

Although use cases can be considered to be operations defined on the complete system, we cannot identify the complete system as a type, because it is neither a class, an interface, a datatype, or component. Therefore, the pre- and postconditions of a use case have no contextual type, or contextual instance. As a consequence the keyword *self* can not be used.

Another consequence is that it is not clear which model elements may be referenced in the expression. Normally, all elements held by the contextual type may be used. Here, there is no contextual type. The model elements that may be used must be explicitly stated. This can be done by formalizing the types mentioned in the use case, e.g. Customer and Order, in an accompanying class diagram, and adding to the use case template a section called 'concerns' with a list of variable declarations, e.g. *newCustomer : Customer, newOrder : Order.*

Yet another consequence is that we cannot write a context definition, as there is no contextual type to be referenced. The OCL expressions may be included in the use case at the position indicated by the use case template.

6.10 CONSTRAINTS AND INHERITANCE

There are no explicit rules in the UML standard that explain whether or not an expression on a superclass is inherited by its subclasses. To use expressions in a meaningful way in a situation where inheritance plays a role, we need to give them proper semantics. The most widely accepted semantics of inheritance is to ensure that any instance of a subclass must behave the same as any instance of its superclass—as far as anyone or any program using the superclass can tell. This principle, called Liskov's Substitution Principle [Liskov94], is defined as follows:

Wherever an instance of a class is expected, one can always substitute an instance of any of its subclasses.

OCL expressions adhere to this principle. This section describes the consequences thereof for invariants, preconditions, and postconditions.

6.10.1 Consequences for Invariants

The invariants put on the superclass must always apply to the subclass too; otherwise, the substitution principle cannot be safely applied. The subclass may strengthen the invariant, because then the superclass invariant will still hold. The general rule for invariants is as follows:

> ***An invariant for a superclass is inherited by its subclasses. A subclass may strengthen the invariant but cannot weaken it.***

In the model in figure 6-2, we can define for the superclass *Stove* the invariant that its temperature must not be hotter than 200 degrees Celsius.

```
context Stove
inv: temperature <= 200
```

It would be dangerous if a subclass *ElectricStove* could exceed that maximum. For example, suppose that *ElectricStove* could have a temperature no hotter than 300 degrees Celsius:

```
context ElectricStove
inv: temperature <= 300
```

ElectricStove cannot be used safely in some places where *Stove* can be used. If I have a location that is fire-safe up to 250 degrees Celsius, I know I can safely put a *Stove* there. If I place a *Stove* at this location and if the *Stove* happens to be an *ElectricStove*, the place may be set on fire—definitely not a good idea.

Under some circumstances Liskov's Substitution Principle looks too restrictive. Subclasses may change superclass operations and add their own attributes and operations. In some cases the superclass invariants should be changed in correspondence with these alterations. Whether or not an invariant on a superclass

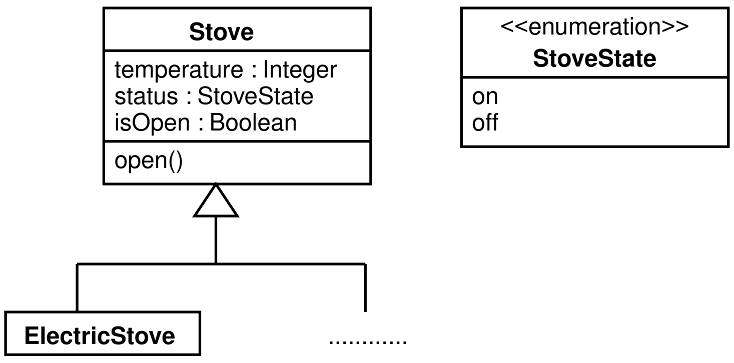

Figure 6-2 *Inheritance of invariants.*

needs to be changed when a new subclass is added, depends on the reason and contents of the invariant.

Take again the *Stove* example. If the invariant on the temperature is put on the *Stove* because its surroundings would catch fire if the temperature was too high, then a new subclass cannot weaken the invariant. If, on the other hand, the invariant is put on the *Stove* because of the materials used in the construction, then the invariant might be changed when a new subclass uses more fireproof materials. Thus, using the subclass would be safer, even when the invariant on the temperature is weakened. In this case, we recommend rewriting the temperature invariant so that it includes information about the materials used to construct the stove. This approach states the intention of the invariant more cleanly and removes the need to redefine it for each subclass.

6.10.2 Consequences for Pre- and Postconditions

When an operation is redefined in a subclass, the question is whether the pre- and postconditions of the original operation in the superclass still apply. To find the answer, we view the pre- and postconditions as the contract for the operation (see Section 3.3.5). The design by contract principle follows Liskov's Substitution Principle. The rules for pre- and postconditions are as follows:

A precondition may be weakened, not strengthened, in a redefinition of an operation in a subclass.

A postcondition may be strengthened, not weakened, in a redefinition of an operation in a subclass.

The following example illustrates these rules. We define the operation *open()* for the class *Stove* in figure 6-2 as follows:

```
context Stove::open()
pre : status = StoveState::off
post: status = StoveState::off and isOpen
```

This means that we expect to be able to open a *Stove* when its status is *off*. After we have opened the stove, we expect its status to be *off* and *isOpen* to be true. Suppose for the subclass *ElectricStove*, we redefine *open()* and give it different pre- and postconditions:

```
context ElectricStove::open()
pre : status = StoveState::off and temperature <= 100
post: isOpen
```

The precondition of the redefined *open()* includes an extra condition *(temperature <= 100)*. The consequence is that *ElectricStove* does not behave like a *Stove* anymore, because it won't be able to open under the conditions that a *Stove* will open *(status = StoveState::off)*. If we want to make sure that an *ElectricStove* can be substituted for a *Stove*, the precondition for the redefined *open()* cannot be strengthened. We could weaken the precondition, because then it will still work as expected for *Stove*.

The postcondition of *open()* in *ElectricStove* is weakened, because the condition *status = StoveState::off* has been removed. The consequence is that the *ElectricStove* won't fulfill the expectations of a *Stove*. After opening, the stove should be in status off. We could have strengthened the postcondition because then the original expectation would still be met.

6.11 SUMMARY

In this chapter ... TBD.

Chapter 7

Basic OCL Elements

This chapter describes the basic elements with which you can write constraints.

7.1 EXPRESSIONS, TYPES AND VALUES

In OCL, each value, whether it is an object, a component instance, or a datavalue, has a certain type, which defines the operations that can be applied on the object. Types in OCL are divided into the following groups.

- Predefined types, as defined in the standard library, including
  - basic types
  - collection types
- User defined types

The predefined basic types are *Integer*, *Real*, *String*, and *Boolean*, which are described in this chapter in some detail. Their definition is close to that in many known languages.

The predefined collection types are *Collection*, *Set*, *Bag*, *OrderedSet* and *Sequence*. They are used to specify the exact results of a navigation through associations in a class diagram. You need to be familiar with these types to write more complex expressions. Collection types and how to use them, will be described in Chapter 9.

User defined types, such as *Customer* or *LoyaltyProgram*, are defined by the user in the UML diagrams. Every instantiable model element, that is, each class, interface, component, or datatype, in a UML diagram is automatically a type in OCL. How to work with user defined types in OCL expressions is explained in Chapter 8.

Each OCL expression represents a value, therefore it has a type as well; either a user defined type or a predefined OCL type. Each OCL expression has a result: the value that results from evaluating the expression. The type of the result value is equal to the type of the expression.

7.1.1 Value Types and Object Types

OCL distinguishes between value types and object types. Both are types, i.e. both specify instances, but there is one important difference. *Value types* define instances that never change. The integer 1, for example, will never change its value and become an integer with a value of 2. *Object types* or classifiers represent types that define instances that can change their value(s). An instance of the class *Person* can change the value of its attribute *name* and still remain the same instance. Other names have been in use to indicate the same difference. For instance, Martin Fowler [Fowler97] calls object types *reference objects* and value types *value objects*.

Another important characteristic of value types has to do with identity. For value types the value identifies the instance, hence the name. Two occurrences of a value type that have the same value are by definition one and the same instance. Two occurrences of an object type are the same instance only if they have the same (object) identity. In other words, value types have value based identity, object types have reference based identity.

Both the predefined basic types and the predefined collection types of OCL are value types. The user-defined types can be either value types or object types. UML datatypes, including enumeration types, are value types. UML classes, components, and interfaces are object types.

7.2 BASIC TYPES AND OPERATORS

In the following sections we will define the basic predefined types and their operations. Basic types are typically not very interesting to read about, so this section is deliberately kept short. It explains only the more interesting operations of the basic types, those that are not exactly the same as in most programming languages and therefore offer some surprises.

7.2.1 The *Boolean* Type

A value of *Boolean* type can only be one of two values: *true* or *false*. The operations defined on *Boolean* include all the familiar ones, shown in Table 7-3. A standard

Table 7-3 *Standard operations for the Boolean type.*

Operation	Notation	Result Type
or	a or b	Boolean
and	a and b	Boolean

Table 7-3 *Standard operations for the Boolean type.*

Operation	Notation	Result Type
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implies	a implies b	Boolean

operation on the *Boolean* type that is uncommon to most programming languages—but often encountered in a more theoretical environment or in specification languages—is the implies operation (denoted by *implies*). This operation states that the result of the total expression is true if it is the case that, when the first *Boolean* operand is true, the second *Boolean* operand is also true. If the first *Boolean* operand is false, the whole *implies* expression always evaluates to true.

Take as an example the class *Service* from the R&L example. The result of the following sample expression is true if for every service it can be said that when it offers bonus points it never burns bonus points. In other words, a customer cannot earn bonus points when using a service that is bought with bonus points.

```
context Service
inv: self.pointsEarned > 0 implies not self.pointsBurned = 0
```

Another interesting operation on the *Boolean* type is the if-then-else. It is denoted in the following manner.

```
if <boolean OCL expression>
then <OCL expression>
else <OCL expression>
endif
```

The result value of an *if-then-else* operation is the result of either the OCL expression in the *then* clause or the OCL expression in the *else* clause, depending on the result of the boolean expression in the *if* clause. You cannot omit the *else* clause of the expression because an OCL expression must result in a value. Omitting the *else* clause causes the expression to result in an undefined state if the boolean OCL expression in the *if* clause is false. Both OCL expressions within the *else* and the *then* clauses must be of the same type.

Here are some other examples of valid *Boolean* expressions:

```
not true
age() > 21 and age() < 65
age() <= 12 xor cards->size > 3
title = 'Mr.' or title = 'Ms.'
name = 'Foobar'
if standard = 'UML'
   then 'using UML standard'
   else 'watch out: non UML features'
endif
```

7.2.2 The *Integer* and *Real* Types

The *Integer* type in OCL represents the mathematical natural numbers. Because OCL is a modeling language, there are no restrictions on the values of integers; in particular, there is no such thing as a maximum integer value. In the same way, the *Real* type in OCL represents the mathematical concept of real values. As in mathematics, *Integer* is a subtype of *Real*.

For the *Integer* and *Real* types, the usual operations apply: addition, subtraction, multiplication, and division. For both the *Integer* and the *Real* types, there is an additional operator *abs* that gives the absolute value of the given value; for example, *-1.abs()* results in 1, and *(2.4).abs()* results in 2.4. An additional operator on the *Real* type is the *floor* operator, which rounds the value of the *Real* down to an integer number; for example, *(4.6).floor()* results in an *Integer* instance with the value 4. The *round* operation on a *Real* results in the closest *Integer*; for example, *(4.6).round()* results in the *Integer* 5. An overview of all operations on integer and real values is given in Table 7-4.

The following examples illustrate the *Real* and *Integer* types. All these examples are expresssion of boolean type, that result in true.

```
2654 * 4.3 + 101 = 11513.2
(3.2).floor / 3 = 1
1.175 * (-8.9).abs - 10 = 0.4575
12 > 22.7 = false
12.max(33) = 33
33.max(12) = 33
13.mod(2) = 1
13.div(2) = 6
33.7.min(12) = 12
-24.abs = 24
(-2.4).floor = -3
```

7.2.3 The *String* Type

Strings are sequences of characters. Literal strings are written with enclosing single quotes, such as *'apple'* or *'weird cow'*. The operations available on *Strings* are *toUpper*, *toLower*, *size*, *substring*, and *concat* (see Table 7-5).

Table 7-4 *Standard operations for the Integer and Real types.*

Operation	Notation	Result type
equals	a = b	Boolean
not equals	a <> b	Boolean
less	a < b	Boolean
more	a > b	Boolean
less or equal	a <= b	Boolean
more or equal	a >= b	Boolean
plus	a + b	Integer or Real
minus	a - b	Integer or Real
multiplication	a * b	Integer or Real
division	a / b	Real
modulus	a.mod(b)	Integer
integer division	a.div(b)	Integer
absolute value	a.abs()	Integer or Real
maximum of a and b	a.max(b)	Integer or Real
minimum of a and b	a.min(b)	Integer or Real
round	a.round()	Integer
floor	a.floor()	Integer

Here are some examples to illustrate the *String* type. All these examples are expresssion of boolean type, that result in true.

```
'Anneke'.size() = 6
('Anneke' = 'Jos') = false
'Anneke '.concat('and Jos') = 'Anneke and Jos'
'Anneke'.toUpper() = 'ANNEKE'
'Anneke'.toLower() = 'anneke'
'Anneke and Jos'.substring(12, 14) = 'Jos'
```

Table 7-5 *Standard operations for the String type.*

Operation	Expression	Result type
concatenation	string.concat(string)	String
size	string.size()	Integer
to lower case	string.toLower()	String
to upper case	string.toUpper()	String
substring	string.substring(int,int)	String
equals	string1 = string2	Boolean
not equals	string1 <> string2	Boolean

7.3 PRECEDENCE RULES

When so many operations are available on an instance of a type, rules are needed to determine the precedence of the operations. Table 7-6 shows the OCL operations, starting with the highest precedence. In case of doubt, the use of parentheses () is always allowed to specify the precedence explicitly.

Table 7-6 *Precedence for OCL operations (highest to lowest).*

Name	Syntax
Pathname	::
Time expression	@pre
The dot and arrow operations	., ->
Unary operations	-, not
Multiplication and division	*, /
Addition and substraction	+, -
Relational operations	<, >, <=, >=, <>, =
Logical operations	and, or, xor
Logical implies	implies

7.4 USE OF INFIX OPERATORS

The use of *infix operators* is allowed in OCL. The operators '+', '-', '*', '/', '<', '>', '<>' '<=' '>=' are used as infix operators. If a user defined type includes one of those operators with the correct signature, they will also be used as infix operators. The correct signature includes only one parameter of the same type as the contextual instance. For the infix operators '<', '>', '<=', '>=', '<>', 'and', 'or', and 'xor' the return type must be Boolean. For the infix operators '+', '-', '*', and '/', the return type must be equal to the type of the contextual instance.

Conceptually the following two expressions are completely equal, both invoke the '+' operation on `a` with `b` as the parameter to the operation. The second notation is not allowed.

```
a + b
a.+(b)
```

7.5 COMMENTS

OCL expressions can contain *comments*. An OCL line comment begins with two hyphens. All text from the hyphens to the end of the line, is considered to be a comment. Comments longer than one line may be enclosed between '/*' and '*/'. For example, the following lines contain valid OCL expressions:

```
-- the expression 20 * 5 + 4 should be evaluated here

20 * 5 + 4 -- this is a comment

/* this is a very long comment that does not enlighten the reader
a bit about what the expression is really about */
```

The following line is not a valid OCL expression:

```
20 * -- this is a comment 5 + 4
```

7.6 SUMMARY

In this chapter we have shown the basic elements of the OCL language.

Chapter 8

User Defined Types

This chapter describes all the constructs in the language that use information from user defined types.

8.1 FEATURES OF USER DEFINED TYPES

When a user defined type is specified in an UML diagram, a number of features of that type are given. The features of a user defined type include:

- Attributes
- Operations
- Class Attributes
- Class Operations
- Associations ends that are derived from associations and aggregations[1]

Each feature can be used in an OCL expression. This section explains how the first four features may be used. Section 8.2 explains how to use the information in an association.

8.1.1 Attributes and Operations

Attributes of user defined types may be used in expressions by writing a dot followed by the attribute name. As with attributes, the operations of user defined types can be used in OCL expressions. However, there is one fundamental restriction. Because OCL is a side-effect-free language, operations that change the state of any object are not allowed. Only so-called *query operations*, which return a value but don't change anything, can be used. According to the UML specification, each operation has a boolean label called *isQuery*. If this label is true, the operation has no side effects and is allowed for use in OCL expressions.

[1] Note that inheritance relationships cannot be navigated, because they don't represent relationships between instances.

The dot notation used to reference attributes, is also used to reference operations. The name of the operation, however, is always followed by two parentheses, which enclose the optional arguments of the operation. Even if an operation has no arguments, the parentheses are mandatory. This is necessary to distinguish between attributes and operations, because UML allows attributes and operations to have identical names.

The visibility of attributes and operations is determined by the rules given in the UML specification. Thus a private attribute of an associated object may not be used in an OCL expression, because it is not visible to the contextual instance.

8.1.2 Class Operations and Attributes

Class operations and class attributes may also be used in OCL expressions. The syntax for referencing a class attribute or operation is the class name followed by two colons, followed by the attribute or operation name (and parameters). For example, in the R&L example in figure 2-1 the attribute *now* of *Date* is a class attribute. It can be used as in the following example.

```
context CustomerCard

inv: goodThru.isAfter( Date::now )
```

8.2 ASSOCIATIONS AND AGGREGATIONS

Another feature of a user defined type that we can use within OCL is derived from the associations in the class model. Every association has a number of association ends. Each end has a multiplicity, a type to which it is connected, an optional ordering marker, and it may have a name. This name is called a *rolename*. Conceptually, an association end is or defines a feature of the class connected to the other end(s) of the association.

Association ends can be used to navigate from one object in the system to another. Therefore the conceptual features that they define, are sometimes called *navigations.* If the name of the association end is missing, the name of the connected type may be used. If using the typename results in an ambiguity, the specification of a rolename is mandatory. Exactly the same holds for associations that are marked to be aggregations. In figure 8-1, which shows part of the Royal and Loyal model, *Customer* has two navigations: *programs* and *cards*. Class *CustomerCard* has one navigation named *owner,* just as class *LoyaltyProgram*: *customer.*

Navigations are treated in OCL as attributes. The dot-notation used to reference attributes is also used to reference navigations. In one type all names, whether attribute names or navigation names, must be unique. This arrangement prevents ambiguities between attribute and navigation names.

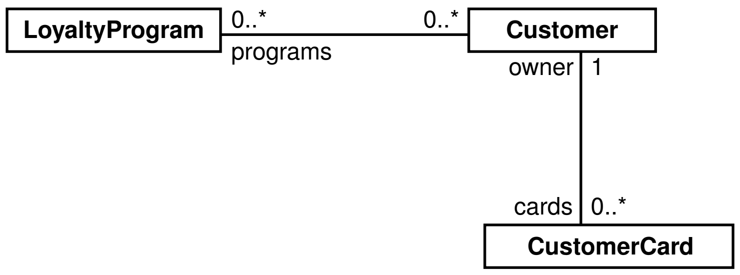

Figure 8-1 *Navigations.*

The type of a navigation is either a user defined type or a collection of user defined types. If the multiplicity of the association end is at most 1, the result type is the user defined type connected to the association end. If the multiplicity is higher than 1, the result is a collection. The elements in the collection must all be of, or conform to, the user defined type connected to the association end. In our example the result type of the *owner* navigation from *CustomerCard* is a user defined type: *Customer*. The result type of both the navigations *programs* and *cards* from *Customer* are collections, in this case (unordered) *Sets*. If the association end had been marked *{ordered}* the result type would have been *OrderedSet*. The differences between sets, ordered sets, bags, and sequences are described in Section 9.1.

For the diagram in figure 8-1 we can define an invariant that uses the *owner* navigation from the context of *CustomerCard*. Because this OCL expression results in a value of type *Customer*, all attributes, and operations on *Customer* that are visible to the contextual instance are now available for use in the remainder of the expression, as shown in the following example.

```
context CustomerCard

inv: self.owner.dateOfBirth.isBefore( Date::now )
```

Next to attributes and operations, all navigations defined on the *Customer* type may be used in the remainder of the expressions as well:

```
context CustomerCard

inv: self.owner.programs->size() > 0
```

The result of navigating more than one association with multiplicity *many* is by definition a *Bag*. If one of the navigations in the series is marked *{ordered}*, the result is a *Sequence*.

When we combine navigations, we have the means to navigate through the complete class diagram. From the context of one class in the class diagram we can write constraints on all connected classes. Surely, this is not good practice. The question of when and how to use navigations is answered in section 3.10.2.

8.2.1 Association Classes

A UML class diagram allows us to define *association classes*, which are classes attached to an association. From an association class you can always navigate to the instances of the classes at all ends of the association, using the same rules for naming as for normal navigations. Note that such a navigation always results in one single value and never in a collection of any kind.

In the R&L model (see figure 8-2) there is one association class: *Membership*. This class has three navigations: *programs* of type *LoyaltyProgram*, *participants* of type *Customer*, and —because of the extra association— *currentLevel* of type *ServiceLevel*. The following invariant states that the actual service level of a membership must aways be a service level of the loyalty program to which the membership belongs.

```
context Membership

inv: programs.level->includes( currentLevel )
```

It is also possible to navigate in the other direction: from the associated classes to the association class. In the R&L model, we can navigate from *Customer* and *LoyaltyProgram* to *Membership*. Because an association class cannot have a rolename, the navigation name is the name of the association class.

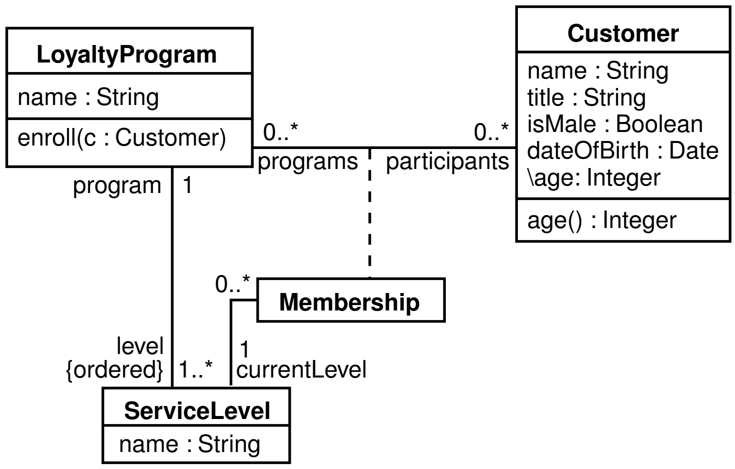

Figure 8-2 *Association class from the UML diagram.*

The multiplicity on the association ends is used to determine the type of the expression. If from a certain context the multiplicity on the opposite end is more than one, then the navigation to the association class will result in a collection of association class instances. If the multiplicity is one, or zero or one, then the navigation to the association class will result in one association class instance. For example, navigating from *Customer* to *Membership* will result in a value of type *Set(LoyaltyProgram)*.

The following invariant makes a statement similar to the preceding one but from the context of the *LoyaltyProgram*: the set of service levels must include the set of all actual levels of all memberships. Note that from the context of *LoyaltyProgram* the expression *Membership.currentLevel* is of type *Bag(ServiceLevel)*, therefore we use the operation *includesAll* instead of *includes*.

```
context LoyaltyProgram

inv: level->includesAll( Membership.currentLevel )
```

8.2.2 Qualified Associations

In a UML class diagram you can use *qualified associations*. Qualified associations can be used in OCL expressions in the same way that normal associations are used. The only difference is that we need a way to indicate the value of the qualifier in the expression. The syntax used for qualified associations is

```
object.navigation[qualifierValue, ...]
```

If there are multiple qualifiers, their values are separated by commas. You can navigate to all associated objects by not specifying a qualifier. This is identical to navigation of normal associations.

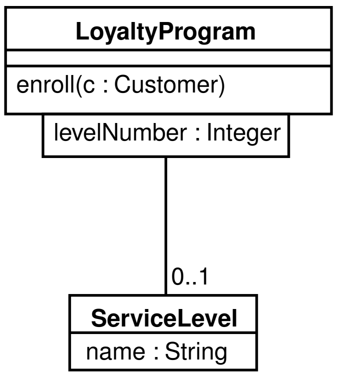

Figure 8-3 *Qualified association in the UML diagram.*

Figure 8-3 shows an alternative class diagram for the R&L model. The ordered association from *LoyaltyProgram* to *ServiceLevel* is replaced by an association with the qualifier *levelNumber*. This means that for each combination of a *LoyaltyProgram* and a *levelNumber* there is zero or one *ServiceLevel*. The *levelNumber* specifies the order of the *ServiceLevels.* To specify that the name of the *ServiceLevel* with *levelNumber* 1 must be *'basic'*, we can write the following invariant:

```
context LoyaltyProgram

inv: self.serviceLevel[1].name = 'basic'
```

If we want to state that there is at least one *ServiceLevel* with the name *'basic'*, disregarding the *levelNumber*, we can state the following invariant:

```
context LoyaltyProgram

inv: self.serviceLevel->exists(name = 'basic')
```

The first part, *self.serviceLevel*, is the collection of all *ServiceLevels* associated with the *LoyaltyProgram*. The *exists* operation states that at least one of those service levels must have its *name* attribute equal to *'basic'*.

8.2.3 Using Package Names in Navigations

Another aspect of modeling with UML is the ability to divide a model into different packages. Of course, associations between classes, types, or interfaces in different packages can be used in OCL expressions. In most cases, the item names used in the various packages are different from one another, and therefore associations between the items in two different packages can be treated in the manner described earlier. If name clashes occur, the item in the second package must be qualified by the package name. The syntax for using a package name is

```
PackageName::rolename
```

This phrase identifies an instance of the type that is identified by *rolename* in the package *PackageName*.

The R&L example is not large enough to make use of packages, but suppose for the sake of the example that the class *ProgramPartner* is part of a package called *PartnerAspects* and the class *LoyaltyProgram* is part of a different package called *ProgramAspects,* as depicted in figure 8-4. In this case, the following constraint can be formulated:

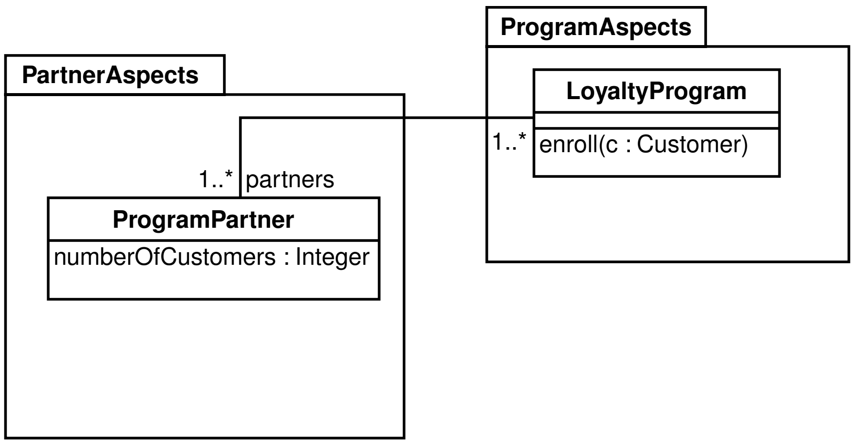

Figure 8-4 *Classes in different packages.*

```
context LoyaltyProgram

inv: self.PartnerAspects::partners->size() < 10
```

The meaning of this constraint is simply that the number of program partners is less than 10.

8.3 USING PATHNAMES IN INHERITANCE RELATIONS

The previous sections describe the common way to use features on instances. When multiple inheritance is used in a class diagram, features need not be unique. Different features with identical names can be inherited from different superclasses. Figure 8-5 shows an example of the class *CassetteBook*, which has two superclasses: *Cassette* and *Book*. The attribute *volume* is inherited twice, with different definitions. For a *Cassette*, *volume* is the strength of the audio signal on the tape. The *volume* of a *Book* is the number that the book has in the series to which it belongs.

The following invariant on *CassetteBook* is ambiguous, because we don't know which of the two *volume* attributes is meant.

```
context CassetteBook

inv: self.volume < 10
```

To resolve this ambiguity, we can use the *pathname* construct of OCL. A pathname allows us to prepend a feature with the inheritance path. The preceding invariant can be stated as

```
context CassetteBook

inv: self.Cassette::volume < 10
```

This shows that the *volume* attribute is the one inherited from *Cassette*.

8.4 ENUMERATION TYPES

An *enumeration type* is a special user defined type often used as a type for attributes. It is defined within the UML class diagram by using the enumeration stereotype, as shown in figure 8-6. The values defined in the enumeration can be used as values within an OCL expression. The notation to indicate one of the enumeration values in an OCL expression, is the enumeration type name, two colons, followed by the enumeration value identifier.

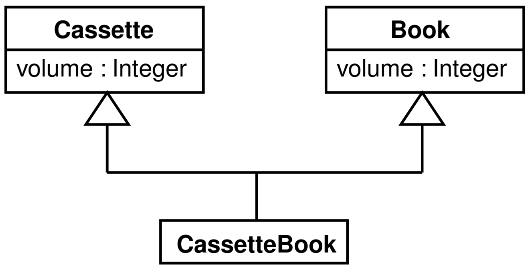

Figure 8-5 *Ambiguity caused by multiple inheritance.*

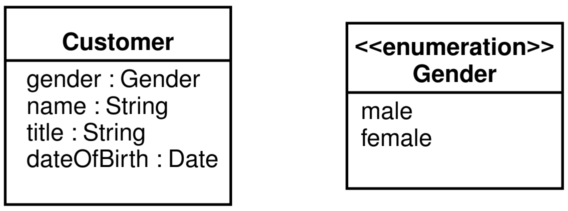

Figure 8-6 *Customer class with enumeration.*

In figure 8-6, the *Customer* class is shown again. Now we have changed the attribute *isMale* to an attribute *gender*. The following invariant states that male *Customer*s must be approached using the title *'Mr.'*.

```
context Customer

inv: gender = Gender::male implies title = 'Mr.'
```

The only operators available on enumeration values are the equality and inequality operators. They are denoted by = and <>, respectively.

8.5 SUMMARY

In this chapter we have shown how the information in the UML diagrams can be used to write expressions. In the expressions we can use all types defined in the UML diagrams. The information from the UML diagrams that can be used in expressions include attributes, operations, associations, aggregations, generalizations, association classes, and packages.

Chapter 9

Collection Types

This chapter describes the collection types that are part of the OCL standard library and the way they can be used in expressions.

9.1 THE COLLECTION TYPES

In object-oriented systems, the manipulation of collections (of objects) is a very common thing. Because one-to-one associations are rare, most associations define a relationship between one object and a collection of other objects. To let you manipulate these collections, OCL predefines a number of types for dealing with collections, sets, and so on.

Within OCL there are five collection types. Four of them—the *Set*, *OrderedSet, Bag*, and *Sequence* types—are concrete types and can be used in expressions. The fifth, the *Collection* type, is the abstract supertype of the other ones and is used to define the operations common to all collection types. In this book, we refer to *Set*, *OrderedSet, Bag*, and *Sequence* as the collection types.

The four concrete collection types are defined as follows:

- A *Set* is a collection that contains instances of a valid OCL type. A *Set* does not contain duplicate elements; any instance can be present only once. Elements in a *Set* are not ordered.
- An *OrderedSet* is a *Set* of which the elements are ordered.
- A *Bag* is a collection that may contain duplicate elements; that is, the same instance may occur in a bag more than once. A *Bag* is typically the result of combining navigations. This concept is introduced in Section 2.4.2 and explained further in Section 8.2. Elements in a *Bag* are not ordered.
- A *Sequence* is a *Bag* of which the elements are ordered.

Note that a *Sequence* or *OrderedSet* is ordered and not sorted. Each element has a sequence number, like array elements in programming languages. This does not mean that the element at a certain sequence number is in any sense less or greater than the element before. (See section 9.3.3.)

In the R&L model, the following navigations in the context of LoyaltyProgram result in a collection:

```
self.customer             -- Set(Customer)
self.serviceLevel         -- OrderedSet(ServiceLevel)
```

9.1.1 Collection Constants

Constant sets, orderedsets, sequences, and bags can be specified by enumerating their elements. Curly brackets should surround the elements of the collection, and the elements are separated by commas. The type of the collection is written before the curly brackets, as in the next examples.

```
Set { 1 , 2 , 5 , 88 }
Set { 'apple' , 'orange', 'strawberry' }
OrderedSet { 'apple' , 'orange', 'strawberry', ‘pear’ }
Sequence { 1, 3, 45, 2, 3 }
Sequence { 'ape', 'nut' }
Bag {1 , 3 , 4, 3, 5 }
```

Because of the usefulness of a Sequence of consecutive integers, there is a special way to specify them. The elements inside the curly brackets can be replaced by an interval specification, which consists of two expressions of type Integer separated by two dots (‘..’). This specifies a sequence all integers between the values of first and second integer expression, including the values of both expressions themselves:

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
-- are both identical to
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

9.1.2 Collection Type Expressions

```
Set(Customer)
Sequence(Set(ProgramPartners))
```

9.1.3 Collections Operations

There is a large number of standard operations defined on the collection types that manipulate collections in various ways. These operations are explained in the following sections.

All operations on collections are denoted in OCL expressions using an arrow, the operation following the arrow is applied to the collection before the arrow. This practice makes it possible for the user to define operations in the model that have the same names as the standard operations. The user defined operation is taken when it follows a dot, the standard operation is taken when it follows an arrow.

All collection types are defined as value types; that is, the value of an instance cannot be changed. Therefore, collection operations do not change a collection, but they may result in a new collection.

The following invariant, from the R&L model, uses the standard operation *size*, and states that the number of participants in a loyalty program must be less then 10,000.

```
context LoyaltyProgram
inv: self.customer->size() < 10000
```

Another invariant uses the standard operation *first*, and states that the first service level of any loyalty program is called *'basic'*.

```
context LoyaltyProgram
inv: self.level->first().name = 'basic'
```

9.1.4 Treating Instances as Collections

Because the OCL syntax for applying collection operations is different from that for user defined type operations, we can use a single instance as a collection. This collection is considered to be a *Set* with the instance as the only element. For example, in the R&L model from figure 2-1 the following constraint results in the value of the user defined operation *isEmpty()* of an instance of *LoyaltyAccount.*

```
context Membership
inv: loyaltyAccount.isEmpty()
```

On the other hand, the next constraint results in the value of the *Set* operation *isEmpty,* where the *LoyaltyAccount* is used as a collection.

```
context Membership
inv: loyaltyAccount->isEmpty()
```

This expression evaluates to true if the link from the instance of *Membership* to an instance of *LoyaltyAccount* is empty; that is, the *Membership* has no attached *LoyaltyAccount*.

9.1.5 Collections of Collections

A special feature of OCL collections is that in most cases collections are automatically *flattened*; that is, a collection does not contain collections but contains only simple objects. An example is given by the following two collections. The first is a collection of collections, the second is the flattened version of the first.

```
Set { Set { 1, 2 }, Set { 3, 4 }, Set { 5, 6 } }
Set { 1, 2, 3, 4, 5, 6 }
```

When a collection is inserted into another collection, the resulting collection is automatically flattened; the elements of the inserted collection are considered direct elements of the resulting collection. The reason for this approach is that collections of collections (and even deeper nested collections) are conceptually difficult to grasp, and are not often used in practice.

For those who do want to use nested collections, there is a way to maintain the layered structure when inserting a collection into a collection. The *collectNested* operation leaves the structure intact. See sections 9.3.10 to 9.3.12 for more information. The following sections give an explanation of the collection operations.

9.2 OPERATIONS ON COLLECTION TYPES

All collection types have the operations shown in Table 9-7 in common. These operations are defined by the abstract supertype *Collection*. Following are examples of the use of the *includes* and *includesAll* operations. In the invariant we specify that the actual service level of a membership must be one of the service levels of the program to which the membership belongs.

```
context Membership
inv: program.serviceLevel->includes(currentLevel)
```

The following example is an invariant which specifies that the available services for a service level must be offered by a partner of the loyalty program to which the service level belongs.

```
context ServiceLevel
inv: program.partners
             ->includesAll(self.availableServices.partner)
```

9.2.1 Operations with Variant Meaning

Some operations are defined for all collection types but have a slightly different, specialized meaning when applied to one type or another. Table 9-8 shows an overview of the operations with variant meaning defined on the four collection types. An 'X' mark indicates that the operation is defined for the given type, an '-' mark indicates that the operation is not defined for that type.

Table 9-7 *Standard operations on all collection types.*

Operation	Description
size()	The number of elements in the collection.
includes(object)	True if object is an element of the collection.
excludes(object)	True if object is not an element of the collection.
count(object)	The number of occurrences of object in the collection.
includesAll(collection)	True if all elements of the parameter collection are present in the current collection.
excludesAll(collection)	True if all elements of the parameter collection are not present in the current collection.
isEmpty()	True if the collection contains no elements.
notEmpty()	True if the collection contains one or more elements.
sum()	The addition of all elements in the collection. The elements must be of a type supporting addition (such as Real, Integer).

The *equals* and *notEquals* operations

The *equals* operator (denoted by =) evaluates to true if all elements in two collections are the same. For sets this means that all elements that are present in the first must be present in the second and vice versa. For ordered sets there is an extra restriction that the order in which the element appear must also be the same. For two bags to be equal, not only all elements must be present in both, but the number of times an element is present, must also be the same. For two sequences, the rules for bags apply, plus the extra restriction that the order of elements must be equal.

The *notEequals* operator (denoted by <>) evaluates to true if all elements in two collections are not the same. The opposite rules as for the *equals* operator apply. Both operations use the infix notation.

The *including* and *excluding* operations

The *including* operation adds one element to the collection. For a bag, this description is completely true. If the collection is a set or ordered set, then the element is

Table 9-8 *Collection Operations with Variant Meaning.*

Operation	Set	OrderedSet	Bag	Sequence
=	X	X	X	X
<>	X	X	X	X
including(object)	X	X	X	X
excluding(object)	X	X	X	X
flatten()	X	X	X	X
asSet()	X	X	X	X
asBag()	X	X	X	X
asOrderedSet()	X	X	X	X
asSequence()	X	X	X	X
union(coll)	X	X	X	X
intersection(coll)	X	-	X	-
-	X	X	-	-
symmetricDifference(coll)	X	-	-	-
append(object)	-	X	-	X
prepend(object)	-	X	-	X
insertAt(index, object)	-	X	-	X
subSequence(lower, upper)	-	X	-	X
at(index)	-	X	-	X
indexOf(object)	-	X	-	X
first()	-	X	-	X
last()	-	X	-	X

added only if it is not already present in the set. If the collection is a sequence or an ordered set, the element is added at the end.

The *excluding* operation removes an element from the collection. From a set or ordered set, it removes only one element. From a bag or sequence, it removes all occurrences of the given object.

The *flatten* operation

The *flatten* operation changes a collection of collections into a collection of single objects. When applied to a bag, the result is also a bag. This means that when a certain object is in more than one subcollection, that object will be included in the resulting bag more than once. When applied to a set, the result is also a set. Thus, when the object is in more than one subcollection, it is included in the result only once. For example, if the original collection is given by the first collection, then the result of the flatten operation would be the second collection.

```
Set { Set { 1, 2 }, Set { 2, 3 }, Set { 4, 5, 6 } }
Set { 1, 2, 3, 4, 5, 6 }
```

The result of the flatten operation executed on a bag instead of a set, is shown in the following example.

```
Bag { Set { 1, 2 }, Set { 1, 2 }, Set { 4, 5, 6 } }
Bag { 1, 1, 2, 2, 4, 5, 6 }
```

When the flatten operation is applied to a sequence or orderedset, the result is a sequence or ordered set (respectively). However, when the subcollections are either bags or sets, the order of the elements can not be determined precisely. In that case, the only thing that can be assured is that elements in a subset that comes before another subset in the original, also come before the elements of the second subset in the result. For example, if the original sequence is given by the first expression, then one of the possible results of the flatten operation could be the second sequence.

```
Sequence { Set { 1, 2 }, Set { 2, 3 }, Set { 4, 5, 6 } }
Sequence { 2, 1, 2, 3, 5, 6, 4 }
```

The elements of the subsets are randomly placed in the resulting sequence. There is no garantee that a second application of the flatten operation to the same sequence of sets will have the same result.

To be done: Check whether this is a deep (recursive) or shallow flatten.

The *asSet*, *asSequence*, *asBag*, *asOrderedSet* operations

Instances of all four concrete collection types can be transformed into instances of another concrete collection type. This can be done using one of the *asSet*, *asSequence*, *asBag*, or *asOrderedSet* operations. Application of *asSet* on a bag, or *asOrderedSet* on a sequence means that of any duplicate elements only one remains in the result. Application of *asBag* on a sequence, or *asSet* on an ordered set means that the ordering is lost. Application of *asOrderedSet*, or *asSequence* on a set or bag means the elements are placed randomly in some order in the result. Appying the operation on the same original twice does not garantee that both results will be equal.

The *union* operation

The *union* operation combines two collections into a new one. The union of a set with a set will result in a set, any duplicate elements are added to the result only once. Combining a set with a bag (and vice versa) results in a bag. A sequence or ordered set may not be combined with either a set or a bag, only with another ordered collection. In the result all elements of the collection on which the operation is called, go before the elements of the parameter collection.

The *intersection* operation

The *intersection* operation results in another collection that contains the elements that are in both collections. This operation is valid for combinations of two sets, a set and a bag, or two bags, but not for combinations involving a sequence or ordered set.

The *minus* operation

The *minus* operation (denoted by -) results in a new set that contains all elements that are in the set on which the operation is called, but not in the parameter set. This operation is defined for sets and ordered sets. When applied to an ordered set, the ordering remains. The minus operation uses an infix notation. Here are some examples:

```
Set{1,4,7,10} - Set{4,7} = Set{1,10}
OrderedSet{12,9,6,3} - Set{1,3,2} = OrderedSet{12,9,6}
```

The *symmetricDifference* operation

The *symmetricDifference* operation results in a set that contains all elements that are in the set on which the operation is called, or in the parameter set, but not in both. This operation is defined on sets only. Here is an example:

```
Set{1,4,7,10}.symmetricDifference(Set{4,5,7}) = Set{1,5,10}
```

9.2.2 Operations on *OrderedSets* and *Sequences* Only

All the operations that are defined only for the *OrderedSet* and *Sequence* types, have to do with the ordering of the elements. There are nine such operations.

- The *first* and *last* operations result in the first and the last elements of the collection, respectively.
- The *at* operation results in the element at the given position.
- The *indexOf* operation results in an integer value that gives the position of the element in the collection. When the element is present more than once in the collection, the result is the position of the first element.
- The *insertAt* operation results in a sequence or ordered set that has an extra element inserted at the given position.
- The *subSequence* operation may be applied to sequences only, and results in a sequence that contains the elements from the lower index to the upper index, including, in the original order.
- The *subOrderedSet* operation may be applied to ordered sets only. It result is equal to the *subSequence* operation, apart from the fact that it results in an ordered set instead of a sequence.
- The *append* and *prepend* operations add an element to a sequence as the last or first element, respectively.

Here are some examples:

```
Sequence{a,b,c,c,d,e}->first = a
OrderedSet{a,b,c,d}->last = d
Sequence{a,b,c,c,d,e}->at( 3 ) = c
Sequence{a,b,c,c,d,e}->indexOf( c ) = 3
OrderedSet{a,b,c,d}->insertAt( 3, X ) = OrderedSet{a,b,X,c,d}
Sequence{a,b,c,c,d,e}->subSequence( 3, 5 ) = Sequence{c,c,d}
OrderedSet{a,b,c,d}->subOrderedSet( 2, 3 ) = OrderedSet{b,c}
Sequence{a,b,c,c,d,e}->append( X ) = Sequence{a,b,c,c,d,e,X}
Sequence{a,b,c,c,d,e}->prepend( X ) = Sequence{X,a,b,c,c,d,e}
```

9.3 LOOP OPERATIONS

OCL has a number of standard operations that allow you to loop over the elements in a collection. These operations take each element in the collection and evaluate an expression on it. Loop operations are also called *iterators* or *iterator operations*. Every loop operation has an OCL expression as parameter. This is called the *body*, or *body parameter* of the operation. The following sections explain each of the loop operations in more detail. Table 9-9 shows an overview of the loop operations defined on the four collection types.

Table 9-9 *Loop operations on all collection types.*

Operation	Description
exists()	The number of elements in the collection.
forAll(object)	True if the object is an element of the collection.
isUnique(object)	True if the object is not an element of the collection.
sortedBy(object)	The number of occurrences of object in the collection.
any(collection)	True if all elements of the parameter collection are present in the current collection.
one(collection)	True if all elements of the parameter collection are not present in the current collection.
select()	True if the collection contains no elements.
reject()	True if the collection contains no elements.
collect()	True if the collection contains no elements.
collectNested()	True if the collection contains one or more elements.
iterate()	The addition of all elements in the collection. The elements must be of a type supporting addition (such as Real, Integer).

9.3.1 Iterator Variables

Every iterator operation may have an extra (optional) parameter, an *iterator variable*. An iterator variable is a variable that is used within the body parameter to indicate the element of the collection for which the body parameter is being calculated. The type of this iterator variable is always the type of the elements in the collection. Because the type is known it may be omitted in the declaration of the iterator variable. Thus, the next two examples are both correct. The following invariants state that the number of the loyalty account must be unique within a loyalty program.

```
context LoyaltyProgram
inv: self.Membership.account->isUnique( acc | acc.number )

context LoyaltyProgram
inv: self.Membership.account->isUnique( acc: LoyaltyAccount
                                         | acc.number )
```

It is recommended to use Iterator variables, when the type of the elements in the collection has features with the same names as the contextual instance. Suppose the class *LoyaltyProgram* itself has an attribute *number*. In that case leaving out the iterator variable, might render the above constraints ambiguous. Is the *number* parameter refering to the number of the loyalty program or to the number of the loyalty account?

In a loop expression namespaces are nested. The innermost namespace is that of the type of the elements of the collection, in this case *LoyaltyAccount*. When a name in the body parameter cannot be found in the innermost namespace, it will be searched in other namespaces. First, the namespaces of any enclosing loop expressions will be searched. Finally, the namespace of the contextual instance will be searched. Thus the following invariant is still a correct representation of the intended meaning.

```
context LoyaltyProgram
inv: self.Membership.account->isUnique( number )
```

When the body parameter should refer to the feature of the contextual instance, it should be prefixed 'self.'. In the above example this results in a rather stupid, but correct invariant. For every account in the program the number of the program itself will be tested. This number will be the same for every element of the collection, therefore the expression will always have *false* as result.

```
context LoyaltyProgram
inv: self.Membership.account->isUnique( self.number )
```

The iterator variable cannot be omitted in all circumstances. It can be omitted only if an explicit reference to the iterator is not needed in the expression. For example, the following expression cannot be rewritten without use of an iterator variable because of the reference to the iterator:

```
context ProgramPartner
inv: self.loyaltyProgram.partners->
                         select(p : ProgramPartner | p <> self)
```

This expression results in the collection of all program partners that are in the same loyalty programs as the context program partner.

9.3.2 The *isUnique* Operation

Quite often in a collection of elements we want a certain aspect of the elements to be unique for each element in the collection. For instance, in a collection of employees of a company the employee number must be unique. To state this fact we can use the *isUnique* operation. The parameter of this operation is usually a feature of the type of the elements in the collection. The result is either true or false. The operation will loop over all elements and will compare the values by calculating the parameter expression for all elements. If none of the values is equal to another the result is true, else the result is false. An example was given in the section 9.3.1.

9.3.3 The *sortedBy* Operation

In section 9.1 we mentioned that a *Sequence* or *OrderedSet* instance is ordered and not sorted. We can demand an ordering on the elements of any collection using the *sortedBy* operation. The parameter of this operation is a property of the type of the elements in the collection. For this property the *lesserThan* operation (denoted by '<') must be defined. The result is a *Sequence* or *OrderedSet*, depending on the type of the original collection. Applying this operation to a *Sequence* will result in a *Sequence*, applying this operation to a *OrderedSet* will result in a *OrderedSet*.

The operation will loop over all elements in the original collection and will order all elements according to the value derived from calculating the parameter property. The first element in the result is the element for which the property is the lowest.

Again we take as example the number of the LoyaltyAccount in the R&L system. The following invariant states that all loyalty accounts of a loyalty program are sorted by number.

```
context LoyaltyProgram
inv: self.Membership.LoyaltyAccount->sortedBy( number )
```

9.3.4 The *select* Operation

Sometimes an expression using operations and navigations results in a collection, but we are interested only in a special subset of the collection. The *select* operation allows us to specify a selection from the original collection. The result of the *select* operation is always a proper subset of the original collection.

The parameter of the *select* operation is a boolean expression that specifies which elements we want to select from the collection. The result of the *select* is the collection that contains all elements for which the boolean expression is true. The following expression selects all transactions on a *CustomerCard* that have more than 100 *points*.

```
context CustomerCard
inv: self.transactions->select( points > 100 )
```

We can explain the meaning of the *select* operation in an operational way, but the *select* is still an operation without side effects; it results in a new set. The result of *select* can be described by the following pseudocode:

```
element = collection.firstElement();
while( collection.notEmpty() ) do
    if( <expression-with-element> )
    then
        result.add(element);
    endif
    element = collection.nextElement();
endwhile
return result;
```

9.3.5 The *reject* Operation

The *reject* operation is analogous to the *select*, with the distinction that the *reject* selects all elements from the collection for which the expression evaluates to false. The existence of *reject* is merely a convenience. The following two invariants are semantically equivalent:

```
context Customer
inv: membership.account->select( points > 0 )

context Customer
inv: membership.account->reject( not (points > 0) )
```

9.3.6 The *any* Operation

To obtain any element from a collection for which a certain condition holds, we can use the *any* operation. The body parameter of this operation is a boolean expression. The result is a single element of the original collection. The operation will loop over all elements in the original collection and find one element that upholds the condition given by the body parameter. If the condition holds for more than one element, one of them is randomly choosen. If the condition does not hold for any element in the source collection, the result is undefined (see section 10.5).

Again we take as example the number of the LoyaltyAccount in the R&L system. The following expression from the context of LoyaltyProgram results in a loyalty account randomly picked from the set of accounts in the program that have a number lower than 10.000.

```
self.Membership.account->any( number < 10.000 )
```

9.3.7 The *forAll* Operation

Many times we want to specify that a certain condition must hold for all elements of a collection. The *forAll* operation on collections can be used for this purpose. The result of the *forAll* operation is a *Boolean*. It is true if the expression is true for all elements of the collection. If the expression is false for one or more elements in the collection, then *forAll* results in false. For example, consider the following expression:

```
context LoyaltyProgram
inv: participants->forAll( age() <= 70 )
```

This expression evaluates to true if the age of all participants in a loyalty program is less than or equal to 70. If the age of at least one (or more) customers exceeds 70, the result is false.

The *forAll* operation has an extended variant in which multiple iterator variables can be declared. All iterator variables iterate over the complete collection. Effectively, this is a short notation for a nested *forAll* expression on the collection. The next example, which is a complex way to expresse the *isUnique* operation, show the use of multiple iterator variables.

```
context LoyaltyProgram
inv: self.customer->forAll(c1, c2 |
                             c1 <> c2 implies c1.name <> c2.name)
```

This expression evaluates to true if the names of all customers of a loyalty program are different. It is semantically equivalent to the following expression, which uses nested *forAll* operations:

```
context LoyaltyProgram
inv: self.customer->forAll( c1 | self.customer->forAll( c2 |
                              c1 <> c2 implies c1.name <> c2.name ))
```

Although the number of iterators is unrestricted, more than two iterators are seldom used. The multiple iterators are allowed only with the *forAll* operation and not with any other operation that uses iterators.

To be done: Check whether this last remark is correct.

9.3.8 The *exists* Operation

Often, we want to specify that there is at least one object in a collection for which a certain condition holds. The *exists* operation on collections can be used for this purpose. The result of the *exists* operation is a *Boolean*. It is true if the expression is true for at least one element of the collection. If the expression is false for all elements in the collection, then the *exists* operation results in false. For example, in the context of a *LoyaltyAccount* we can state that if the attribute *points* is greater than zero, there exists a *Transaction* with *points* greater than zero.

```
context LoyaltyAccount
inv: points > 0 implies transaction->exists(t | t.points > 0)
```

Obviously, there is a relationship between the *exists* and the *forAll* operations. The following two expressions are equivalent.

```
collection->exists( <expression> )
not collection->forAll( not < expression> )
```

9.3.9 The *one* Operation

The *one* operation gives a boolean result stating whether or not there is exactly one element in the collection for which a condition holds. The body parameter of this operation, stating the condition, is a boolean expression. The operation will loop over all elements in the original collection and find all elements for which the condition holds. If there is exactly one such element then the result is true, else the result is false.

Again we take as example the number of the LoyaltyAccount in the R&L system. The following invariant states that there may be only one loyalty account, that has a number lower than 10.000.

```
context LoyaltyProgram
inv: self.Membership.account->one( number < 10.000 )
```

Note the difference between the *any* and *one* operations. The *any* operation can be seen as a variant of the *select* operation. Its result is an element selected from the source collection. The *one* operation is a variant of the *exists* operation. Its result is either true or false depending on whether or not a certain element exists in the source collection.

9.3.10 The *collect* Operation

The *collect* operation iterates over the collection, computes a value for each element of the collection, and gathers the evaluated values into a new collection. The type of the elements in the resulting collection is usually different from the type of the elements in the collection on which the operation is applied. The following expression from the context of LoyaltyAccount represents a collection of *Integer* values that holds the values of the *point* attribute in all linked *Transaction* instances.

```
transaction->collect( points )
```

We can use this expression to state an invariant on this collection of *Integer* values. For example, we could demand that at least one of the values must be 500.

```
context LoyaltyAccount
inv: transaction->collect( points )->
                         exists( p : Integer | p = 500 )
```

The result of the *collect* operation on a *Set* or *Bag* is a *Bag*; on an *OrderedSet* or *Sequence* the result is a *Sequence*. The result of the *collect* is always a flattened collection (see section 9.1.5).

9.3.11 Shorthand Notation for *collect*

Because the *collect* operation is used extensively, a shorthand notation has been introduced. This shorthand can be used only when there can be no misinterpretations. Instead of the preceding constraint, we can write

```
context LoyaltyAccount
inv: transactions.points->exist(p : Integer | p = 500 )
```

In this expression, *transactions* is a set of *Transaction*s; therefore, only the set properties can be used on it. The notation *transactions.points* is shorthand for *transactions->collect(points)*. Thus, when we take a property of a collection using a dot, this is interpreted as applying the collect operation, where the property is used as the body parameter.

9.3.12 The *collectNested* Operation

The *collectNested* operation iterates over the collection, like the *collect* operation. Whereas the result of the *collect* is always a flattened collection (see section 9.1.5), the result of the *collectNested* operation maintains the nested structure of collections within collections. The following expression from the context of LoyaltyProgram represents a collection of collections of *Service* instances.

```
self->collect(partners)->collectNested( deliveredServices )
```

9.3.13 The *iterate* Operation

The *iterate* operation is the most fundamental and complex of the loop operations. At the same time, it is the most generic loop operation. All other loop operations can be described as a special case of *iterate*. The syntax of the *iterate* operation is as follows:

```
collection->iterate( element : Type1;
                     result  : Type2 = <expression>
                   | <expression-with-element-and-result>)
```

The variable *element* is the iterator variable. The resulting value is accumulated in the variable *result,* which is also called the *accumulator.* The accumulator gets an initial value, given by the expression after the equal sign. None of the parameters is optional.

The result of the *iterate* operation is a value obtained by iterating over all elements in a collection. For each successive element in the source collection the body expression (*<expression-with-element-and-result>*) is calculated using the previous value of the *result* variable. A simple example of the iterate operation is given by the following expression, which results in the sum of the elements of a set of integers.

```
Set{1,2,3}->iterate( i: Integer, sum: Integer = 0 | sum + i )
```

In this case, we could have simplyfied the constraint by using the *sum* operation defined on *Integer*s. The *sum* operation is a shortcut for the specific use of the *iterate* operation.

A more complex example of the *iterate* operation can be found in the R&L model. Suppose that the class ProgramPartner needs a query operation that returns all transactions on services of that partner that burn points. It can be defined as followes.

```
context ProgramPartner
def: getBurningTransactions(): Set(Burning) =
     self.services.transaction->iterate(
        t         : Transaction;
        resultSet : Set(Burning) = Set{} |
        if t.isOclType( Burning ) then
            resultSet.including( t )
        else
            resultSet
        endif
```

```
          )
```

First a bag of transactions is obtained by collecting the transactions of all services of this partner. For every successive transaction in the bag the if-expression is evaluated. At the start *resultSet* is empty, but when a burning transaction is encountered, it is added to *resultSet*. If the current element is not a burning transaction, the result of the evaluation of this element is an unchanged *resultSet*. This value will be used in the evaluation of the next element of the source collection.

9.4 SUMMARY

In this chapter ... TBD.

Chapter 10

Advanced Constructs

This chapter describes all the constructs with which you can write more advanced expressions ranging from expressions that can only be used in postconditions to definitions of variables that are local to a context definition.

10.1 CONSTRUCTS FOR POSTCONDITIONS

There are two ways to write constraints for operations: preconditions and postconditions. In postconditions we can use a number of special constructs. Two special keywords represent, to some extent, the working of time: *result* and *@pre*. Another aspect of time is represented by messaging. The following sections explain all constructs that can be used in postconditions only.

10.1.1 The *@pre* Keyword

The *@pre* keyword indicates the value of an attribute or association at the start of the execution of the operation. The keyword must be postfixed to the name of the item concerned, as shown in the following example.

```
context LoyaltyProgram::enroll(c : Customer)
pre : not participants->includes(c)
post: participants = participants@pre->including(c)
```

The precondition of this example states that the customer to be enrolled is not already a member of the program. The postcondition states that the set of customers after the enroll operation is identical to the set of customers before the operation with the enrolled customer added to it. We could also add a second postcondition stating that the membership for the new customer has a loyalty account with zero points and no transactions.

```
post: membership->select(participants = c)->forAll(
          loyaltyAccount->notEmpty() and
```

```
                  loyaltyAccount.points = 0  and
                  loyaltyAccount.transactions->isEmpty )
```

10.1.2 The *result* Keyword

The keyword *result* indicates the return value from the operation. The type of *result* is defined by the return type of the operation. In the following example, the type of *result* is *LoyaltyProgram*.

```
context Transaction::getProgram():LoyaltyProgram
post: result = self.card.membership.programs
```

In this example, the result of the *getProgram* operation is the loyalty program against which the transaction was made.

10.1.3 The *oclIsNew* Operation

To determine whether an object is created during the execution of an operation the *oclIsNew* operation can be used. It returns true if the object to which it is applied did not exist at precondition time, but does exist at postcondition time. In the following example the postcondition states that in between pre- and postcondition time at least one customer has been added to the set of participants. This customer object is created between pre- and postcondition time, its name is equal to the parameter *n*, and its date of birth is equal to the parameter *d*.

```
context LoyaltyProgram::enrollNewCustomer( n : String, d: Date )
pre : -- none
post: let newCustomer : Customer =
                (participants - participants@pre)->any( true)
      in newCustomer.oclIsNew() and
         newCustomer.name = n and
         newCustomer.dateOfBirth = d
```

Note that a postcondition does not specify the statements in the body of the operation. There are many ways in which the postcondition can become true.

10.1.4 The *isSent* Operator

To specify that communication has taken place, the *isSent* operator (denoted as '^') may be used in postconditions. This operator takes a target object, and a message as operands. It has a boolean result. The next example states that the message *update(12, 14)* has been sent to the target object *observer* between pre- and postcondition time of the *hasChanged* operation.

```
context Subject::hasChanged()
post:  observer^update(12, 14)
```

As with messages in an interaction diagram *update* is either a call to an operation that is defined in the type of *observer*, or it is the sending of a signal specified in the UML model. The argument(s) of the message expression (12 and 14 in this example) must conform to the parameters of the operation or signal definition.

If the actual arguments of the operation or signal are not known, or not restricted in any way, they can be left unspecified. This is indicated by a question mark. Following the question mark is an optional type, which may be neccessary to find the correct operation when the same operation exists with different parameter types. For example:

```
context Subject::hasChanged()
post:  observer^update(? : Integer, ? : Integer)
```

This example states that the message *update* has been sent to *observer*, but that the actual values of the parameters are not known or not relevant.

Note that the postcondition does not state that the message has been received by the target. Depending on communication mechanism between source and target, some messages may get lost or be delivered later than postcondition time.

10.1.5 The *message* Operator

During execution of an operation many messages may have been sent that call the same operation, or send signals according to the same signal definition. To work with collections of messages, OCL defines a special *OclMessage* type. Any operation call or signal being send is virtually wrapped in an instance of *OclMessage*. One can obtain access to all *OclMessages* that wrap a matching call or signal, through the *message operator* (denoted as '^^'), as in the next example.

```
observer^^update(12, 14)
```

This expression results in the sequence of messages sent that match *update(12, 14)* being send to the object called *observer*. Each element of the sequence is an instance of *OclMessage*. Any collection operation can subsequently be applied to this sequence.

Because the message operator results in a sequence of instances of *OclMessage*, it can also be used to specify collections of messages sent to different targets. For instance:

```
context Subject::hasChanged()
post:  let messages : Sequence(OclMessage) =
               observers->collect(o |
                          o^^update(? : Integer, ? : Integer) )
       in messages->forAll(m | m.i <= m.j )
```

The local variable *messages* is a sequence of *OclMessage* instances build from the messages that match *update(? : Integer, ? : Integer)* and have been sent to one of the *observers*. Note that the collect operation flattens the sequence, so that its elements are of type *OclMessage*, and not of type *Sequence(OclMessage)*.

The message operator helps explain the *isSent* operator in another way. Virtually the *isSent* operator represents the application of the operation *notEmpty* on sequence of *OclMessages*. So the next postcondition is semantically equal to the postcondition in section 10.1.4.

```
context Subject::hasChanged()
post:  observer^^update(12, 14)->notEmpty()
```

In a postcondition with a message operand one can refer to the parameters of the operation or signal using the formal parameter names from the operation or signal definition. For example, if the operation *update* has been defined with formal parameters named *i* and *j*, we can write:

```
context Subject::hasChanged()
post: let messages : Sequence(OclMessage) =
                     observer^^update(? : Integer, ? : Integer) in
      messages->notEmpty() and
      messages->exists( m | m.i > 0 and m.j >= m.i )
```

In this example the values of the parameters *i* and *j* are not known, but some restrictions apply. First, *i* must be greater than zero, and second, the value of parameter *j* must be larger than or equal to *i*.

10.2 OPERATIONS OF THE OCLMESSAGE TYPE

The OclMessage type, as explained in section 10.1.5, has a number of operations. The operations defined for the *OclMessage* type are shown in Table 10-2.

10.2.1 The *hasReturned* and *result* Operations

Some messages have result values. A signal sent message is by definition asynchronous, so there never is a return value. If there is a logical return value it must be modeled as a separate signal message. Yet, for an operation call there is a potential return value, indicated by the return type in its signature.

In a postcondition the return value of an operation call is accessible from an *OclMessage* instance. It is only available if the operation has already returned upon postcondtion time. This is not always the case, because operation calls may be aynchronous.

Table 10-1 *Operations on any OCL instance.*

Expression	Result type
isSignalSent()	Boolean
isOperationCall()	Boolean
hasReturned()	Boolean
result()	return type of called operation

Therefore the OclMessage type has two operations: *hasReturned* and *result*. The *hasReturned* operation results in true if the operation call wrapped in the *OclMessage* instance has already finished executing and has returned a value. The *result* operation results in the return value of the called operation. For instance, if *getMoney* is an operation on class *Company* that returns a boolean, as in *Company::getMoney(amount : Integer) : Boolean*, we can write:

```
context Person::giveSalary(amount : Integer)
post: let message : OclMessage =
                        company^^getMoney(amount)->any( true )
      in message.hasReturned() and
         message.result() = true
```

If the *hasReturned* operation results in false, then the *result* operation will return undefined (see section 10.5).

10.2.2 The *isSignalCall* and *isOperationCall* operations

There are two other operations defined on the *OclMessage* type. The *isSignalCall* and *isOperationCall* operations can be used to distinguish whether a message corresponds to an operation or to a signal. Both have a boolean result, and no parameters.

10.3 LOCAL VARIABLES

Sometimes a sub-expression is used more than once in an expression. A so-called *let* expression allows one to define a local variable to represent the value of the sub-expression. The variable must be declared with a name, a type, and an expression that gives its value. The variable may be used only in the expression following the keyword *in*. A let expression may be included in any OCL expression. For example:

```
context Person
inv: let income : Integer = self.job.salary->sum()
     in
       if isUnemployed then
           income < 100
       else
           income >= 100
       endif
```

Let expressions may be nested, but a more convenient manner of defining a number of local variables is to include them in one let expression. For example:

```
context Person
inv: let income : Integer = self.job.salary->sum(),
         carSize : real = self.car.engine.maxPower
     in
       if isUnemployed then
           income < 100 and carSize < 1.0
       else
           income >= 100  and carSize >= 1.0
       endif
```

10.4 TUPLES AND TUPLETYPES

It is possible to compose several values into a *tuple*. A tuple consists of named parts, each of which can have a distinct type. Each tuple is a value itself. As OCL is a strong type language, each tuple has its own type. These tuple types do not have a name. They are either implicitly defined by a certain tuple value, or explicitly, but nameless, defined in for instance the type expression of local variable. Some examples of tuple values, and the way to write them, are:

```
Tuple {name: String = 'John',  age: Integer = 10}
Tuple {a: Collection(Integer) = Set{1, 3, 4},
       b: String = 'foo',
       c: String = 'bar'}
```

The parts of a tuple are enclosed in curly brackets, and they are separated by commas. The type names are optional, and the order of the parts is unimportant. Thus the following three expressions indicate the same tuple.

```
Tuple {name: String = 'John', age: Integer = 10}
Tuple {name = 'John', age = 10}
Tuple {age = 10, name = 'John'}
```

Some examples of TupleTypes, and the way to write them, are:

```
TupleType(name: String, age: Integer)
TupleType(a: Collection(Integer),
          b: String,
          c: String)
```

Here, the parts of the tuple are enclosed in normal brackets, again separated by commas. The type names are mandatory, but the order of the parts is still unimportant.

In a tuple value the values of the parts may be given by arbitrary OCL expressions, so for example we may write:

```
context Person
def: statistics : Set(TupleType(company: Company,
                                numEmployees: Integer,
                                wellpaidEmployees: Set(Person),
                                totalSalary: Integer)) =
     managedCompanies->collect(c |
        Tuple { company: Company = c,
                numEmployees: Integer = c.employee->size(),
                wellpaidEmployees: Set(Person) =
                   c.job->select(salary>10000).employee->asSet(),
                totalSalary: Integer = c.job.salary->sum()
              }
     )
```

The above expression results in a bag of tuples summarizing the company, number of employees, the best paid employees, and total salary costs of each company a person manages.

The parts of a tuple are accessed by their names, using the same dot notation that is used for accessing attributes. Thus the following two statements are true, if somewhat pointless.

```
Tuple {x: Integer = 5, y: String = 'hi'}.x = 5
Tuple {x: Integer = 5, y: String = 'hi'}.y = 'hi'
```

A more meaningful example, using the definition of *statistics* above, is:

```
context Company::isTopEmployee( p: Person ) : Boolean
body: statistics->sortedBy(totalSalary)
                  ->last().wellpaidEmployees->includes(p)
```

The operation *isTopEmployee* is a query operation that returns true if the given person is one of the best-paid employees of the company, and manages the highest total salary. In this expression, both 'totalSalary' and 'wellpaidEmployees' are accessing tuple parts.

10.5 UNDEFINED VALUES, THE OCLVOID TYPE

Some expressions will, when evaluated, have an undefined value. For instance, typecasting with the *oclAsType* operation to a type that the object does not support, or getting an element from an empty collection will result in undefined.

The undefined value is the only instance of the type *OclVoid*. The *OclVoid* type conforms to all types in the system. There is an explicit operation for testing if the value of an expression is undefined. The operation *oclIsUndefined* is an operation on *OclAny* (see section 10.9) that results in true if its argument is undefined and false otherwise.

In general, an expression where one of the parts is undefined will itself be undefined. There are some important exceptions to this rule, however. First, there are the logical operators:

- True or undefined = True
- False and undefined = False
- False implies undefined = True

The rules for *or* and *and* are valid irrespective of the order of the arguments and irrespective of whether the value of the other sub-expression is known or not. The if-expression is another exception. It will be valid as long as the chosen branch is valid, irrespective of the value of the other branch.

10.6 RE-TYPING OR CASTING

In some circumstances, it is desirable to use a feature that is defined on a subtype of the type that is expected at that point in the expression. Because the feature is not defined on the expected type, this results in a type conformance error.

When you are certain that the actual type of the object is the subtype, the object can be re-typed using the operation *oclAsType*, that takes as parameter a type name. This operation results in the same object, but the expected type within the expression is the indicated parameter. When there is an object *object* of type *Type1* and *Type2* is another type, it is allowed to write:

```
object.oclAsType(Type2) --- evaluates to object with type Type2
```

An object can only be re-typed to one of its subtypes; therefore, in the example, *Type2* must be a subtype of *Type1*.

If the actual type of the object is not a subtype of the type to which it is re-typed, the result of the expression is undefined (see section 10.5).

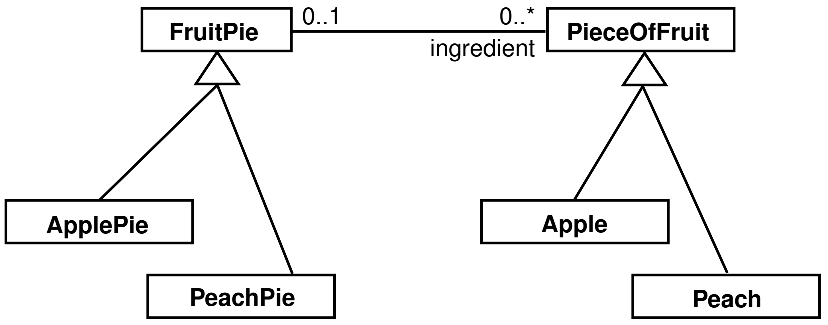

Figure 10-1 *Invariants on subtypes.*

10.7 TYPE CONFORMANCE RULES

OCL is a typed language. When you are constructing a new expression from other expressions, the subexpressions and the operator must "fit." If they don't, the expression is invalid, and any parser will complain that the expression contains a type conformance error. The definition of *conformance* that is used within OCL is as follows:

> ***Type1 conforms to Type2 if an instance of Type1 can be substituted at each place where an instance of Type2 is expected.***

These are the type conformance rules used in OCL expressions:

1. *Type1* conforms to *Type2* when they are identical.
2. *Type1* conforms to *Type2* when *Type1* is a subtype of *Type2*.
3. Each type is a subtype of *OclAny*.
4. Type conformance is transitive; that is, if *Type1* conforms to *Type2* and *Type2* conforms to *Type3,* then *Type1* conforms to *Type3*. Together with the first rule, this means that a type conforms to any of its predecessors in an inheritance tree.
5. *Integer* is a subtype of *Real* and therefore conforms to *Real*.
6. Every type *Collection(T)* is a subtype of *OclAny*. The types *Set(T)*, *Bag(T),* and *Sequence(T)* are all subtypes of *Collection(T)*.
7. *Collection(Type1)* conforms to *Collection(Type2)* if *Type1* conforms to *Type2*.
8. *Set(T)* does not conform to *Bag(T)* or *Sequence(T)*.
9. *Bag(T)* does not conform to *Set(T)* or *Sequence(T)*.
10. *Sequence(T)* does not conform to *Set(T)* or *Bag(T)*.

For example, in figure 10-1, *PeachPie* and *ApplePie* are two separate subtypes of *FruitPie*. In this example the following statements are true.

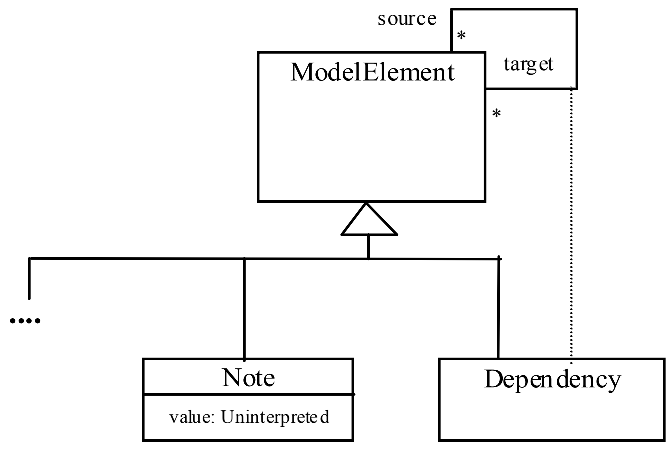

Figure 10-2 *Accessing Overridden features Example*

- *Set(ApplePie)* conforms to *Set(FruitPie)*.
- *Set(ApplePie)* conforms to *Collection(ApplePie)*.
- *Set(ApplePie)* conforms to *Collection(FruitPie)*.
- *Set(ApplePie)* does not conform to *Bag(ApplePie)*.
- *Bag(ApplePie)* does not conform to *Set(ApplePie)*.
- *Set(AppePie)* does not conform to *Set(PeachPie)*.

10.8 ACCESSING OVERRIDEN FEATURES

Whenever features are redefined within a type, the feature of the supertypes can be accessed using the *oclAsType()* operation. Whenever we have a class B as a subtype of class A, and a feature f1 of both A and B, we can write:

```
context B
inv: self.oclAsType(A).f1  -- accesses the f1 feature defined in A
inv: self.f1               -- accesses the f1 feature defined in B
```

Figure 10-2 shows an example where such a construct is needed. In this model fragment there is an ambiguity in the following OCL expression on class *Dependency*:

```
context Dependency
inv: self.source <> self
```

This can either mean normal association navigation, which is inherited from ModelElement, or it might mean navigation through the dotted line as an association class. Both possible navigations use the same role-name, so this is always ambiguous. Using *oclAsType()* we can distinguish between them with:

```
context Dependency
inv: self.oclAsType(Dependency).source->isEmpty()
inv: self.oclAsType(ModelElement).source->isEmpty()
```

10.9 THE OCLANY TYPE

A number of operations are useful for every type of OCL instance. Thereto the OCL standard library has a type called *OclAny.* When OCL expressions are being evaluated, the type *OclAny* is considered to be the supertype of all types in the model. All predefined types and all user defined types inherit the features of *OclAny.*

To avoid name conflicts between features from the model and the features inherited from *OclAny,* all names of the features of *OclAny* start with *ocl.* Although theoretically there may still be name conflicts, you can avoid them by not using the *ocl* prefix in your user defined types. You can also use the pathname construct (see Section 8.3) to refer to the *OclAny* features explicitly. The operations defined for all OCL objects are shown in Table 10-2.

10.9.1 Operations on OclAny

The *equals* and *notEquals* operations are redefined for most types in the Ocl standard library, and are explained elsewhere in this book. The *oclIsNew* operation is explained in section 10.1.3, the *oclAsType* operation is explained in section 10.6, and the *oclIsUndefined* operation is explained in section 10.5. The two other operations both allow access to the meta-level of your model, which can be useful for advanced modelers.

The *oclIsTypeOf* operation results in true only if the type of the object is identical to the argument. The *oclIsKindOf* operation results in true if the type of the object is identical to the argument, or identical to any of the subtypes of the argument. The following examples, which are based on figure 10-3 show the difference between the *oclIsKindOf* and *oclIsTypeOf* operations. For a *Transaction,* the following invariants are valid:

```
context Transaction
inv: self.oclType = Transaction
inv: self.oclIsKindOf(Transaction) = true
inv: self.oclIsTypeOf(Transaction) = true
inv: self.oclIsTypeOf(Burning) = false
```

Table 10-2 *Operations on any OCL instance.*

Expression	Result type
object = (object2 : OclAny)	Boolean
object <> (object2 : OclAny)	Boolean
object.oclIsUndefined()	Boolean
object.oclIsKindOf(type : OclType)	Boolean
object.oclIsTypeOf(type : OclType)	Boolean
object.oclIsNew()	Boolean
object.oclInState()	Boolean
object.oclAsType(type : OclType)	type
object.oclInState(str: StateName)	Boolean
type::allInstances()	Set(type)

```
self.oclIsKindOf(Burning) = false
```

For the subclass *Burning*, the following invariants are valid:

```
context Burning
inv: self.oclType = Burning
self.oclIsKindOf(Transaction) = true
self.oclIsTypeOf(Transaction) = false
self.oclIsTypeOf(Burning) = true
self.oclIsKindOf(Burning) = true
self.oclIsTypeOf(Earning) = false
self.oclIsKindOf(Earning) = false
```

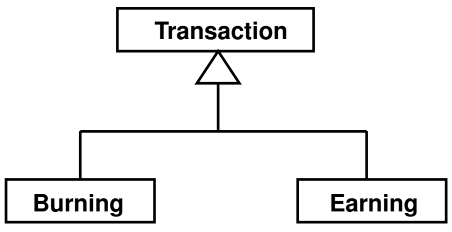

Figure 10-3 *Difference between oclIsKindOf and oclIsTypeOf.*

The *oclIsKindOf*, and *oclIsTypeOf* operations are often used to specify invariants on subclasses. For example, figure 10-1 shows a general association between *FruitPie* and *PieceOfFruit*. For the different subtypes of *FruitPie*, only specific subtypes of *PieceOfFruit* are acceptable.

Using *oclType* or one of the other operations, we can state the invariants for the subtypes of *FruitPie*: *ApplePie* and *PeachPie*.

```
context ApplePie
inv: self.ingredient->forAll(oclIsKindOf(Apple))

context PeachPie
inv: self.ingredient->forAll(oclIsKindOf(Peach))
```

The *oclInState* operation

The *oclInState* operation is operation that has been defined on the OclAny type, but in fact it is only useful for types that may have a statechart attached. It takes as parameter a state name, and results in true if the object is in that state. For nested states the statenames can be combined using the double colon '::'. In the example statemachine in figure 10-4, the state names used can be *On*, *Off*, *Off::Standby*, *Off::NoPower*. If the classifier of *object* has the above associated statemachine valid OCL expressions are:

```
object.oclInState(On)
object.oclInState(Off)
object.oclInstate(Off::Standby)
object.oclInState(Off::NoPower)
```

If there are multiple statemachines attached to the object's classifier, then the statename can be prefixed with the name of the statemachine containing the state and the double colon '::', as with nested states.

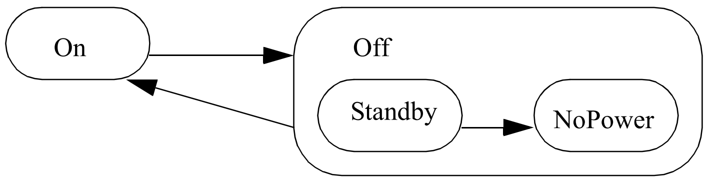

Figure 10-4 *Example statechart.*

10.9.2 The *allInstances* Operation

The *allInstances* operation is a class operation can only be applied to classes. In all other cases it will result in undefined. For a class it results in a set of all instances of that class, including all instances of its subclasses. The following expression results in the set of all instances of class *Transaction*, including all instances of type *Burning* and *Earning*.

```
Transaction::allInstances()
```

The use of *allInstances* is discouraged (see section 3.10.3).

10.10 PACKAGING EXPRESSIONS

Like every other model element OCL expressions may be included in a package. In general the model element that is the context of an OCL expression, is the owner of that expression, and the expression will be part of the same package as its context. When the expression is present in a diagram, it is obvious that the package that owns the expression is the package that owns its context. Because expressions are often written separate from the diagrams, it is not always clear to which package they belong.

To specify explicitly in which package an OCL expression belongs, there are two options. The context definition may contain a pathname indicating the package to which the context belongs, or the context definition may be enclosed between 'package' and 'endpackage' statements. The package statements have the following syntax, where the boldface words are keywords:

```
package Package::SubPackage

context X
inv: ... some invariant ...

context X::operationName(..)
pre: ... some precondition ...

endpackage
```

The above example has the same meaning as the following two expressions.

```
context Package::SubPackage::X
inv: ... some invariant ...

context Package::SubPackage::X::operationName(..)
pre: ... some precondition ...
```

An OCL file (or stream) may contain any number of package statements and any number of expressions enclosed in context definitions, thus allowing all invariants, preconditions and postconditions to be written and stored in one file. This file may co-exist with a UML model containing the diagrams as a separate entity.

10.11 SUMMARY

In this chapter ... TBD.

Appendix A

Glossary

Change event
An event that is generated when one or more attributes or associations change value according to an expression.

Context
The element in the part of the model that is specified by UML diagrams, that is linked to an OCL expression.

Context definition
Text or symbol that defines the relation between an OCL expression and the element in the part of the model that is specified by UML diagrams, and is linked to this OCL expression.

Contextual type
The type of the object for which an OCL expression will be evaluated.

Contextual instance
An instance of the contextual type of an OCL expression.

Contract
To be done.

Diagram
A visible rendering of (a part of) a model.

Derivation rule
A rule that specifies the value of a derived element.

Derived class
A class whose features can be derived completely from already existing (base) classes and other derived classes.

Design by contract
To be done.

Dynamic multiplicity
The multiplicity of the association is dynamic, when it should be determined based on another value in the system.

Features
The attributes, operations, and associations that are defined for a type.

Guard
A condition on a transition in a statechart.

Invariant
An invariant is a boolean expression attached to a type that states a condition that must always be met by all instances of the type for which it is defined.

Iterator variable
A variable that is used within the body of a loop expression to indicate the element of the collection for which the body is being calculated.

Language
A clearly defined way to model (parts of) a system. A definition of a language always consists of a syntax definition and a semantics definition.

Metamodel
A description or definition of a well-defined language. Equivalent to **Meta Language**.

Model
A consistent, coherent set of model elements that have features and restrictions.

Model Driven Architecture
TBD

Model driven software development
The process of developing software using different models on different levels of abstraction with (automated) transformations between these models.

Model repository
The storage of all model elements in an automated tool.

Navigation
TBD.

Object type
A type that has reference based identity.

Optional multiplicity
The multiplicity of an association is optional, when its lowest bound is zero.

Platform
A set of software pieces implemented with a specific technology on specific hardware, and/or specific hardware pieces that constitute the execution environment of a system.

Platform independent model
A model that contains no details that have meaning only within a specific platform.

Platform specific model
A model that contains details which have meaning only within a specific platform.

Postcondition
An expression attached to an operation that must be true at the moment that the operation has just ended its execution.

Precondition
An expression attached to an operation that must be true at the moment that the operation is going to be executed.

Query operation
An operation that has no side-effects.

Reference based identity
Instances have reference based identity when they are considered to be equal when two references refer to the same instance.

Semantics
A definition of the meaning of models that are well-formed according a syntax of a specific language.

Standard type
TBD.

Syntax
A set of rules that define which models are well-formed in a specific language.

System
A part of the world that is the subject of communication or reasoning.

Transformation
A transformation is the generation of a model based on another model and a set of transformation rules, while preserving the meaning of the source model in the target model insofar this meaning can be expressed in both models.

Type
A term used in this book to indicate either a class, a datatype, an interface, or a component.

Type conformance
TBD.

UML profile
TBD.

User defined type
TBD.

Value based identity
Instances have value based identity when they are considered to be equal when their values or the values of their parts are the same.

Value type
A type that has value based identity.

Appendix B

The Grammar Rules

This appendix describes the grammar for OCL context declarations and expressions. This section is taken from [OCL2002], which is the part of the UML 2.0 OMG standard that defines OCL. A free version of the Octopus tool, that implements this grammar is available from the OCL web site: www.object-constraint-language.info.

The grammar description uses the EBNF syntax, in which | means a choice , ? means optionality, * means zero or more times, and + means one or more times. In the description the syntax for lexical tokens is not made explicit. Instead it is indicated by <String>. This leaves the possiblity open to use the grammar togehter with different natural language alphabeths.

B.1 EBNF RULES FOR CONTEXT DECLARATION

```
contextDeclarationCS ::=
             attributeContextDeclCS
           | guardContextDeclCS
           | constraintContextDeclCS
```

```
attributeContextDeclCS ::=
             'context' pathNameCS ‘::’ simpleName
                            ':' typeCS 'init' ':'  OclExpression
           | 'context' pathNameCS ‘::’ simpleName
                            ':' typeCS 'derive' ':'  OclExpression
```

```
guardContextDeclCS ::= 'context' pathNameCS '::' simpleNameCS[1]
                                  -> simpleNameCS[2] 'guard' ':'
                                          OclExpressionCs
```

```
constraintContextDeclCS ::=
             invariantDeclCS
           | definitionDeclCS
           | operationDeclCS
```

```
invariantContextDeclCS ::=
             'context' pathNameCS
                  'inv' (simpleNameCS)? ':' OclExpressionCS
```

```
definitionContextDeclCS ::= 'context' pathNameCS
                                     'def' (simpleNameCS)? ':'
                                      AttributeDefinitionListCS?
                                      OperationDefinitionListCS?
```

```
operationContextDeclCS ::=
             'context' operationCS
                  'pre' (simpleNameCS)? ':' OclExpressionCS
           | 'context' operationCS
                  'post' (simpleNameCS)? ':' OclExpressionCS
```

```
operationCS ::= pathNameCS '::' simpleNameCS
                           '(' parametersCS? ')' ':' typeCS?
```

B.2 EBNF RULES FOR EXPRESSION

```
ExpressionInOclCS ::= OclExpressionCS
```

```
OclExpressionCS ::=
             PropertyCallExpCS
           | VariableExpCS
           | LiteralExpCS
           | LetExpCS
           | OclMessageExpCS
           | IfExpCS
```

```
VariableExpCS ::= simpleNameCS
```

```
simpleNameCS ::= <String>
```

```
pathNameCS ::= simpleNameCS ('::' pathNameCS )?
```

```
LiteralExpCS ::=
             EnumLiteralExpCS
           | CollectionLiteralExpCS
           | TupleLiteralExpCS
           | PrimitiveLiteralExpCS
```

```
EnumLiteralExpCS ::= pathNameCS '::' simpleNameCS
```

```
CollectionLiteralExpCS ::= CollectionTypeIdentifierCS
                           ‘{‘ CollectionLiteralPartsCS? ‘}’
```

```
CollectionTypeIdentifierCS ::=
              'Set'
            | 'Bag'
            | 'Sequence'
            | 'OrderedSet'
            | 'Collection'
```

```
CollectionLiteralPartsCS = CollectionLiteralPartCS
                         ( ',' CollectionLiteralPartsCS )?
```

```
CollectionLiteralPartCS ::=
             CollectionRangeCS
           | OclExpressionCS
```

```
CollectionRangeCS ::= OclExpressionCS '..' OclExpressionCS
```

```
PrimitiveLiteralExpCS ::=
             IntegerLiteralExpCS
           | RealLiteralExpCS
           | StringLiteralExpCS
           | BooleanLiteralExpCS
```

```
TupleLiteralExpCS ::= ‘Tuple’ ‘{‘ variableDeclarationListCS ‘}’
```

```
IntegerLiteralExpCS ::= <String>
```

```
RealLiteralExpCS ::= <String>
```

```
StringLiteralExpCS ::= ''' <String> '''
```

```
BooleanLiteralExpCS ::=
             'true'
           | 'false'
```

```
PropertyCallExpCS ::=
             ModelPropertyCallExpCS
           | LoopExpCS
```

```
LoopExpCS ::=
             IteratorExpCS
           | IterateExpCS
```

```
IteratorExpCS ::=
            OclExpressionCS '->' simpleNameCS
                     '(' (VariableDeclarationCS,
                          (',' VariableDeclarationCS)? '|' )?
                         OclExpressionCS
                     ')'
          | OclExpressionCS '.' simpleNameCS '('argumentsCS?')'
          | OclExpressionCS '.' simpleNameCS
          | OclExpressionCS '.' simpleNameCS ('[' argumentsCS ']')?
```

```
IterateExpCS ::= OclExpressionCS '->' 'iterate'
                 '(' (VariableDeclarationCS ';')?
                      VariableDeclarationCS '|'
                      OclExpressionCS
                 ')'
```

```
VariableDeclarationCS ::= simpleNameCS (':' typeCS)?
                          ( '=' OclExpressionCS )?
```

```
typeCS ::=
            pathNameCS
          | collectionTypeCS
          | tupleTypeCS
```

```
collectionTypeCS ::= collectionTypeIdentifierCS '(' typeCS ')'
```

```
tupletypeCS ::= 'TupleType' '(' variableDeclarationListCS? ')'
```

```
ModelPropertyCallExpCS ::=
            OperationCallExpCS
          | AttributeCallExpCS
          | NavigationCallExpCS
```

```
OperationCallExpCS ::=
            OclExpressionCS simpleNameCS OclExpressionCS
          | OclExpressionCS '->' simpleNameCS '(' argumentsCS? ')'
          | OclExpressionCS '.' simpleNameCS '(' argumentsCS? ')'
          | simpleNameCS  '(' argumentsCS? ')'
          | OclExpressionCS '.' simpleNameCS isMarkedPreCS
                                           '(' argumentsCS? ')'
          | simpleNameCS isMarkedPreCS '(' argumentsCS? ')'
          | pathNameCS  '(' argumentsCS? ')'
          | simpleNameCS OclExpressionCS
```

```
AttributeCallExpCS ::=
            OclExpressionCS '.' simpleNameCS isMarkedPreCS?
          | simpleNameCS isMarkedPreCS?
          | pathNameCS
```

```
NavigationCallExpCS ::= AssociationEndCallExpCS
          | AssociationClassCallExpCS
```

```
AssociationEndCallExpCS ::= (OclExpressionCS '.')? simpleNameCS
                                ('[' argumentsCS ']')? isMarkedPreCS?
```

```
AssociationClassCallExpCS ::=
            OclExpressionCS '.' simpleNameCS
                     ('[' argumentsCS ']')? isMarkedPreCS?
          | simpleNameCS
                     ('[' argumentsCS ']')? isMarkedPreCS?
```

```
isMarkedPreCS ::= '@' 'pre'
```

```
argumentsCS ::= OclExpressionCS ( ',' argumentsCS )?
```

```
LetExpCS ::= 'let' VariableDeclarationCS
                    LetExpSubCS
```

```
LetExpSubCS ::=
            ',' VariableDeclarationCS LetExpSubCS
          | 'in' OclExpressionCS
```

```
OclMessageExpCS ::=
            OclExpressionCS '^^'
                 simpleNameCS '(' OclMessageArgumentsCS? ')'
          | OclExpressionCS '^'
                 simpleNameCS '(' OclMessageArgumentsCS? ')'
```

```
OclMessageArgumentsCS ::= OclMessageArgCS (‘,' OclMessageArgumentsCS)?
```

```
OclMessageArgCS ::=
            '?' (':' typeCS)?
          | OclExpressionCS
```

```
IfExpCS ::= 'if'   OclExpression
                'then' OclExpression
                'else' OclExpression
                'endif'
```

```
AttributeDefinitionCS ::= 'attr' VariableDeclarationListCS
```

```
OperationDefinitionListCS ::= 'oper' OperationListCS
```

```
OperationListCS ::= OperationDefinitionCS (‘,’ OperationListCS)
```

```
OperationDefinitionCS ::= 'simpleNameCS '(' parametersCS? ')'
                                 ':' typeCS ('=' OclExpresionCS)?
```

```
parametersCS ::= variableDeclarationCS (‘,’ parametersCS )?
```

Appendix C

A Business Modeling Syntax for OCL

This appendix describes an alternative syntax for OCL expressions, one that is easier for business modelers. The alternative syntax is explained and formal mapping rules to the abstract syntax which is defined in the standard [BIBREF], are given.

The Octopus tool, already mentioned in appendix REF, implements this syntax next to the official one. The tool is available from the OCL web site: www.object-constraint-language.info. Note that this syntax is not standardized, it can not be read by any tool, but Octopus is able to generate it.

C.1 INTRODUCTION

In the past remarks have been made on the syntax of OCL. Some find it too difficult, others find it too different from mathematical languages. As it is, the OCL standard separates the concepts in the language from the concrete syntax used. This means that alternative concrete syntaxes can be made available. OCL expressions written in such an alternative syntax have exactly the same meaning as expressions written in the standard syntax. To give an example and prove the feasibility of this approach, a syntax has been deviced that is aimed at the business modeler.

The alternative syntax, from here on called business modeling syntax (BM syntax), resembles the outward appearance of SQL, but supports all of the concepts in OCL. The main difference between the official syntax and the business modeling syntax is that the notation for the predefined operations on collections and for the predefined iterators (loop expressions) is different. Another difference is that there is no implicit collect. Every collect operation needs to be explicitly stated. To give you a first 'feel' of the syntax, one of the expressions from Chapter 2 ("OCL By Example") is given in the business modeling syntax.

```
context Customer
inv: size of programs = size of
```

```
            select i: CustomerCard from cards where i.valid = true
```

In the standard OCL syntax this would be:

```
context Customer
inv: programs->size() = cards->select( i.valid = true )
```

All examples given in this appendix are based on the R&L example from Chapter 2 ("OCL By Example"). As a convenience we repeat the diagram for the R&L example in figure C-1.

C.2 INFORMAL DEFINITION

As indicated in the previous section the main difference between the standard syntax and the business modeling syntax lies in the predefined iterator expressions and the predefined collection operations. The following sections explain the alternative syntax for these two items. The syntax for the other language concepts remains for a large part the same as the standard syntax. The small number of differences are given in section C.2.3.

C.2.1 Iterators

Every OCL iterator (loop expression) has the following format according to the standard syntax.

```
<source> ->iterator( <iters> | <body> )
```

The terms between the < > brackets serve as placeholders, and *iterator* stands for the name of the iterator from the standard library. The placeholder <source> indicates the collection over which to iterate, <iters> stands for the iterator variables, and <body> stands for the body parameter of the iterator.

In the BM syntax each iterator has its own specialized format. Take for example the *select* iterator. Intuitively one would say that one selects a thing or things from a set taking into account any criteria. The BM syntax reflects this manner of speaking, as shown in the following expression, written in the context of LoyaltyProgram.

```
select pp: ProgramPartner from partners
    where pp.numberOfCustomers > 1000
```

The corresponding standard syntax is:

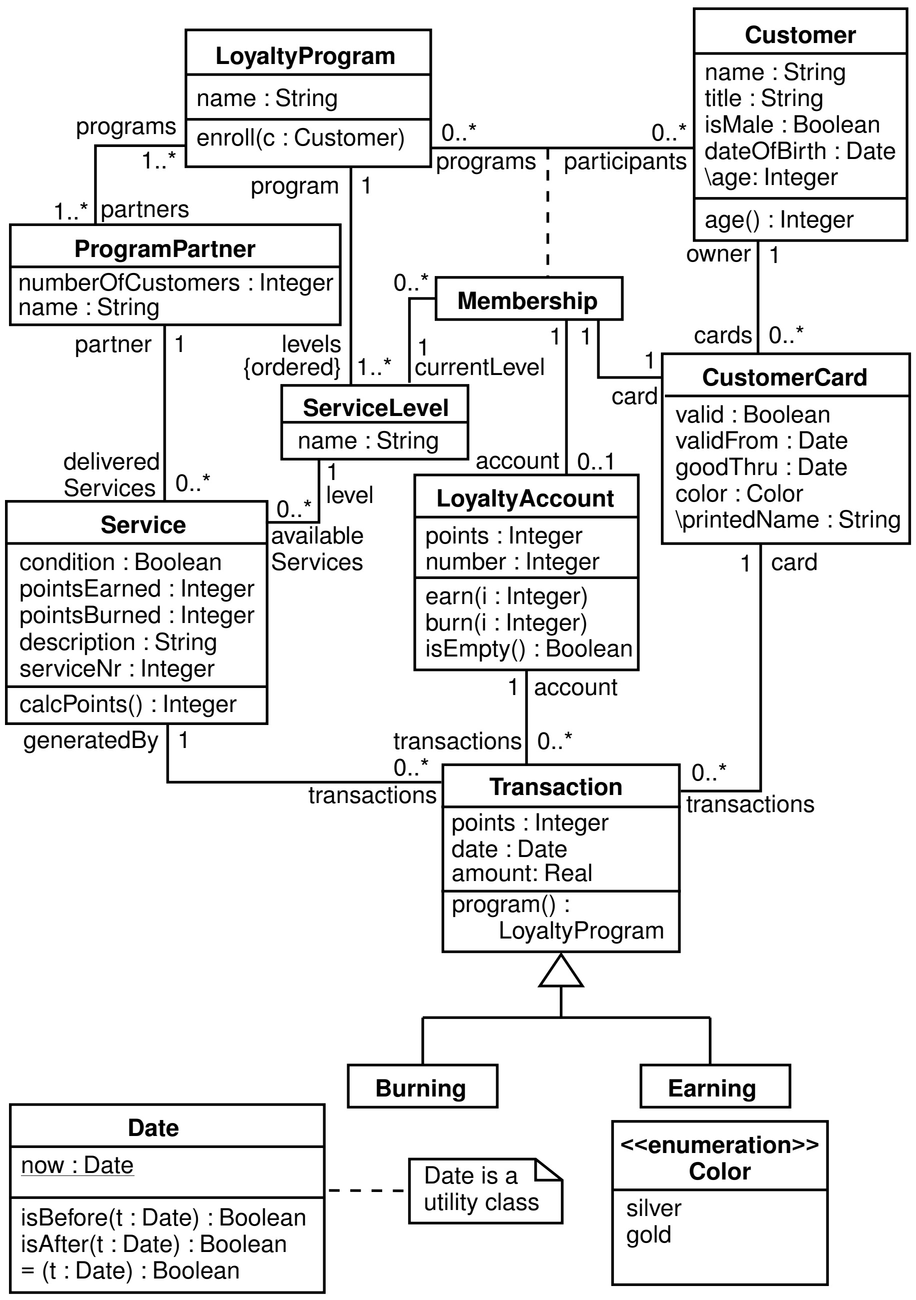

Figure C-1 *The Royal and Loyal model.*

```
partners->select(pp | pp.numberOfCustomers > 1000)
```

The *reject* and *any* iterators are written in a similar fashion, using keywords *reject* and *selectAny*, respectively.

The following expression, again in the context of LoyaltyProgram, is an example of the BM syntax for the *collect* iterator. It is build according to the intuition that one collects a thing (or things) from a set. To indicate the item to be collected you may use a reference to an iterator variable, which name and type are given after the keyword *using*. The *collectNested* iterator is written in the same manner, using the keyword *collectNested*.

```
collect p.deliveredServices using p: ProgramPartner
    from partners
```

The BM syntax for the *exists* and *one* iterator are build on the intuition that one asks whether an element exists in a set given some criteria. For example:

```
context CustomerCard
inv: exists t: Transaction in transactions
     where t.date.isBefore( Date::now )

context CustomerCard
inv: existsOne t: Transaction in transactions
     where t.date.isBefore( Date::now )
```

Remain the *forAll*, *sortedBy* and *isUnique* iterators. Examples of the BM syntax for these iterators are (all written in the context Customer):

```
forall c: CustomerCard in cards isTrue c.valid

sort cards using c: CustomerCard by c.goodThru

isUnique c.color using c: CustomerCard in cards
```

In Table C-1 you can find the general description of the alternative syntax for each iterator using the placeholders <source>, <iters>, and <body>. As in the standard syntax the iterator variables may be omitted. In that case the corresponding keyword is omitted as well. For instance:

```
forall cards isTrue valid

sort cards by goodThru

isUnique color in cards
```

The more general *iterate* expression is in the standard syntax defined as:

```
<source> ->iterate( <iters> ; <result> = <initialValue> | <body> )
```

Again the placeholder <source> indicates the collection over which to iterate, the <iters> stands for the iterator varables, and the <body> stands for the body of the iterator. Additionally there are the placeholders <result> and <initialValue>. The first stands for the result variable name and type declaration. The second holds the initial value of the result variable.

The general description of the alternative syntax for the iterate expression is included in Table C-1. An example is the following invariant, which is equal to the expression *transactions.points->sum()*:

```
context CustomerCard
inv: iterate t: Transaction over transactions
             result myResult : Integer
             initialValue 0
             nextValue myResult + points
```

Table C-1 *Business Modeling Syntax for Predefined Iterators*

iterator	**Business Modeling Syntax**
any	select any <iters> from <source> where <body>
collect	collect <body> using <iters> from <source>
collect nested	collect nested <body> using <iters> from <source>
exists	exists <iters> in <source> where <body>
forall	forall <iters> in <source> isTrue <body>
isunique	is unique <body> forall <iters> in <source>
iterate	iterate <iters> over <source> result <result> initial value <initialvalue> next value <body>
one	exists one <iters> in <source> where <body>
reject	reject <iters> from <source> where <body>
select	select <iters> from <source> where <body>
sortedby	sort <source> with <iters> by <body>

C.2.2 Collection operations

According to the standard syntax collection operations have the following format.

```
<source> -> operator( <arg1>, <arg2> )
```

Again we use placeholders to indicate parts of the expression. The place holder <source> gives the collection over which to iterate. The <arg1> and <arg2> stand for the arguments of the operation, which are both optional. The term *operator* stands for the name of the collection operation from the standard library.

All collection operations with no arguments are written as a keyword followed by the collection to which they are applied. The keyword is usually the equivalent of the operation name. The following expressions in the context *Customer* are examples of the BM syntax for collection operations with no arguments.

```
sizeOf cards
isEmpty programs
asSequence cards
lastOf asSequence cards
```

The BM syntax for collection operations with one argument uses an extra keyword next to the operation name. For example, the *including* operation is written using the keywords *is* and *includedIn*.

```
is self includedIn
    collect c.owner using c: CustomerCard from cards
```

The *union* operation uses the keywords *unionOf* and *with*:

```
unionOf
    select c: CustomerCard from cards where c.valid
with
    select c: CustomerCard from cards where not c.valid
```

The BM syntax for collection operations with two arguments uses two extra keyword next to the operation name. One of the keywords separates the arguments. For example, the *insertAt* operation is written using the keywords *insert*, *at* and *in*.

```
insert self at 3 in
    asSequence
        collect c.owner using c: CustomerCard from cards
```

In Table C-2 you can find the general description of the alternative syntax for each collection operation using the common placeholders.

Table C-2 *Business Modeling Syntax of Collection Operations*

operation	Business Modeling Syntax
append	append <arg1> to <source>
asBag	asBag <source>
asOrderedSet	asOrderedSet <source>
asSequence	asSequence <source>
asSet	asSet <source>
at	at <arg1> from <source>
count	count <arg1> in <source>
excludes	is <arg1> notIncludedIn <source>
excludesAll	isAllOf <arg1> notIncludedIn <source>
excluding	exclude <arg1> from <source>
first	firstOf <source>
flatten	flatten <source>
includes	is <arg1> includedIn <source>
includesAll	isAllOf <arg1> includedIn <source>
including	include <arg1> in <source>
indexof	index of <arg1> from <source>
insertAt	insert <arg1> at <arg2> in <source>
intersection	intersection <arg1> with <source>
isEmpty	isEmpty <source>
last	lastOf <source>
notEmpty	notEmpty <source>
prepend	prepend <arg1> to <source>
size	sizeOf <source>
subsequence	subsequence <arg1> to <arg2> of <source>
sum	sumOf <source>

Table C-2 *Business Modeling Syntax of Collection Operations*

operation	**Business Modeling Syntax**
symmetricDifference	symmetricDifference <arg1> with <source>
union	unionOf <arg1> with <source>

C.2.3 Other differences

The local variable definition (let expression) is one of the expressions that are written slightly differently in the BM syntax. As usual the difference lies in the use of keywords. The equal sign used to give the value of the variable is changed into the keyword *be*. For example:

```
context LoyaltyProgram
inv: let noc: Integer be
          collect numberOfCustomers from partners
     in forall pp: ProgramPartner in partners isTrue noc >= 10.000
```

Another difference between the standard and BM syntax is in the keyword indicating the value of a feature at precondition time. In the standard it is written as *@pre*, in the BM syntax it is written as *atPre*. For instance:

```
context Company::hireEmployee(p : Person)
post: employees = include p in employees atPre and
      stockprice() = stockprice atPre + 10
```

The last difference is the notation of the isSent and message operators. In the standard syntax they are written as '^' and '^^' respectively. In the BM syntax the keywords *isSentTo* and *sentTo* are used, and the order of target and message is switched. For example:

```
context File::save()
post: forAll b: Builder in self.project.builders
      isTrue incrementalBuild() isSentTo b

context Subject::hasChanged()
post:  let messages : Sequence(OclMessage) be
               collect update(? : Integer, ? : Integer) sentTo obs
               using obs
               from observers
       in forAll m in messages isTrue m.i <= m.j
```

C.3 SOME REMARKS ON THE RESEMBLANCE TO SQL

The BM syntax resembles the syntax of SQL. Still there are important differences between SQL and OCL. The main differenc eis that SQL statements work on complete tables. OCL expressions always take a single object as starting point. They work on what is visible from that object.

An often occurring mistake is to use the OCL *select* operation in the same manner as an SQL select statement. There is an important difference between the two. An OCL select operation results in a proper subset of the collection it was applied to. An SQL select statement does not result in a subset of the records in the table it is working on. In OCL terminology it collects values that are visible from the records in the table. Let's think, for the sake of the argument, that the R&L diagram in figure C-1 represents a database schema. The following SQL statement would result in list of Dates.

```
SELECT goodThru FROM CustomerCard WHERE valid = true
```

The OCL equivalent of this SQL statement would be:

```
collect cc.goodThru using cc: CustomerCard
from select cc: CustomerCard from allInstances of CustomerCard
     where cc.valid = true
```

C.4 SOME EXTENSIVE EXAMPLES

This section contains some extensive examples of the use of the BM syntax. An OCL expression is taken and written in the alternative notation. For sake of clarity the keywords used in the BM notation are written with capital letters.

The expression:

```
context Customer
inv: programs->size() = cards->select( valid = true )->size()
```

becomes:

```
context Customer
inv: size of programs = size of
    select i: CustomerCard from cards where i.valid = true
```

The expression:

```
context LoyaltyProgram
inv: partners.deliveredServices->forAll(
            pointsEarned = 0 and pointsBurned = 0 )
        implies membership.LoyaltyAccount->isEmpty()
```

becomes:

```
context LoyaltyProgram
inv: forall s : Service in
         collect p.deliveredServices
         using p : ProgramPartner from partners
     is true ( ( pointsEarned= 0 and pointsBurned= 0 )
            implies isempty
                   collect LoyaltyAccount
                   using l: LoyaltyAccount from membership )
```

The expression:

```
context LoyaltyProgram
def: getServices(levelName: String)
      = Servicelevel->select( name = levelName ).availableServices
```

becomes:

```
context LoyaltyProgram
def: getServices(String levelName)
     = collect s.availableServices
       using s: ServiceLevel from
              select sl: ServiceLevel
              from Servicelevel
              where sl.name = levelName
```

The expression:

```
context ProgramPartner
inv: deliveredServices.transactions
         ->select( isOclType( Burning ) )
                 ->collect( points )->sum() < 10,000
```

becomes:

```
context ProgramPartner
inv: sum of
    collect t.points
    using t : Transaction from
        collect s.transactions
        using s : Service from deliveredServices< 10,000
```

C.5 THE EBNF RULES

The grammar description gives alternative rules for only a small number of the non-terminals in appendix B, the rules for the other non-terminals still hold, and use the same EBNF syntax.

TBD: These rules are not yet complete. There should be rules for each iterator with disambiguating rules.

```
IteratorExpCS ::=
             simpleNameCS
                      (VariableDeclarationCS
                           (',' VariableDeclarationCS)? 'IN' )?
                     OclExpressionCS 'WHERE' '('? OclExpressionCS ')'?
           | OclExpressionCS '.' simpleNameCS '('argumentsCS?')'
           | OclExpressionCS '.' simpleNameCS
           | OclExpressionCS '.' simpleNameCS ('[' argumentsCS ']')?
```

```
IterateExpCS ::=
             'ITERATE' VariableDeclarationCS1
                          (',' VariableDeclarationCS2)?
             'OVER'    OclExpressionCS
             'RESULT'  VariableDeclarationCS3
             'NEXT VALUE'   '('? OclExpressionCS ')'?
     // VariableDeclarationCS3 moet 'INITIAL VALUE' bevatten ipv. '='
```

```
OperationCallExpCS ::=
             OclExpressionCS simpleNameCS OclExpressionCS
           | simpleNameCS '(' argumentsCS? ')' 'of' OclExpressionCS
           | simpleNameCS '(' argumentsCS? ')'
           | simpleNameCS isMarkedPreCS '(' argumentsCS? ')'
                                              'of' OclExpressionCS
           | pathNameCS  '(' argumentsCS? ')'
           | simpleNameCS OclExpressionCS
```

```
isMarkedPreCS ::= 'AT' 'PRECONDITION'
```

```
OclMessageExpCS ::=
             simpleNameCS '(' OclMessageArgumentsCS? ')'
                'SENT' 'TO' OclExpressionCS
           | simpleNameCS '(' OclMessageArgumentsCS? ')'
                'SENT' 'TO' OclExpressionCS
```

Bibliography

[Akehurst01] D.H. Akehurst and B. Bordbar, *On Querying UML Data Models with OCL*, <<UML>> 2001 - The Unified Modeling Language, Modeling Languages, Concepts and Tools, 4th International Conference, Toronto, Canada, 2001.

[Balsters03] H. Balsters and E.O. de Brock, *Derived Classes in UML/OCL applied to Relational Database Design*, to appear in Software and Systems Modeling, 2003.

[Blaha98] Michael Blaha and William Premerlani, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, 1998.

[Booch94] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings, 1994.

[CMM95]Carnegie Mellon University/Software Engineering Institute, *The Capability Maturity Model, Guidelines for improving the Software process*, Addison-Wesley, 1995

[Coleman94] Derek Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Chilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice-Hall, 1994.

[Cook94] Steve Cook and John Daniels, *Designing Object Systems—Object Oriented Modeling with Syntropy*, Prentice-Hall, 1994.

[D'Souza99] Desmond F. D'Souza and Alan C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley Longman, 1999.

[EJB01] *UML/EJB Mapping specification*, Java Community Process document JSR26, 2001

[Eriksson00] H. Eriksson and M. Penker, *Business Modeling with UML*, OMG Press, Wiley, 2000

[Fowler97] Martin Fowler, *UML Distilled: Applyig the Standard Object Modeling Language*, Addison Wesley Longman, 1997.

[Graham95] Ian Graham, *Migrating to Object Technology*, Addison-Wesley, 1995.

[Kleppe03] Anneke Kleppe, Jos Warmer, and Wim Bast, *MDA Explaine*d, *Practice and Promise of the Model Driven Architecture*, Addison-Wesley, 2003.

[Liskov94] Barbara Liskov and Jeanette Wing, "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems*, Vol 16, No 6, November 1994, pp. 1811–1841.

[Meyer85] Bertrand Meyer, “On Formalism in Specifications”, *IEEE Software*, January 1985.

[Meyer88] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[Meyer91] Bertrand Meyer, “Design by Contract,” in *Advances in Object-Oriented Software Engineering*, Prentice-Hall, 1991, pp. 1–50

[Meyer92] Bertrand Meyer, *Applying Design by Contract*, in IEEE Computer, october 1992.

[OCL97] *Object Constraint Language Specification*, version 1.1, OMG document ad970808, 1997.

[Richters01] Mark Richters, *A Precise Approach to Validating UML Models and OCL Constraints*, Logos Verlag Berlin, 2001

[Rumbaugh91] James Rumbaugh, Michael Blaha, William Premelani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[Selic94] Bran Selic, Garth Gullekson, and Paul T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.

[UML97] *UML 1.1 Specification*, OMG documents ad970802–ad0809, 1997.

[Waldén95] Kim Waldén and Jean-Marc Nerson, *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*, Prentice Hall, 1995.

[Wirfs-Brock90] Rebecca Wirfs-brock, Brian Wilkerson, and Lauren Wiener, *Designing Object-Oriented Software,* Prentice-Hall, 1990.

[Wordsworth92] J. Wordsworth, *Software Development with Z,* Addison-Wesley, Wokingham, Berkshire, 1992.

[Eriksson00] Hans-Erik Eriksson and Magnus Penker, *Business Modeling with UML, Business Patterns at Work,* Addison-Wesley, 2000.

Index