Großer Beleg

Java Code Generation for Dresden OCL2 for Eclipse

submitted by

Claas Wilke

born 16.04.1983 in Buxtehude

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: Dr.-Ing. Birgit Demuth
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted February 19, 2009

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DEN GROSSEN BELEG

Name des Studenten: **Claas Wilke**

Immatrikulations-Nr.: **3155376**

Thema: **Java-Codegenerierung für Dresden OCL for Eclipse**

Zielstellung:

Die Unified Modelling Language (UML) liefert mächtige Konzepte für die objektorientierte Modellierung einschließlich der Formulierung von Integritätsbedingungen mit Hilfe der Object Constraint Language (OCL). Ein Anwendungsfall dafür ist die modellgetriebenen Entwicklung von Java-Anwendungen. Aufbauend auf den Arbeiten der TU Dresden zum Dresden OCL Toolkit soll die Java-Codegenerierung für die neue Architektur von "Dresden OCL for Eclipse" reimplementiert werden.

Dazu sind folgende Teilaufgaben zu lösen:

- Einarbeitung in den Großen Beleg Eisenreich (Varianzanalyse zur Generierung imperativen Codes aus OCL-Ausdrücken)
- Analyse und Entwurf eines neuen Java-Codegenerators für die pivotmodellbasierte Architektur des Dresden OCL Toolkits auf Basis der Variationspunkte für die Codegenerierung (GB Eisenreich), dabei insbesondere
- Überarbeitung des Konzeptes der Instrumentierung von Java-Anwendungsprogrammen hinsichtlich eines aspektorientierten Ansatzes
- Erweiterung der Mächtigkeit des existierenden Java-Codegenerators (OCL22Java) hinsichtlich der Unterstützung des vollen Sprachumfanges, der durch den Dresden OCL2-Parser implementiert ist
- Implementierung und Test des neuen Java-Codegenerators
- Validierung und Diskussion der Ergebnisse

Betreuer: Dr. Birgit Demuth

Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Beginn am: 01.10.2008
Einzureichen am: 01.04.2009

Dresden, den 23.10.2008

Prof.Dr.rer.nat.habil. U. Aßmann
verantwortl. Hochschullehrer

Contents

1 Introduction **1**

2 The Dresden OCL Toolkit **3**
2.1 The Dresden OCL Toolkit . . . 3
2.2 The Dresden OCL2 Toolkit . . . 4
2.3 Dresden OCL2 for Eclipse . . . 4

3 Employed Programming Techniques **7**
3.1 Aspect-Oriented Programming . . . 7
3.2 StringTemplate . . . 8

4 Requirement Analysis **11**
4.1 The DOT2 and Requirement Analysis . . . 11
4.1.1 Variation of the Fragment Generation . . . 11
4.1.2 Variation of the Fragment Instrumentation . . . 13
4.1.3 Parametrization of the Code Generation . . . 14
4.2 Related Work . . . 16

5 Design and Fragment Generation **17**
5.1 The Architecture . . . 17
5.1.1 The Package Structure . . . 17
5.1.2 The Class Structure . . . 17
5.2 Type Mapping . . . 23
5.2.1 Primitive Types . . . 23
5.2.2 Enumerations . . . 24
5.2.3 Tuples . . . 25
5.2.4 Collection Types . . . 25
5.2.5 Special OCL types . . . 25
5.3 Fragment Generation . . . 26
5.3.1 Property Call Expressions . . . 26
5.3.2 Operation Call Expressions . . . 27
5.3.3 Collection Literal Expressions . . . 30
5.3.4 Iterator Expressions . . . 30

6 Fragment Instrumentation **33**
6.1 Initial and Derived Values . . . 33
6.1.1 Initial Values with Init . . . 33
6.1.2 Derived Values with Derive . . . 34
6.2 Method Implementation with Body . . . 34

6.3 Attribute and Method Definition with Def . . . 36
6.4 Preconditions . . . 38
6.5 Postconditions . . . 41
6.5.1 The Special Property @pre . . . 41
6.5.2 The Special Operation OclIsNew . . . 42
6.6 Invariants . . . 45
6.6.1 Strong Verification . . . 45
6.6.2 Weak Verification . . . 46
6.6.3 Transactional Verification . . . 48
6.7 The Special Operation AllInstances . . . 48

7 GUI Implementation and Test **51**
7.1 The Code Generation Wizard . . . 51
7.2 Tests on the Implementation . . . 51
7.2.1 Fragment Generation . . . 54
7.2.2 Fragment Instrumentation . . . 54
7.2.3 Performance Test . . . 54

8 Evaluation and Outlook on Future Works **57**
8.1 The Task of this Work . . . 57
8.2 The Provided Features . . . 57
8.2.1 Variation of the Fragment Generation . . . 57
8.2.2 Variation of the Fragment Instrumentation . . . 58
8.2.3 Parameterization of the Code Generation . . . 59
8.3 Outlook on Future Works . . . 59

A Type Mapping **61**

B Operation Mapping **63**

C Code Fragment Templates **69**

D The Royal and Loyal Example **77**

Bibliography **88**

Chapter 1

Introduction

Today, the *Object Constraint Language (OCL)* is an accepted and common technique to enrich UML models with constraints and model extensions. The code generation for object-oriented languages like Java from UML models is commonly used in all major case tools. But the support of OCL code generation is commonly missing or incomplete. The *Dresden OCL Toolkit* provides a collection of tools enabling software developers to extend their toolkits by using features developed for OCL such as an OCL parser or an OCL interpreter. The code generation of OCL constraints was supported by the *Dresden OCL Toolkit* in its last version which is outdated nowadays. Furthermore, the old code generator creates unreadable and complicate Java code which can not be refactored easily.

OCL and the *Dresden OCL Toolkit* have evolved and an usable code generator with a satisfying support of the OCL 2.0 standard is missing. This work will tackle that task. A new Java code generator which will support as much features of the OCL 2.0 standard as possible will be developed. Furthermore, this work will analyze how aspect-oriented techniques can be used to instrument generated code into existing model code in Java.

The *Object Constraint Language (OCL)* is a "standard add-on of the *Unified Modeling Language*" [WK04, p. 19]. OCL enables the software developer to extend his UML diagrams with additional and more precise information. New attributes, associations and methods can be defined, initial and derived values or operation bodies can be added. The main features of OCL provide the definition of constraints for preconditions (conditions which must be valid before the execution of a method), postconditions (conditions which must be valid after the execution of a method) and invariants (conditions which must be valid during the lifetime of an UML object). Such constraints are enforced and checked during the runtime of the constrained model code.

The first version of OCL was developed by IBM in 1995 [HH01]. In 1997 OCL became part of the *Unified Modeling Language (UML)* and was released as an *Object Management Group (OMG)* specification in the version 1.1 [Wik09] [OMG97]. In 1999 the development of *UML 2.0* and OCL 2.0 were started with the *UML 2.0 and UML 2.0 OCL Request for Proposals* [Wik09]. In September 2004 the 2.0 versions of *UML* and OCL were released. The OCL 2.0 specification has been published by the *OMG* and is available at [OMG06].

The major task of this work is the development of a new Java code gener-

ator for the *Dresden OCL Toolkit*. The minor thesis (Großer Beleg) of Katrin Eisenreich [Eis06] who already evaluated the code generator of the last toolkit version will be used as a base for the requirements analysis and the design of the new Java code generator. The new code generator will use *aspect-oriented programming* techniques to realize the fragment code instrumentation. A support of the OCL as far as supported by the *Essential OCL* meta model will be provided. The implementation will be tested and evaluated.

This work is structured as follows: The first part deals with the *Dresden OCL Toolkit* and its different versions which will be introduced in chapter 2. Afterwards, some programming techniques used for the new code generator and its generated code such as *aspect-oriented programming* will be presented in chapter 3. A requirement analysis based on the minor thesis of Katrin Eisenreich [Eis06] and the *OCL2J approach* [BDL04] will follow in chapter 4. In chapter 5 and 6, the main part of this work, the design of the code generator, the fragment code generation (including type the operation mapping) and the fragment code instrumentation will be explained. Chapter 7 will discuss briefly the implementation of a graphical user interface and will present the test suites which were used during development of the Java code generator. Finally, chapter 8 will evaluate the results of this work and will look at some tasks which could be realized in future works.

Some typographical conventions are used in this work to highlight special key words or language constructs:

- *Italics* are used to highlight important keywords and scientific terms.
- `Typewriter font` is used to sign model elements or language constructs of different programming and modeling languages such as *Java* or *OCL*.
- Blue color is used to denote hyperlinks between references in the digital publication of this work.

Chapter 2

The Dresden OCL Toolkit

The Dresden OCL Toolkit has been developed at the Technische Universität Dresden since 1999. Today, the toolkit is one of the major software projects at the chair of software technology and three different versions of the toolkit have already been released. Figure 2.1 shows a time line illustrating the different releases of OCL and the Dresden OCL Toolkit.

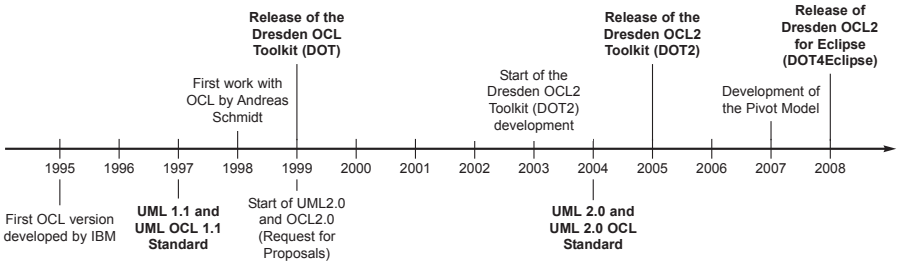

Figure 2.1: The different release of OCL and the Dresden OCL Toolkit.

2.1 The Dresden OCL Toolkit

A first work which examined OCL at the Techische Universität Dresden was done by Andreas Schmidt in 1998, examining how OCL could be mapped to the *Structured Query Language (SQL)* [Sch98]. The first base for the toolkit was realized by Frank Finger in 1999 [Fin99] [Fin00]. An OCL standard library and possibilities to load and parse UML models and OCL constraints and furthermore the possibility to generate Java code from OCL constraints were implemented. The instrumentation of the created Java code was realized by Ralf Wiebicke [Wie00]. This first released version of the toolkit was called *Dresden OCL Toolkit (DOT)*.

2.2 The Dresden OCL2 Toolkit

The development of the second version of the toolkit started in 2003. The basis of this version was created with the adaptation of the *DOT* to the *Netbeans MDR Repository* by Stefan Ocke [Ock03]. This version of the toolkit was named *Dresden OCL2 Toolkit (DOT2)*. It was based on the OCL standard in the version 2.0. The *DOT2* provided the loading and parsing of UML models and OCL constraints and the transformation of constrained models into SQL [Hei05] [Hei06].

The possibility to generate and instrument Java code for OCL constraints was adapted from the *DOT* to the *DOT2* by Ronny Brandt in 2006 [Bra06]. The *DOT2* is the last release of the *Dresden OCL Toolkit* which provides Java code generation and instrumentation. Thus the *DOT2* is the basis for the evaluation of the code generation in this work.

2.3 Dresden OCL2 for Eclipse

Since 2007 the *DOT2* has been replaced by a new version of the *Dresden OCL Toolkit*. The implementation of a *pivot model* by Matthias Bräuer [Brä07] made the newest version of the toolkit independent from specific repositories and it can therefore be adapted to many different meta models. By now, adaptions to the *Netbeans MDR Repository* used by the *DOT2*, to the UML2 meta model of the *Eclipse Model Development Tools Project* [MDT09] and to the *Ecore* meta model are supported.

In addition to the implementation of the pivot model an OCL parser to load and verify OCL constraints [Thi07] and an OCL interpreter [Bra07] were integrated. For this last release of the toolkit named *Dresden OCL2 for Eclipse (DOT4Eclipse)* the new code generator of this work will be developed.

The architecture of *DOT4Eclipse* is shown in figure 2.2. The architecture is the result of the work of Matthias Bräuer [Brä07] and can easily be extended. The architecture can be separated into three layers: The back-end, the toolkit basis and the toolkit tools.

The back-end layer represents the repository and the meta model which can easily be exchanged because all other packages of the *DOT4Eclipse* do not directly communicate with the meta model but use the *Pivot Model* which delegates all requests to the meta model instead. For example a possible meta model is the UML2 meta model of the *Eclipse Model Development Tools Project* [MDT09].

The second layer is the toolkit basis layer which contains the `Pivot Model`, `Essential OCL` and the `Model Bus`. The use of the `Pivot Model` was mentioned before. The package `Essential OCL` extends the `Pivot Model` and implements the OCL Standard Library to extend loaded models with OCL constraints. The package `Model Bus` loads, manages and provides access to models the user wants to work with.

The third layer contains all tools which are provided by the toolkit. By now this layer already contains the OCL interpreter and the OCL parser which uses the packages of the second layer to load, verify and interpret OCL constraints. The new code generator will be a third tool which will be located in the third layer and which will use the `Pivot Model`, the `Essential OCL` and the `Model`

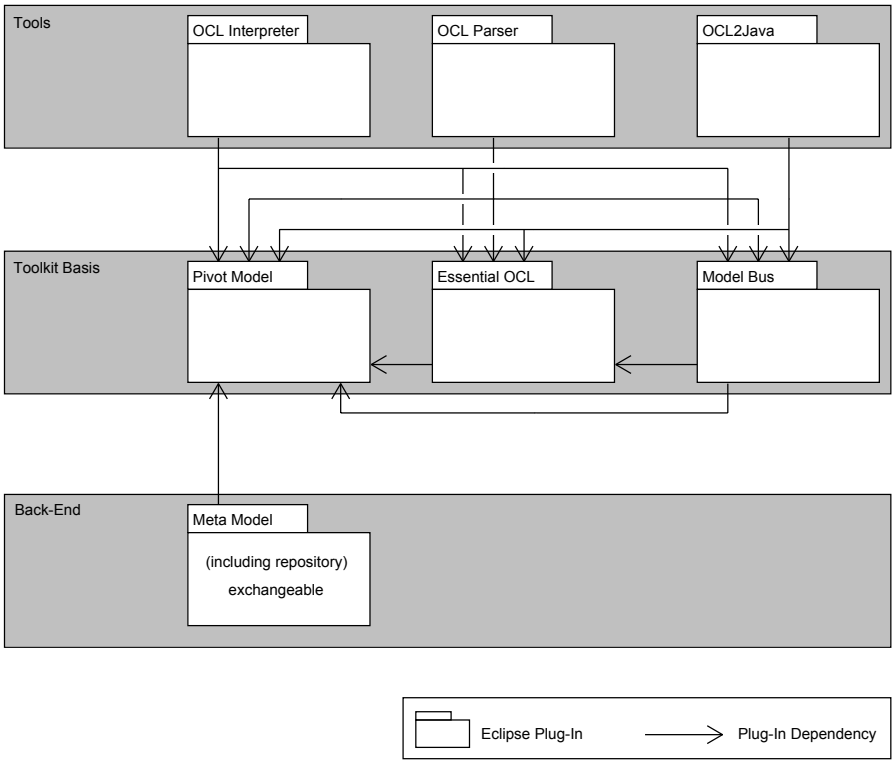

Figure 2.2: The architecture of DOT4Eclipse.

`Bus` package.

The *DOT4Eclipse* has been developed as a set of Eclipse plug-ins. All packages which are located in the *Toolkit Basis* and *Tools Layer* represent different Eclipse plug-ins. Additionally, the *DOT4Eclipse* contains some plug-ins to provide GUI elements such as wizards and examples to run the *DOT4Eclipse* with some simple models and OCL expressions.

Chapter 3

Employed Programming Techniques

This Chapter presents some techniques which were used to realize the newly developed code generator. At first, *aspect-oriented programming* will be explained and the language *AspectJ* will be presented. Afterwards the template engine *StringTemplate* will be introduced.

3.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a programming paradigm which solves some problems of object-oriented programming such as *crosscutting concerns* [AK07].

Crosscutting concerns are "design and implementation problems that cut across several places in [a] program" [AK07]. They occur in many software systems and their implementation leads to code tangling, scattering and code replication [AK07]. A typical example for crosscutting concerns are logging mechanisms which log some runtime activities such as the entry of any method. If such a logging mechanism will be implemented manually, any method of any class which will be logged has to be refactored by adding additional lines for the logging mechanism. If the logging mechanism shall be removed or adapted, any of these methods has to be refactored again. Another typical example for crosscutting concerns are constraint implementations [Eis06, p. 32].

AOP defines additional code fragments in separate files called *aspects*. An *aspect* is a piece of code which [AK07] [ABR06]:

1. Specifies one or several points in the code which cause events, if the control flow reaches these points (*join points*),
2. Specifies sets of join points (*pointcuts*),
3. Defines what happens if one of these events occur; meaning which code will be executed additionally at these points (*advices*).

Aspects are always defined in relation to a core, are specified separately from this core and are woven into this core by an aspect weaver [Aßm03, p. 260].

```
1  public aspect LoggingAspect {
2
3    protected pointcut
4      publicMethods(SampleClass aClass):
5        call(public SampleClass.*(..)) && this(aClass);
6
7    before(SampleClass aClass) : publicMethods(aClass) {
8      System.out.println("Entered a public method.");
9    }
10 }
```

Listing 3.1: A simple logging aspect.

For the logging example such an aspect must define a pointcut which describes the entry joinpoint of any method which shall be logged and an advice describing the additional logging code which shall be executed before the execution of any of these methods.

Some languages which realize aspect-oriented programming are *AspectJ*, *AspectC++* and *Eos* [AK07].

The developed code generator of *DOT4Eclipse* uses *AspectJ* to instrument the generated code fragments for OCL constraints into the Java classes of the constrained model. *AspectJ* will be introduced briefly in the following. "*AspectJ* is a seamless aspect-oriented extension to the Java programming language that enables clean modularization of [these] *crosscutting concerns*" [Asp09b].

Aspects defined by *AspectJ* are basically separated into pointcut definitions and advice definitions. A simple aspect is shown in listing 3.1. It describes all public methods of a class `SampleClass` in the pointcut `publicMethods(SampleClass)` and a simple advice which is executed before the call of these methods.

AspectJ provides instrumentation technologies for all constraint types defined in OCL. Post- and preconditions can be defined by using `before` or `after` advices. Properties can easily be observed by the pointcut types `get` and `set`. More keywords exist to support the realization of definitions, initializations and body expressions.

A detailed documentation of *AspectJ* is available in [Böh06] and at [Asp09b]. More details about the instrumentation of OCL constraints using *AspectJ* are presented in chapter 6.

3.2 StringTemplate

StringTemplate is "a java template engine to generate source code, [...] or other formatted text" [Str09]. It has already been used during the development of the *Ocl22Sql* code generation tool of the *DOT2* developed by Florian Heidenreich [Hei06]. Heidenreich evaluated different template engines and decided that *StringTemplate* was the template engine which satisfied the requirements of the code generation tool best [Hei06, p. 34f].

During this work *StringTemplate* is used for fragment generation and fragment instrumentation. Some simple examples for *StringTemplate* templates will now be explained. More information about *StringTemplate* is available at [Str09].

```
templateName(param1, param2, ...) ::=<<
This is some generated code using $param1$ and $param2$
at specified positions.
>>
```

Listing 3.2: A simple StringTemplate.

```
$if(param1)$
  This code contains $param1$.
$else$
  Param1 was not set.
$endif$
```

Listing 3.3: An if expression.

```
$values, types:{aValue, aType |
  $aValue$ is of type $aType$. }; separator = "\n"$
```

Listing 3.4: An iteration.

```
template1(param1) ::=<<
$template2(param1 = param1)$
```

Listing 3.5: A template referring to another template.

StringTemplates can be defined in text files and loaded into class instances of `StringTemplateAdapter` which I refactored from the class `DeclarativeTemplate` developed by Florian Heidenreich [Hei06, p. 42ff]. The templates can be bound with parameters by using the method `setAttribte("name", "value")` and the template can be converted into code using the method `toString()`.

The structure of a simple template which can be parameterized with the parameters `param1` and `param2` is shown in listing 3.2. The structure of an if expression which checks if the parameter `param1` is set and prints different code depending on this condition is shown in listing 3.3. Listing 3.4 shows an iteration over the parameters `values` and `types` which can be set multiple times using the method `setAttribute(..)` again and again. The code contained in the iteration is printed for any iteration of the two parameters. Finally, *StringTemplates* can refer to other templates. Listing 3.5 shows a template `template1(param1)` which refers to the template `template2(param1)`.

Chapter 4

Requirement Analysis

This chapter evaluates the Java code generator of the *Dresden OCL2 Toolkit (DOT2)* and compares it with another existing tool called *OCL2J*. Advantages and disadvantages of the code generators are presented and requirements and improvements are developed.

4.1 The DOT2 and Requirement Analysis

The Java code generator of the *DOT2* has already been evaluated during the minor thesis (Großer Beleg) of Katrin Eisenreich in 2006 [Eis06]. Katrin Eisenreich did a variance analysis of the code generator and developed a feature tree to compare different code generators. Furthermore, she evaluated the code generator of the *DOT2* using this feature tree and pointed out some possible improvements for following versions of the code generator. In the following the features pointed out by Katrin Eisenreich are presented and discussed.

The features of code generation were separated into three different categories [Eis06, p. 17ff]:

1. The variation of the fragment generation,
2. the variation of the fragment instrumentation,
3. and the configuration of the code generation.

4.1.1 Variation of the Fragment Generation

The category *variation of fragment generation* collects the following feature points [Eis06, p. 17ff]:

- The supported constraint types,
- coding conventions,
- the representation of OCL types,
- the access to model attributes in the generated code,
- and the technology to create the different code fragments which are instrumented during the code instrumentation.

Constraint Types

Generally, a code generator should support as much constraints of the OCL specification as possible. The new code generator which will be developed during this work has to support all constraint types which are supported by the *Essential OCL* meta model and the existing OCL parser of *DOT4Eclipse*.

The supported constraint types contain invariants, pre- and postconditions like the code generator of the *DOT2*, but also initial and derived values, definitions of new attributes and methods, body expressions for methods and let expressions to define temporary variables. Messages and state expressions will not be supported because they are not supported by *Essential OCL* [Thi07, p. 100].

Furthermore, the new code generator will support some special OCL operations like `oclIsNew()` and `oclAllInstances()` which are not supported by the code generator of the *DOT2*.

Coding Conventions

The feature point *coding conventions* describes the possibility to format the generated code to project specific coding conventions but also the possibility to create code for different programming languages.

The code generator developed during this work will only support the generation of Java code. But the code generator will be developed variably to provide the possibility to extend or adapt the code generator for code generation of other object-oriented programming languages. Coding conventions will not be supported by the code generator because features to format code have already been implemented in the *Eclipse Java Development Workbench*.

Representation of OCL Types

The *DOT2* defines special types in Java for all OCL types. The use of such an *OCL standard library* improves the code generation because the created fragments become very simple.

However, a standard library also causes some disadvantages: The created code contains overhead because a lot of the defined OCL types already exist in Java (for example primitive types like `String` or `Integer`). The created code contains more delegations to additionally defined Java objects and thus causes less performance than possible [BDL04, p. 11f].

The code generator developed during this work will not use a complete standard library to implement the OCL types in Java. All types which can be represented by already existing Java types will be directly mapped. Some exclusive types which can not be directly mapped to Java types will be implemented in new Java classes.

Attribute Access

A common problem during code generation is the access to class attributes and methods of the already existing model code. Java supports the declaration of private or protected attributes and methods which are not visible for other class instances. In [Eis06, p. 23ff] different solutions for attribute access are discussed.

The developed code generator will instrument the created code fragments using the aspect-oriented Java extension *AspectJ*. *AspectJ* supports an easy solution for the access problem by providing a keyword `privileged` which enables defined aspects to access private attributes and methods of affected class instances [BDL05, p. 2].

Fragment Generation

The creation of code fragments will be implemented as in the old code generator of the *DOT2* by traversing over the abstract syntax graph of the constrained model and its defined constraints.

The old code generator creates code fragments from strings which are directly implemented in the code of the code traversation [Eis06, p. 26]. Such a technology is both inefficient and hardly to refactor. Thus the new code generator will separate the traverse meachanism and the code generation by using templates which are parameterized during the code generations. The templates are externally saved. Thus the templates can be easily updated, exchanged and refactored.

4.1.2 Variation of the Fragment Instrumentation

The fragment instrumentation can be separated into the following features [Eis06, p. 29ff]:

- The code instrumentation,
- the instrumentation location,
- the reversibility of the instrumentation,
- the technology used for instrumentation,
- and the reaction on violated constraints during runtime.

This work will use the aspect-oriented language*AspectJ* to realize the implementation. Thus some of the remarked features are strongly related to each other an can not be evaluated independently. Fundamentally the *instrumentation technology* and the *reaction on constraint violation* remain.

Instrumentation Technology

The code generator of the *DOT2* directly inserts the code fragments into the affected classes. For some constraints new *wrapper methods* are added which call the methods constrained by the generated code [Wie00, p. 14ff]. This attempt causes some disadvantages: The source code of all affected classes has to be available because the code instrumentation does not work with Java byte code. In addition to this fact the direct code instrumentation causes code which can not be refactored easily. Constraint and model code can not be separated. An independent refactoring of model and constraint code is not possible. The reversibility of the instrumentation needs some precautions and marks in the instrumented code [Eis06, p. 34ff].

To avoid all the problems mentioned above, the new code generator will use *AspectJ* to realize the instrumentation. The instrumentation code is not directly

inserted into the Java source code but is declared in so-called *aspects* which are woven by an *Aspect Weaver* into the model code (see also section 3.1).

This technology has a lot of advantages: The fragment code can be instrumented into Java source code and Java byte code as well. The source code of all affected classes has not to be available [BDL04]. An independent refactoring of constraint and model code is possible, the reversibility of the instrumentation can be easily performed by executing the Java code without the aspect weaver. Moreover by using *AspectJ* the implementation of special OCL operations such as `oclAllInstances()` can easily be implemented which would be hard (if not impossible) in simple Java code [BDL04, p. 41ff].

Reaction on Constraint Violations

To react on violated constraints during the runtime of the constrained code, the new code generator will use the same technology as the code generator of the *DOT2*. The code generator will provide the possibility to configure a so-called *violation macro* which has already been introduced by Ronny Brandt [Bra06, p. 16]. The violation macro represents some lines of code and will be executed at any position in the code where a constraint is violated. Examples for violation macros are a simple print statement such as `System.out.println("Constraint was violated.");` or a throw statement of a new runtime exception such as `throw new RuntimeException("Constraint was violated.");`.

4.1.3 Parametrization of the Code Generation

The last group of features pointed out by Katrin Eisenreich is the parametrization or configuration of the code generation. Four major features belong to this group [Eis06, p. 43ff]:

- The selection of constraints for which code will be generated,
- the strength with which invariants will be verified,
- inheritance principles for constraints,
- and the configuration of violation macros.

The Selection of Constraints

The selection of constraints for code generation can be realized in a central configuration or decentral (for example by annotations in the constraint files). The configuration can be realized by manual selection of all constraints or by a rule based selection (for example all invariants could be selected) [Eis06, p. 43ff]. The new code generator of *DOT4Eclipse* will provide a GUI wizard which enables the user to do both, manual selection and rule based selection in a central configuration menu.

The Strength of Invariant Checks

A central question during constraint code generation is the question in which situations during runtime invariants will be checked and verified. Different options are possible which would be useful in different situations and user scenarios [Eis06, p. 45] [Wie00, p. 22] [BDL04, p. 20f] [BDF+04, p. 30]:

1. Invariants could be checked after the execution of any constructor and the change of any attribute or association which is in scope of the invariant condition. This range could be too strict in some scenarios, because programmers could store temporary values in attributes or associations during computation which violate constraints.
2. Invariants could be checked after the execution of any method of the constrained class. This variant could also be to strict for some user scenarios.
3. Invariants could be checked after the execution of any constructor or public method of the constraint class. This variant could be to liberal for some user scenarios.
4. Invariants could be checked after any method which was marked by the programmer for example via an annotation.
5. And finally, invariants could only be checked if the user calls a special method at runtime such as `checkConstraints()`. This scenario is similar to the transaction technique of database systems.

Due the fact that all the different possibilities have advantages and disadvantages, the new code generator of the *DOT4Eclipse* will enable the user to decide which verification technique he wants. The new code generator will provide the three scenarios (1), (3) and (5) called *strong verification*, *weak verification* and *transactional verification*.

These three scenarios could be useful for users in different situations. If a user wants to verify strongly that his constraints are verified after any change of any dependent attribute he should use *strong verification*. If he wants to use attributes to temporary store values and constraints should only be verified if any external class instance wants to access values of the constrained class, he should use *weak verification*. If the user wants to work with databases or other remote communication and the state of his constraint classes should be only validated before data transmission, he should use the scenario *transactional verification*.

The Inheritance of Constraints

Normally the inheritance of OCL constraints follows *Liskov's substitution principle* which declares that every constraint of a constrained class must also be checked for any subclass of the constraint class [WK04, p. 145] [BDL04, p. 21f.]. Postconditions and invariants can be strengthened during inheritance, preconditions can be weakened.

Enforcing Liskov's substitution principle during code generation is a very difficult task because the code generator has to check for any precondition to see if the precondition will be weakened by another constraint defined over any sub-class. Thus the new code generator will not follow Liskov's substitution principle. But the developed wizard will provide an option which will let the user decide whether or not invariants, pre- and postconditions shall be inherited.

The Reaction on Violated Constraints

As already mentioned, the new code generator will provide a violation macro technique to enable the user to decide how the constraint code will react on

violated constraints. This violation macro can be set generally for all constraints or individually for any constraint by the user [Eis06, p. 45f]. The new code generator will provide both possibilities in the code generation wizard.

4.2 Related Work

For all I know there is only one project which already tried to realize an aspect-oriented realization of an OCL code generator. This project is called the *OCL2J approach*. The *OCL2J approach* was developed at the Carleton University Ottawa, Canada in 2004 [BDL04] [BDL05] [DBL06]. The work intensively investigated the Java code generator of the *DOT* and studied which advantages and disadvantages would be gained by an OCL code generator which instrumentation technique is based on aspect-oriented programming. Unfortunately no source or byte code was available to test the *OCL2J* tool.

The tool uses the aspect-oriented programming language *AspectJ* and solves many of the problems and tasks mentioned during the evaluation of Katrin Eisenreich: Source and byte code weaving is possible using *OCL2J*, constrained and constraint code can be developed independently [BDL04, p. 17f].

Like the old code generator of the *DOT2*, the *OCL2J* tool supports code generation for invariants, pre- and postconditions. Let and body expressions, enumerations, defined and derived values are not supported. The tool does not use an OCL standard library for the OCL type representations in Java, but tries to map as much types as possible directly to Java types. Primitive types in OCL are separated into primitive and wrapper types in Java [BDL04, p. 18, 32ff].

Special OCL operations and properties like `@pre`, `allInstances()` and `oclIsNew()` are supported [BDL04, p. 8, 29, 39ff]. The code generator uses Java reflection to resolve information from the provided model source or byte code [BDL04, p. 19f]. Invariants are checked after constructors and before and after the execution of public methods of the constrained class [BDL04, p. 20]. Reactions on constraint violations are runtime exceptions or error messages [BDL04, p. 50]. The tool was intensively tested using the *royal and loyal model* developed by Warmer and Kleppe [WK04, p. 39ff] [BDL04, p. 50ff, 95f] which can be found in appendix D.

All things considered the *OCL2J* approach presents a good basis for a code generator implementation using aspect-oriented technologies for code instrumentation. The tool provides many solutions for tasks and problems which have to be solved during this work. Interesting solutions for problems like the OCL method `allInstances()` are provided. Thus, the tool was used as a central background for this research.

Chapter 5

Design and Fragment Generation

This chapter presents the architecture of the developed code generator. It describes which plug-ins of *DOT4Eclipse* are referred and which interfaces are introduced. The *ExpressionSwitch* class which is used to iterate over the data structure of the pivot model and its constraints is also presented. Finally, solutions for the type mapping to Java and for the fragment code generation from OCL constraints to Java are shown.

5.1 The Architecture

This section introduces briefly the architecture of the new code generator. The package structure and the class structure will be presented. The class *ExpressionSwitch* will be introduced.

5.1.1 The Package Structure

The Java code generator is realized as a new plug-in of *DOT4Eclipse* which depends on different other plug-ins of the toolkit. Like the OCL parser and the OCL interpreter introduced in section 2.3, the code generator directly depends on three different plug-ins. These three plug-ins are the *Pivot Model*, the *Essential OCL* plug-in and the *Model Bus* plug-in. Figure 5.1 illustrates these dependencies.

5.1.2 The Class Structure

This section explains some classes and interfaces introduced for the Java code generator.

Some Internal Classes

Figure 5.2 shows two different interfaces which are internally used by the code generator. The first interface `ITransformedCode` describes fragments of code generated by the code generator. It provides methods to add or get the contained

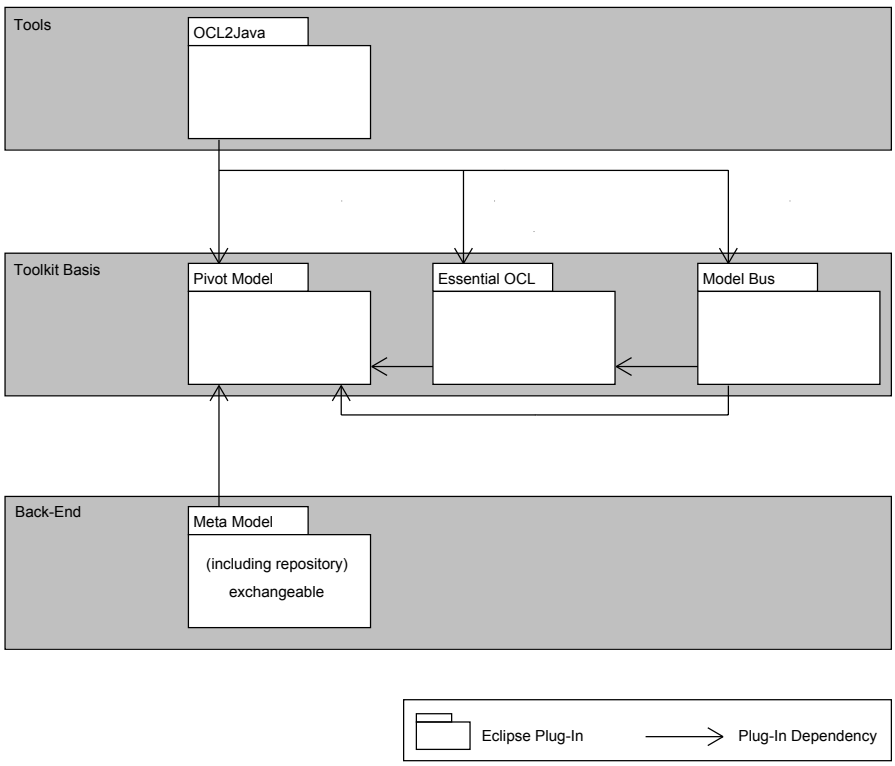

Figure 5.1: The plug-in dependencies of Ocl2Java.

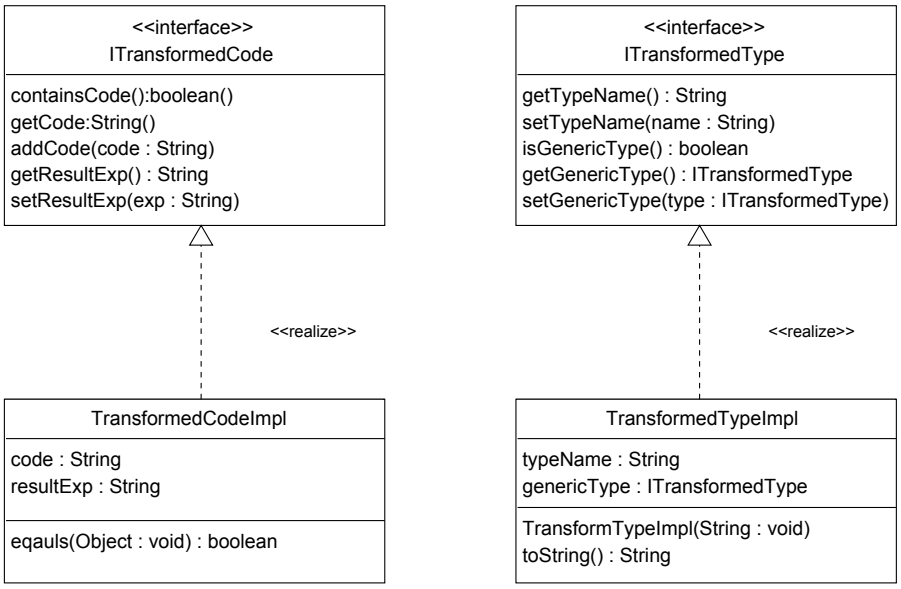

Figure 5.2: The interfaces ITransformedCode and ITransformedType and their realizations.

code and to set or get the result expression. The result expression describes a variable or expression containing the result of the contained code. The contained code for example could be a simple expression such as `int result = 2;`. Then, the result expression would be the variable expression `result`, because this variable contains the result of the whole code. The interface `ITransformedCode` is realized by the class `TransformedCodeImpl` which additionally provides the method `equals(Object)` to compare to different pieces of transformed code.

The second interface called `ITransformedType` describes already transformed types during code generation. It provides methods to set and get its type's name and eventually to get or set a generic type as well. The realization `TransformedTypeImpl` additionally provides the method `toString()` which returns a string containing the name of the type (eventually including the name of the generic type).

Both interfaces are used internally during code generation, but are furthermore visible for other plug-ins, because the interface `ITransformedCode` is also used to set *violation macros* for the code generation.

Public Interfaces

The code generator plug-in defines two public interfaces which are visible for other plug-ins. These two interfaces are shown in figure 5.3. The first interface called `IOcl2Code` represents code generators and defines three methods. The method `getSettings()` returns an instance of the second interface `IOcl2CodeSettings` which provides a lot of methods to configure the associated `IOcl2Code` instance. The two other methods of `IOcl2Code` can be used to

transform instrumentation code or fragment code for a given list of constraints.

Currently, only one class implements the interface `IOcl2Code`, which is the `Ocl2Java` class realizing the Java code generator. More implementations are possible to implement other code generators, for example a code generator which generates *C++* and *AspectC++* code instead of Java code. The interface could be used by an adaption of the declarative SQL code generator *OCL22SQL* [Hei06] as well, but such a code generator would not need the provided method `transformInstrumentationCode()`.

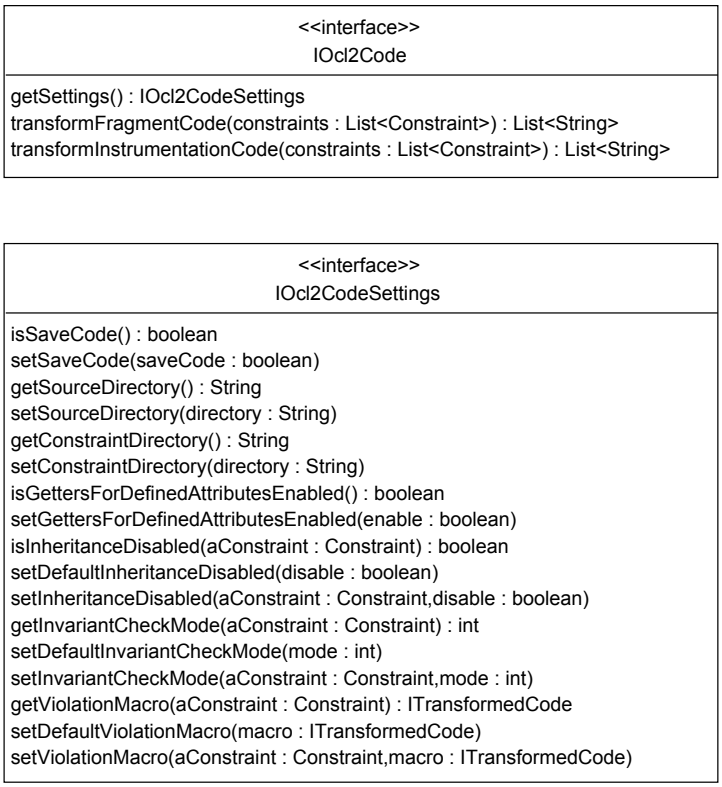

Figure 5.3: The interfaces IOcl2Code and IOcl2CodeSettings.

The Expression Switch Pattern

The code generator uses the class `ExpressionSwitch` which is provided by the *Essential OCL* plug-in as shown in figure 5.4. The class `ExpressionSwitch` implements the *Switch Class Pattern* which can be generated for *EMF* models and can be used to iterate over the *abstract syntax model (ASM)* of these models [BSM+03, p. 207ff]. "The switch class implements a `switch`-like callback mechanism that is used for dispatching based on a model object's type" [BSM+03, p. 207]. The class `ExpressionSwitch` already generated by Matthias Bräuer provides such a switch mechanism for *Essential OCL* models.

During the development of the OCL interpreter of *DOT4Eclipse* Ronny Brandt compared the *Switch Pattern* with the commonly known *Visitor Pattern*

Figure 5.4: The class ExpressionSwitch.

[GHJV95, p. 331ff] and pointed out that the *Switch Pattern* provides a good solution to iterate over an ASM of an *Essential OCL* model [Bra07, p. 25ff]. The key benefit of the *Switch Pattern* is the fact that only one class in the class structure has to be adapted if the meta model has been changed. The class which has to be adapted is the switch class itself.

The same criteria which lead Ronny Brandt to use the *SwitchPattern* for the implementation of his OCL interpreter were used to decide that the *SwitchPattern* would provide a good solution for the iteration over the *Essential OCL* models to generate Java code as well.

The mechanism of the *SwitchPattern* is very simple. In *DOT4Eclipse* the class *ExpressionSwitch* provides the method `doSwith(EObject)` which can be called to iterate over *Essential OCL* models. The `doSwitch(EObject)` method checks, which type of `EObject` was given as parameter first and then calls the depending `case()` method. "[... E]ach case walks up the inheritance hierarchy from the actual type of the object to `EObject`, calling out to a specific `case()` handler method for each class. It stops when one of these methods return a non-`null` value, which `doSwitch()` then returns" [BSM+03, p. 208].

The advantage of this pattern is the fact that the developer does not have to think about the tree structure of his ASM. He only has to implement all methods of the *SwitchPattern* which handle cases of model elements that are concrete and not abstract. For example the developer has to implement the methods `caseIntegerLiteralExp()` and `caseStringLiteralExp()` but not the method `caseLiteralExp()`.

The class `ExpressionSwitch` is an abstract class which can be parameterized with a type that represents the result type of all `case()` methods. In the depending `case()` methods the developer implements the code which will be executed when such an element has been reached during iteration. During code execution the `doSwitch(EObject)` can be called recursively to react on sub elements of the currently handled element of the ASM. The Java code generator parameterizes the pattern with the type `ITransformCode`. Thus, any `case()` method returns a piece of transformed code.

The class `ExpressionSwitch` provides the central iteration mechanism of the new code generator. The `case()` methods contain the code which generates the corresponding Java code to the given *Essential OCL* expressions. The `case()` methods use *StringTemplates*, parameterize them and use them to generate the Java code.

The Ocl2Java Class

Figure 5.5 shows the implementation of the interface `IOcl2Code` called `Ocl2Java` which represents the central class of the developed code generator (Please note that not all methods of the classes `Ocl2Code` and `ExpressionSwitch` are shown in the diagram). The code generator extends the class `ExpressionSwitch` to iterate over the constraints and their expressions to generate code.

To generate code fragments the method `transformFragmentCode(List<Constraint>)` is called which internally calls the private method `transformFragmentCode(Constraint)` for any constraint of the given list. This method invokes the method `doSwitch(Eobject)` for the expression of the given constraint. `doSwitch(Eobject)` is the method which starts the iteration provided by the `ExpressionsSwitch` class.

For any type of expression the depending `case()` method is implemented to generate the code for the expression. In addition to methods to handle different expressions, the class `Ocl2Java` also provides methods to transform types of the pivot model into Java types. The class `Ocl2Java` possesses an environment called `IOcl2CodeEnvironment` which provides methods to generate variable names or to store some values during code generation such as referred attributes or variables on which the special OCL operation `@pre` is invoked.

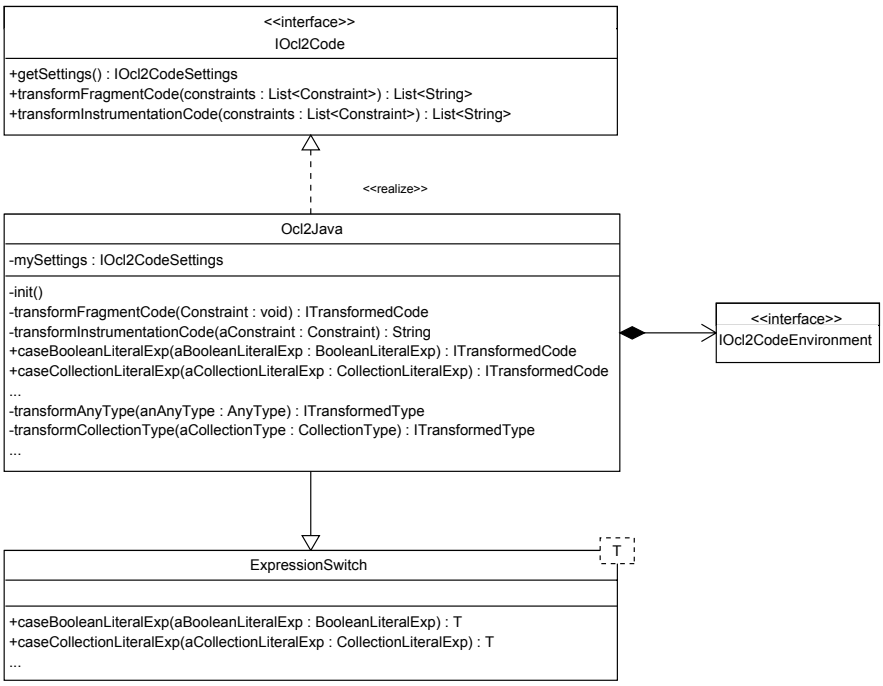

Figure 5.5: The code generator Ocl2Java.

5.2 Type Mapping

As already mentioned in section 4.1.1, the new code generator does not use a standard library to transform OCL types into Java types. The types are transformed into already existing Java types instead (if this is possible). The type mapping is realized in the code generator via simple *String Templates* which results in the name of the mapped type in Java. The type mapping from OCL to Java types is explained briefly in the following. An overview of all mapped OCL types can be found in appendix A.

5.2.1 Primitive Types

OCL contains the primitive types `Integer`, `Real`, `UnlimitedNatural`, `Boolean` and `String`. Warmer and Kleppe propose to map these types to the primitive types in Java as well [WK04, p. 97]. `Integer` becomes `int`, `Real` becomes `float` and `Boolean` becomes `boolean`.

Type in OCL	Type in Java
`Boolean`	`java.lang.Boolean`
`Integer`	`java.lang.Integer`
`Real`	`java.lang.Float`
`String`	`java.lang.String`
`UnlimitedNatural`	`java.lang.Long`

Table 5.1: Type mapping from OCL to Java for primitive types.

This solution causes trouble because primitive Types in Java can not be used as generic types, for example in the declaration of collections. The automatic substitution from primitive to wrapper types in the Java compiler (called *autoboxing* [Ull06, Section 8.2.5]) solves this problem in many situations, but the initialization of collections with primitive generic types is not not possible in Java.

Furthermore, primitive types also cause problems when all their instances for the special OCL operations `allInstances()` and `oclIsNew()` shall be collected. In addition to that the `instanceof` operator in Java needed for the OCL operations `oclIsKindOf()` and `oclIsTypeOf()` is not available for primitive Java types. Finally, new defined methods returning a primitive type can not be implemented by default using the Java statement `return null;`

The *OCL2J* approach (introduced briefly in section 4.2) solves this problem by standardly mapping to primitive types and converting these primitive types into wrapper types in any situation were this is compulsory [BDL04, p. 32ff]. The newly developed code generator avoids this technique and thus supports the possibility to exchange any type of the type mapping (e. g. if the code generator will be adapted to another target language). It only uses wrapper types in Java. Table 5.1 shows all primitive OCL types and their corresponding wrapper types in Java.

This solution solves the initialization problem for generic collections but causes some new problems as well: the implementation of already defined methods in the model using `body` constraints could be error-prone if the originally defined method returns a primitive type in Java. For example a method `getAge():int` could cause an error if the generated code tries to return an `Integer` instance. The same problem could occur if the generated code tries to address join points of a Java method by using a wrapper type instead of a primitive type as return type. The second problem is solved by using wild cards in the pointcut declaration but the first problem is ignored by the developed code generator and should be addressed and finally solved in future works.

5.2.2 Enumerations

OCL provides the possibility to use Enumeration types. Enumeration types exist in Java as well. Thus, enumerations are mapped to the corresponding Java type as shown in table 5.2.

Type in OCL	Type in Java
`Enumeration<T>`	`java.lang.Enum<T>`

Table 5.2: Type mapping from OCL to Java for enumerations.

Type in OCL	Type in Java
`TupleType(<attribute1>: <type1>, ...)`	`java.util.HashMap<String, Object>`

Table 5.3: Type mapping from OCL to Java for tuples.

5.2.3 Tuples

OCL provides a `Tuple` type with which the user can define a collection of different values (possibly of different types) identified by an attribute name inside the tuple [OMG06, p. 35]. Java does not support tuple types. Thus, tuples are transformed into a map containing the name of the attributes as a key of the Java type `java.lang.String` and a value of any type (sub-classes of `java.lang.Object`) such as proposed in [WK04, p. 100]. The mapping for tuple types is shown in table 5.3.

A map can be used for tuples in Java because OCL does not define any operations over tuples which are not provided be the `java.util.Map` interface. The access to tuple attributes can be simply transformed into a value call like `aTuple.get("attributeName")`.

5.2.4 Collection Types

OCL defines four different collection types: `Bag`, `OrderedSet`, `Sequence`, and `Set` [OMG06, p. 144ff]. Some of them such as `Bag` can be directly mapped to Java classes or interfaces like `java.util.List`. But the OCL specification defines many operations over its collection types which are not present for collection types in Java. Thus, the four collection types are the only types for which the code generator introduces new Java types.

A new plug-in of the *DOT4Eclipse* called `tudresden.ocl20.pivot.ocl2-java.types` has been created. This plug-in contains implementations for the four collection types and a general abstract collection class. Any code generated by the new code generator has to import these type package in order to be able to work with collections defined in OCL. The type mapping for the OCL collections is shown in table 5.4.

5.2.5 Special OCL types

OCL introduces some other special types called `OclAny`, `OclType`, `OclUndefined`, `OclVoid` and `OclInvalid`. `OclAny` can be mapped to `java.lang.Object` and `OclType` to `java.lang.Class`. `OclUndefined` and `OclVoid` are not mapped to any specific type in Java, because undefined literal expressions are mapped to `null` in Java and void literal expressions are mapped to an empty piece of code in Java because they are used as generic type of collections whose generic type has not been set. The type `OclInvalid` has not to be mapped because `OclInvalid` is the result of all invalid constraints which are already blocked by

Type in OCL	Type in Java
`Bag<T>`	`tudresden.ocl20.pivot.ocl2java.types.OclBag<T>`
`OrderedSet<T>`	`tudresden.ocl20.pivot.ocl2java.types.OclOrderedSet<T>`
`Sequence<T>`	`tudresden.ocl20.pivot.ocl2java.types.OclSequence<T>`
`Set<T>`	`tudresden.ocl20.pivot.ocl2java.types.OclSet<T>`

Table 5.4: Type mapping from OCL to Java for collections.

Type in OCL	Type in Java
`OclAny`	`java.lang.Object`
`OclInvalid`	*no mapping provided*
`OclType`	`java.lang.Class`
`OclUndefined`	*no type just* `'null'`
`OclVoid`	*an empty piece of code*

Table 5.5: Type mapping from OCL to Java for special OCL types.

the parser before code can be generated for them. The type mapping for the special OCL types is presented in table 5.5.

The special type `OclAny` supports some operations such as `allInstances()`, `oclIsNew()` or `OclAsType()`. Some of these methods are directly transformed into corresponding operations or operators in Java such as `instanceof` explained in section 5.3.2. Others are realized during fragment instrumentation and are explained in sections 6.5 and 6.7.

5.3 Fragment Generation

This section explains briefly how the literals of the given OCL expressions are transformed into Java code. For all code transformations, the code generator uses *StringTemplates* to generate the Java code inside the corresponding `case()` method of the *ExpressionSwitch* class. The *StringTemplates* used can be found in appendix C.

This section will not explain all different expressions possible in OCL. The transformation for primitive types, enumerations, if expressions and some special OCL types such as `OclAny` are not described. Some details about the transformation of the type expressions can be found in section 5.2.

The transformation of property call expressions, operation call expressions, collection literal expressions and iterator expressions will be explained in the following.

5.3.1 Property Call Expressions

"A PropertyCallExpression is a reference to an Attribute of a Classifier defined in a[n] UML model. It evaluates to the value of the attribute" [OMG06, p. 44]. Generally, a property call expression can simply be transformed from OCL to

```
1 Tuple {name: String = 'John', age: Integer = 10}.name
```

Listing 5.1: OCL code for a property call expression on a tuple type.

```
1 java.util.HashMap<String, Object> tuple1;
2 tuple1 = new java.util.HashMap<String, Object>();
3
4 tuple1.put(name, "John");
5 tuple1.put(age, new Integer(10));
6
7 tuple1.get("name");
```

Listing 5.2: Generated Java code for a property call expression on a tuple type.

Java. For example the OCL expression `Person.age` will be transformed to the Java code `Person.age` which is exactly the same.

An interesting case occurs, when a property call expression will be transformed which references to a tuple attribute which is not implemented as a tuple type in Java (see also section 5.2.3). For example the OCL expression shown in listing 5.1 leads to such a property call expression. The transformed Java code for this expression is presented in listing 5.2. As shown in the listing, the general property call has to be transformed into a method call on the map which represents the tuple in Java.

5.3.2 Operation Call Expressions

"An OperationCallExp refers to an operation defined in a Classifier. The expression may contain a list of argument expressions if the operation is defined to have parameters" [OMG06, p. 45]. Generally, such an operation call expression can be simply transformed from OCL to Java. The OCL expression `Person.getAge()` for example will be transformed to the Java code `Person.getAge()` which is exactly the same expression. All operation mappings from OCL to Java used by the new code generator are listed in appendix B. In the following some special cases are discussed briefly.

Operations Delegated to other Types

Some operations which are provided in OCL must be referenced to other types or utility classes in Java. An example is the operation `abs()` which is available for numeric types in OCL. In Java this operation is delegated to the utility class `java.lang.Math` (see listing 5.3).

Equality on Primitive Types

The decision to transform all primitive OCL types to wrapper type in Java (see section 5.2.1), causes a problem invoking the OCL operation `=` on primitive types in Java. Primitive types in Java are normally compared using the operator `==`. But using the operator `==` does not work on wrapper types in Java. `new Integer(0) == new Integer(0)` results in `false` because the operator `==` checks if both `Integers` are the same instance. The operation `Object.equals()`

```
1 // Java code for aNumeric.abs().
2 result = java.lang.Math.abs(aNumeric);
```

Listing 5.3: Java code for the OCL operation abs().

```
1 // Java code for aCollection->size() = 1.
2 ((Object) aCollection.size()).equals(new Integer(1));
```

Listing 5.4: Java code for the equality operation on primitive types.

```
1 // Java code for anInteger / anotherInteger.
2 result = ((Float) anInteger / (Float) anotherInteger);
```

Listing 5.5: Java code for the division of two Integers.

```
1 // Java code for sum on Collection<GenericType>.
2 GenericType result;
3 result = new GenericType(0);
4
5 /* Compute the result of a sum() operation. */
6 for (GenericType anElement : aCollection) {
7     result += anElement;
8 }
```

Listing 5.6: Java code for the OCL operation sum().

```
1 // Java code for anObject.allInstances().
2 result = (new OclSet<aType>((java.util.Set<aType>)
3     this.allInstances.get(anObject.getClass()
4         .getCanonicalName()).keySet()));
```

Listing 5.7: Java code for the OCL operation allInstances().

```
1 // Java code for anObject.oclAsType().
2 result = ((aType) anObject);
3
4 // Java code for anObject.oclIsKindOf().
5 result = (anObject instanceof aType);
6
7 // Java code for anObject.oclIsOfType().
8 result = (anObject.getClass().getCanonicalName()
9   .equals("Cannonical name of aType");
```

Listing 5.8: Java solutions for oclAsType(), oclIsKindOf() and oclIsOfType().

must be used instead to compare wrapper types. This operation works but some operations such as `Collection.size()` result in a primitive type in Java such as `int`. Invoking `equals()` on a primitive type in Java is not possible. Thus the code generator generates a special code for the `=` operation on primitive OCL types. The left hand expression of such an expression in OCL is cast to `Object` in Java and then the operation `equals()` is invoked. Such a solution works in Java because of the *autoboxing* mechanism [Ull06, Section 8.2.5]. The presented solution is shown in listing 5.4. The same solution is used for the OCL operation `<>`.

Division on Integer Types

Another special case for the `Integer` type in OCL is the division operand. In OCL the result of a regular division on `Integer` types is a value of the type `Real`. In Java, the normal division results in an `Integer` type. Thus, a cast must be performed before the given integer literal expression is divided (see listing 5.5).

Sum on Collection Types

A special problem occurs when the OCL operation `sum()` shall be implemented. The `sum()` operation is provided for all collection types and results in the sum of all elements contained in the collection. Such a result can only be computed if the elements of the collection are of numeric types [OMG06, p. 146]. Such an operation normally could be implemented in the new implemented collection types.

However, such an implementation of a collection which has a generic element type can not be realized in Java. The problem occurs when the collection does not contain any element. The operation `sum()` has to decide whether it has to return `null` or `0` depending on the question whether or not the collection has a numeric generic type. But the generic type of a collection is not bound when the collection is initialized but when the collection gets its first element. Thus this decision can not be done for an empty collection. Runtime errors can occur for example when a collection with the generic type `String` does not contain any element. The operation `sum()` which results it the collection's generic type (which is `String`) returns an `Integer` because the collection does not know that its generic type is not a numeric type.

Thus the operation `sum()` is not implemented in the Java collection classes but is generated during code generation as a `for()` loop. Listing 5.6 shows such an implementation in Java. The advantage of this solution is that the OCL parser always knows the result type of the `sum()` operation and thus the code can be generated for empty collections also.

Operations on the Type OclAny

The special OCL type `OclAny` supports some special operations, which are available for all types in OCL. The operation `allInstances()` returns all instances of the type the operation has been invoked on. Such an operation is not available in Java and can only be implemented by using advanced programming techniques such as *aspect-oriented programming*. Section 6.7 explains how this

```
1 Sequence { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
2
3 Sequence { 1..10 }
```

Listing 5.9: Two ways to initialize a collection in OCL.

```
1 OclSequence<Integer> collection;
2 collection = new OclSequence<Integer>();
3
4 /* TODO: Auto-generated initialization does
5    only work for numeric values. */
6 for (Integer index = new Integer(1);
7   index <= new Integer(10); index++) {
8     collection.add(index1);
9 }
```

Listing 5.10: Java code for a collection with range initialization.

method is implemented during fragment instrumentation. Listing 5.7 shows a call on the map which realizes the storage of all instances in the generated code (see section 5.2.4).

Other operations of `oclAny()` are `oclAsType()`, `oclIsKindOf()` and `oclIsTypeOf()`. Their transformations are shown in listing 5.8.

5.3.3 Collection Literal Expressions

The transformation of collection literal expressions from OCL to Java is very simple. The only difference between OCL and Java is the initialization of the collections. In OCL two different possibilities are provided to initialize collections [OMG06, p. 177]:

1. Any item can be described by its own CollectionLiteralPart,
2. A set of items can be described by a CollectionLiteralRange.

Both possibilities are shown in listing 5.9. If a collection is initialized in OCL using a range expression the resulting Java code must contain a loop which iterates over the given range and adds an element to the collection for every element in the range. Such a Java implementation is shown in listing 5.10.

Please note that this construct only works for a collection which has an `Integer` element type. The OCL parser of *DOT4Eclipse* only accepts range expressions if they are defined over `Integer` values. Thus, the code generator does not have to consider the collection's element type.

5.3.4 Iterator Expressions

OCL defines a lot of operations over collections, which compute a result by iterating over all elements of the collections [OMG06, p. 25]. Such operations are called *iterators* and are represented by iterator expressions in the OCL meta model. All these iterator operations can be implemented in Java by defining loops over the collection. Such a Java implementation is shown in listing 5.11

```
1  Boolean result;
2  result = true;
3
4  /* Iterator: aCollection->forAll(Object anElement :
       aBooleanExpression) */
5  for (Object anElement : aCollection) {
6
7    if (!aBooleanExpression) {
8      result = false;
9      break;
10   }
11   // no else
12 }
```

Listing 5.11: Java code for the iterator forAll.

for the special iterator `forAll` which evaluates a given Boolean expression for all elements of the collection [OMG06, p. 27].

All other iterators use similar Java code to iterate over their collection and to compute their result. In OCL, however, iterators can also be used to iterate over more than one element: an iterator can iterate over the same collection twice to compare pairs of elements in a collection. Java does not support loops over more than one collection item. But loops can be nested in Java. Warmer and Kleppe pointed out that any iterator expression which uses more than one iterator variable can be transformed into a nested iterator [WK04, p. 179]. Thus, the transformed Java code for an iterator with more than one iterator variable results in nested `for()` loops in Java.

Another special case is the OCL iterator `sortedBy`, which iterates over a given collection and compares its elements by a given comparator expression. Let us assume we have a class `ClassX` with an attribute `attributeY` and want to iterate through a collection of `ClassX`s by comparing their `attributeY`s. Listing 5.12 shows such an OCL expression.

Java provides the utility class `Comparator` do define a compare strategy over a given class. Using `Comparator`s, the operation `java.util.Collections.sort (aCollection, aComparator)` can be used to sort collections. The code generator uses these utility classes in the generated Java code for `sortedBy` iterators (see listing 5.13).

```
1 inv: aCollection->sortedBy(attributeY)
```

Listing 5.12: A 'sortedBy' iterator in OCL.

```
1  ClassX result;
2  java.util.Comparator<ClassX> comparator;
3
4  comparator = new java.util.Comparator<ClassX>() {
5
6    /** Method which compares two
7        elements of the collection. */
8    public int compare(ClassX elem1, ClassX elem2) {
9
10     int compareResult;
11     compareResult = 0;
12
13     if (elem1.attributeY < elem2.attributeY) {
14       compareResult = -1;
15     }
16
17     else if (elem1.attributeY > elem2.attributeY) {
18       compareResult = 1;
19     }
20
21     return compareResult;
22   }
23 };
24
25 result = java.util.Collections
26   .sort(aCollection, comparator);
```

Listing 5.13: Java code for a 'sortedBy' iterator.

Chapter 6

Fragment Instrumentation

For the instrumentation of the transformed code fragments into the Java classes I decided to use the aspect-oriented language *AspectJ*. A short introduction in *aspect-oriented programming* can be found in section 3.1.

This chapter will present the instrumentation of all constraint types realized by the developed code generator. The solutions for some special OCL operations and attributes which can not be realized without code instrumentation are also presented. All shown examples use the *royal and loyal example* and its constraints which were published in [WK04]. The *royal and loyal example* can be found in appendix D.

6.1 Initial and Derived Values

Initial and derived values of attributes and association ends can be defined using the OCL expressions `init` and `derive` [OMG06, p. 9].

6.1.1 Initial Values with Init

By using the OCL expression `init` it is possible to define initial values for attributes which have already been defined by another OCL expression or an UML model [OMG06, p. 164]. Listing 6.1 shows a simple `init` expression which defines the initial value of the attribute `points` for the class `LoyaltyAccount` [WK04, p. 43]. Thus, the attribute `points` will be initialized with `0` during the creation of any `LoyaltyAccount` instance.

The easiest solution to instrument this `init` expression is to define an aspect containing a pointcut which describes all constructors of the class `LoyaltyAccount` and an advice which is executed after the defined pointcut and which initializes the attribute `points`. Such an aspect is described in listing 6.2.

Please note that the given aspect uses the modifier `privileged` to enable the access to eventually referenced attributes of the class `LoyaltyAccount` which are not visible (attributes defined using the modifiers `private` or `protected`).

The presented solution also works with inheritance relationships. Eventually other `init` expressions for a sub-class of `LoyaltyAccount` can override the `init` expression of `LoyaltyAccount` because their aspect code is executed after the execution of the aspect code for the class `LoyaltyAccount`.

```
1 context LoyaltyAccount::points
2 init: 0
```

Listing 6.1: A simple init constraint.

```
1  public privileged aspect InitAspect {
2
3      protected pointcut
4          allConstructors(LoyaltyAccount aClass):
5          execution(public LoyaltyAccount.new())
6          && this(aClass);
7
8      after(LoyaltyAccount aClass) :
9          allConstructors(aClass) {
10         aClass.points = 0;
11     }
12 }
```

Listing 6.2: An aspect instrumenting a simple init expression.

6.1.2 Derived Values with Derive

Using the OCL expression `derive`, derived values for attributes or association ends can be defined. Derived values are derived from other attributes and/or association ends of the constrained class [OMG06, p. 165]. Listing 6.3 shows a simple `derive` expression which derives the attribute `printedName` of the class `CustomerCard` from the association `owner` [WK04, p. 43].

My first idea to compute derived values was to define setter pointcuts for all attributes and association ends from which the derived value is derived using the *ApsectJ* expression `set`. However, this solution is error-prone. Attributes or association ends can be implemented by collections for which the pointcut `set` is only affected if the collection is redefined and not if any element is added or removed from the collection.

I therefore decided to derive the value when the derived value is read using the *ApsectJ* expression `get`. Listing 6.4 shows such an aspect for the constraint given in listing 6.3. The pointcut `printedNameGetter(CustomerCard aClass` is affected at any time when the attribute `printedName` of the class `CustomerCard` is read and the advice `before(CustomerCard aClass)` always executes the code to derive the attribute before the `printedName` attribute is read.

As the solution for `init` expressions, this solution also works with inheritance. The solution could possibly be improved if the derived attribute would only be computed if the value of the dependent attributes and associations ends did really change.

6.2 Method Implementation with Body

Using the OCL expression `body`, the bodies of methods can be defined in OCL [OMG06, p. 165]. Listing 6.5 shows a simple `body` expression which defines an implementation of the method `LoyaltyProgram.getName()`.

```
1 context CustomerCard::printedName
2 derive: owner.title.concat(' ').concat(owner.name)
```

Listing 6.3: A simple derive constraint.

```
1  public privileged aspect DeriveAspect {
2
3      protected pointcut
4          printedNameGetter(CustomerCard aClass) :
5            get(String printedName) && this(aClass);
6
7      before(CustomerCard aClass):
8          printedNameGetter(aClass) {
9            aClass.printedName =
10             aClass.owner.title.concat(" ")
11                 .concat(aClass.owner.name);
12     }
13 }
```

Listing 6.4: An aspect instrumenting a simple derive expression.

```
1 context LoyaltyProgram::getName() : String
2 body: self.name
```

Listing 6.5: A simple body constraint.

```
1  public privileged aspect BodyAspect {
2
3      protected pointcut
4        getNameCaller(LoyaltyProgram aClass):
5          call(String LoyaltyProgram.getName())
6          && target(aClass);
7
8      String around(LoyaltyProgram aClass):
9        getNameCaller(aClass) {
10         return aClass.name;
11     }
12 }
```

Listing 6.6: An aspect instrumenting a simple body expression.

The instrumentation of a method body using *AspectJ* is very easy. Listing 6.6 shows such an instrumentation. The aspect defines a pointcut `getNameCaller(LoyaltyProgram aClass)` which describes all calls of the method `LoyaltyProgram.getName()` which will be implemented as well as an advice which provides the code which will be executed to realize the methods using the advice type `around`.

This solution does also work with inheritance. But the code assumes that the method which will be implemented must have already been defined. Otherwise the advice defined by the aspect will not be affected and executed and thus the method will not be implemented. The solution for `def` constraints in the following will show a solution which works for not yet defined methods.

6.3 Attribute and Method Definition with Def

The OCL expression `def` can be used to define new methods, attributes or association ends for an already defined class [OMG06, p. 161]. Listing 6.7 shows a simple `def` expression which defines the new attribute `turnover` for the class `LoyaltyAccount` [WK04, p. 45]. The defined attribute has the type `Real` and derives from the collection operation `sum()` of the association `transactions`.

The derivation of the attribute value does work exactly like the derivation of attributes defined by `init` expressions which were mentioned above in 6.1.2. The interesting question is how to define the new attribute into the already existing class such as `LoyaltAccount`?

Unfortunately *AspectJ* does not provide any mechanism to define new attributes or methods of an affected class. New attributes and methods can be defined inside an aspect, but these attributes and methods are only visible for advices of aspect files. However, the new defined properties should also be available for other Java classes as well.

The problem is solved by using the *AspectJ* keyword `declare parents` which can be used to declare a new super class for a class. For example a given class `BaseClass` which extends the class `SuperClass` can be extended by using the *AspectJ* statement `declare parents : BaseClass extends ExtendedClass;`

The changed inheritance relationship between `BaseClass` and `SuperClass` is shown in figure 6.1. I call this solution the *ExtendedClass Pattern*.

Using the *ExtendedClass Pattern*, new attributes and methods can be defined in the new super class of the constrained class. For the constraint class `LoyaltyAccount` in listing 6.7 we can define a new super class `ExtendedLoyaltyAccount` containing the newly defined attribute `turnover`. An aspect using this solution is shown in listing 6.8.

As mentioned above, the derivation of the attribute value uses the same mechanism as the instrumentation of `derive` expressions 6.1.2.

`Def` expressions can also be used to define new methods. The instrumentation code for newly defined methods uses the *ExtendedClass Pattern* as well. The derivation of the method's return value uses the same mechanisms as the instrumentation of `body` expressions explained in section 6.2.

The instrumentation of `def` expressions does also work with inheritance.

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

Listing 6.7: A simple constraint for an attribute definition.

```
1  public privileged aspect DefAspect {
2
3      declare parents : LoyaltyAccount
4        extends ExtendedLoyaltyAccount;
5
6      protected pointcut
7        turnoverGetter(LoyaltyAccount aClass) :
8          get(float turnover) && this(aClass);
9
10     before(LoyaltyAccount aClass):
11         turnoverGetter(aClass) {
12
13         OclBag<Float> result1;
14         result1 = new OclBag<Float>();
15
16         /* Iterate and collect elements. */
17         for (Transaction anElement1 :
18             aClass.transactions) {
19             result1.add(anElement1.amount);
20         }
21
22         float result2;
23         result2 = 0;
24
25         /* Compute the result of sum(). */
26         for (float anElement2 : result1) {
27             result2 += anElement2;
28         }
29
30         aClass.turnover = result2;
31     }
32 }
```

Listing 6.8: An aspect instrumenting a simple def expression for an attribute.

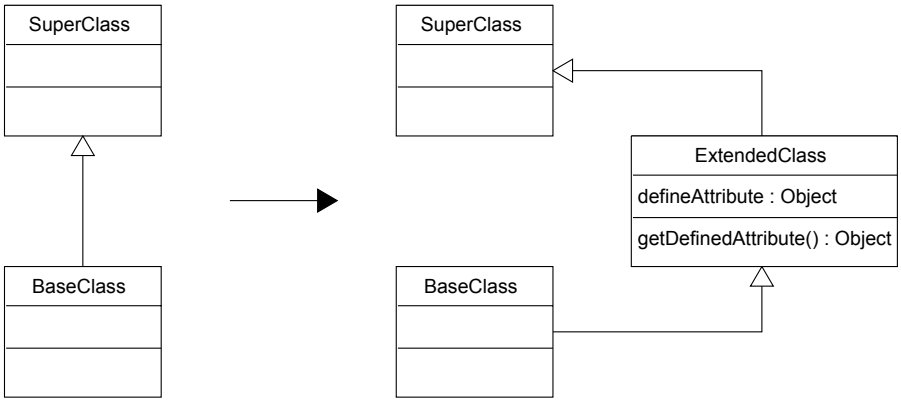

Figure 6.1: The ExtendedClass pattern.

6.4 Preconditions

Preconditions define conditions which must be true before the execution of a method specified in the context of a constraint [OMG06, p. 8].

Listing 6.9 shows a simple precondition over the method `enroll(Customer c)` for the class `LoyaltyProgramm` [WK04, p. 62]. The precondition declares that for any `Customer c`, who will be enrolled into a `LoyaltyProgram`, its name does not have to be empty.

Listing 6.10 shows an instrumentation of this precondition using *AspectJ*. The aspect defines a pointcut `enrollCaller(..)` which describes all calls of the method `enroll(Customer)` which will be constrained and an advice which provides the code which will be executed before the execution of the method `enroll(Customer)` using the advice type `before`.

Normally, this solution also works with inheritance. However, the situation gets more complicated when sub-classes of the constrained class are constrained by own preconditions. According to *Liskov's substitution principle*, preconditions can be weakened by preconditions defined in sub-classes [WK04, p. 145]. Implementing Liskov's substitution principle would be very complicate because the code generator would have to check for every precondition to control if another precondition defined for any sub-class of the constrained class would be weaker.

Thus the new code generator ignores Liskov's substitution principle but provides the possibility to decide whether or not a precondition should be enforced for all sub-classes of the constrained class as well. If the user decides not to enforce the constraint for sub-classes, the generated code will be extended by a check over the canonical name (a combination of all package names and the class name according to the Java Language Specification [GJSB05, p. 145f]) of the constrained class (see listing 6.11).

```
1 context LoyaltyProgram::enroll(c : Customer)
2 pre: c.name <> ''
```

Listing 6.9: A simple precondition.

```
1  public privileged aspect PreAspect {
2
3      protected pointcut
4        enrollCaller(LoyaltyProgram aClass, Customer c):
5          call(Boolean LoyaltyProgram.enroll(Customer))
6          && target(aClass) && args(c);
7
8      before(LoyaltyProgram aClass, Customer c):
9          enrollCaller(aClass, c) {
10         if (!!c.name.equals("")) {
11                 throw new RuntimeException(
12                   "Error: Constraint was violated.");
13         }
14         // no else.
15     }
16 }
```

Listing 6.10: An aspect instrumenting a simple precondition.

```
1  before(LoyaltyProgram aClass, Customer c): enrollCaller(
       aClass, c) {
2      if(aClass.getClass().getCanonicalName()
3          .equals("LoyaltyProgram") {
4
5          if (!!c.name.equals("")) {
6              throw new RuntimeException(
7                      "Error: Constraint was violated.");
8          }
9          // no else.
10     }
11     // no else.
12 }
```

Listing 6.11: An advice for a precondition which is not enforced on subclasses.

```
1 context LoyaltyAccount::isEmpty():Boolean
2 post: result = (points = 0)
```

Listing 6.12: A simple postcondition.

```
1  public privileged aspect PostAspect {
2
3      protected pointcut isEmptyCaller(LoyaltyAccount
4          aClass):
5          call(Boolean LoyaltyAccount.isEmpty())
6          && target(aClass);
7
8      Boolean around(LoyaltyAccount aClass):
9          isEmptyCaller(aClass) {
10
11         Boolean result;
12         result = proceed(aClass);
13
14         if (!(result ==
15             (aClass.points == new Integer(0)))) {
16                 throw new RuntimeException(
17                   "Error: Constraint was violated.");
18         }
19         // no else.
20
21         return result;
22     }
23 }
```

Listing 6.13: An aspect instrumenting a simple postcondition.

6.5 Postconditions

Postconditions define conditions which must be true after the execution of a method defined in the context of a constraint [OMG06, p. 8].

Listing 6.12 shows a simple expression which defines a postcondition over the method `isEmpty()` for the class `LoyaltyAccount` [WK04, p. 53]. The postcondition declares that the result of the method `isEmpty()` always has the same result as the check if the `points` of the `LoyaltyAccount` are `0`.

Listing 6.13 shows an instrumentation of this precondition using *AspectJ*. The aspect defines a pointcut `isEmptyCaller(..)` which describes all calls of the method `isEmpty()` and an advice which provides the code which will be executed around the execution of the method using the *advice* type `around`.

At first, the result of the executed method `isEmpty()` is saved in a variable `result` of the same type as the result of the method. This variable has the same name as the special variable `result` in OCL which can be referred in any postcondition and which always contains the result of the execution of the method [OMG06, p. 15f]. Afterwards, the defined postcondition is verified. Finally, the result of method's execution is returned.

Similar to the solution for preconditions this solution does also work with inheritance. Regarding Liskov's substitution principle, preconditions can be strengthened by constraints defined over sub-classes of the constrained class [WK04, p. 145]. Nevertheless, the code generator provides the possibility to disable the enforcement of inheritance for postconditions as well. The technical solution is the same as for preconditions explained in section 6.4.

Besides `result` OCL provides more special properties and operations which can be used in postconditions to refer to values of the constrained class instance before the constrained method's execution. This properties and operations can be simply implemented using *AspectJ* and are discussed in the following.

6.5.1 The Special Property @pre

OCL defines the special property `@pre` which can be used to compare values in postconditions with their values before the constrained method's execution [OMG06, p. 23f]. Listing 6.14 shows such a postcondition. In the instrumentation aspects such values have to be stored if they are referred to the following constraint.

Listing 6.15 shows an *AspectJ* solution for such a constraint. Before storing the `result` value and the method execution, variables are defined for all `@pre` values which are referred in the following code. Then these variables are initialized. The initialization depends on the type of the `@pre` value.

Primitive types and collections are simply copied using the Java constructor `new`. Collections are simply copied as well because the code generator assumes that the user wants to compare the length of the Collection using `@pre` or similar checks like whether or not a specific element is contained into the collection.

However, the situation gets worse when the user uses `@pre` to refer to instances of classes which are not primitive types or collections. The code generator can not simply copy these objects because it does not know whether or not any attribute of the class instance has to be copied as well. This problem is similar to commonly known problems using the provided Java operation `clone()` [Ull06, Section 6.8.4].

```
1 context LoyaltyProgram::enroll(c : Customer)
2 post: participants = participants@pre->including(c)
```

Listing 6.14: A simple postcondition using @pre.

```
1  Boolean around(LoyaltyProgram aClass, Customer c):
2      enrollCaller(aClass, c) {
3
4      OclSet<Customer> atPreValue;
5      atPreValue1 = new OclSet<Customer>(aClass
6        .participants);
7
8      Boolean result;
9      result = proceed(aClass, c);
10
11     if (!aClass.participants.equals(atPreValue1
12         .including(c))) {
13         throw new RuntimeException(
14             "Error: Constraint was violated.");
15     }
16     // no else.
17
18     return result;
19 }
```

Listing 6.15: An aspect instrumenting a postcondition using @pre.

In this situation the code generator pushes the copy problem to the user. The code generator generates a method `createCopy()` which is called during the initialization of the `@pre` value which will be stored. This method has to be implemented by the user to provide a correct initialization of the stored value. The listings 6.16 and 6.17 show such an example in OCL and *AspectJ*. The generated method `createCopy()` contains a default implementation which returns the same object given as argument to the method. The method contains a `TODO` comment which informs the user that he has to implement this method. Such a comment will be highlighted in case tools such as Eclipse.

6.5.2 The Special Operation OclIsNew

OCL defines another special operation `oclIsNew()` for postconditions which can be used to check if an instance was created during the constrained method's execution or not [OMG06, p. 20f]. Listing 6.18 shows such a postcondition [WK04, p. 184].

Listing 6.19 shows an *AspectJ* solution for such a constraint which was invented during the development of the *OCL2J approach* [BDL04, p. 41ff]. The aspect creates a map which collects all instances of the types on which the operation `oclIsNew()` is called during the postcondition. An advice observes all constructors of these types and collects all instances of these types which are newly created. During the check of the postcondition the map is cleared first, then the constrained method is executed. After the execution of the method the

```
context LoyaltyProgram::enroll(c : Customer)
post: membership = membership@pre
```

Listing 6.16: A simple postcondition using @pre.

```
public privileged aspect PostAspect {

    protected Membership createCopy(Membership anObject) {

        Membership result;

        /* TODO: Auto-generated code to copy values */
        result = anObject;

        return result;
    }

    protected pointcut
      enrollCaller(LoyaltyProgram aClass, Customer c):
        call(Boolean LoyaltyProgram.enroll(Customer))
        && target(aClass) && args(c);

    Boolean around(LoyaltyProgram aClass, Customer c):
        enrollCaller(aClass, c) {
        Membership atPreValue1;
        atPreValue1 = this.createCopy(aClass.membership);

        Boolean result;
        result = proceed(aClass, c);

        if (!aClass.membership.equals(atPreValue1)) {
                throw new RuntimeException(
                  "Error: Constraint was violated.");
        }
        // no else.

        return result;
    }
}
```

Listing 6.17: An aspect instrumenting a postcondition using @pre.

```
1 context Transaction::getProgram(): LoyaltyProgram
2 post: not result.oclIsNew()
```

Listing 6.18: A simple postcondition using oclIsNew().

```
1  public privileged aspect PostAspect {
2
3      protected java.util.Map<Object, Object> newInstances =
4          new java.util.WeakHashMap<Object, Object>();
5
6      after(LoyaltyProgram aClass) :
7        execution(LoyaltyProgram.new(..))
8        && this(aClass) {
9          this.newInstances.put(aClass, null);
10     }
11
12     protected pointcut
13       getProgramCaller(Transaction aClass):
14         call(LoyaltyProgram Transaction.getProgram())
15         && target(aClass);
16
17     LoyaltyProgram around(Transaction aClass):
18       getProgramCaller(aClass) {
19         this.newInstances.clear();
20
21         LoyaltyProgram result;
22         result = proceed(aClass);
23
24         if (!!this.newInstances.containsKey(result)) {
25                 throw new RuntimeException(
26                   "Error: Constraint was violated.");
27         }
28         // no else.
29
30         return result;
31     }
32 }
```

Listing 6.19: An aspect instrumenting a postcondition using oclIsNew().

```
1 context Customer
2 inv ofAge: age >= 18
```

Listing 6.20: A simple invariant.

map only contains all instances which were created during the execution and the map can be used for calls on `oclIsNew()`.

An important question is why the code generator uses a map to store instances as keys and `null` values as value. The reason is that Java provides the special map type `WeakHashMap` which provides a map which only weakly store keys. Thus all instances which are collected in the key set of this map are removed by the garbage collector if no other reference to the instance exists anymore. This is very important to avoid long time collection of objects which are not needed anymore by any other class or object.

6.6 Invariants

Invariants define conditions which must be true at any time during the life cycle of all instances of a class defined in the context of the constraint [OMG06, p. 7]. Listing 6.20 shows a simple invariant over the class `Customer` [WK04, p. 46]. This invariant declares that any `Customer` must have an `age` which is greater than 18 or equal to 18 at any time.

As mentioned in section 4.1.3, the new code generator provides three different variants, when instrumented invariants will be verified during runtime:

1. Invariants can be checked after construction of an object and after any change of an attribute or association which is in scope of the invariant condition (*Strong Verification*).

2. Invariants can be checked after construction of an object and before or after the execution of any public method of the constrained class (*Weak Verification*).

3. And finally, invariants can only be checked if the user calls a special method at runtime (*Transactional Verification*).

The example shown in listing 6.20 is used to present the instrumentation code for all three verification variants. All of them use the same mechanism such as the instrumentation of pre- and postconditions to enable or disable the inheritance of the instrumented constraints.

6.6.1 Strong Verification

Listing 6.21 shows the instrumentation of the shown invariant which is verified after construction of an object and after any change of an attribute or association which is in scope of the invariant condition.

The aspect defines a pointcut `allConstructors(Customer)` which describes all constructors of the constrained class `Customer`. For every attribute which is in the scope of the constraint verification another pointcut which observes

```
1  public privileged aspect InvAspect {
2
3      protected pointcut allConstructors(Customer aClass):
4          execution(Customer.new(..)) && this(aClass);
5
6      protected pointcut ageSetter(Customer aClass) :
7          set(* Customer.age) && this(aClass);
8
9      protected pointcut allSetters(Customer aClass) :
10         ageSetter(aClass);
11
12     after(Customer aClass) : allConstructors(aClass)
13         || allSetters(aClass) {
14         if (!(aClass.age >= new Integer(18))) {
15                 throw new RuntimeException(
16                   "Error: Constraint was violated.");
17         }
18         // no else.
19     }
20 }
```

Listing 6.21: An aspect for an invariant with strong verification.

changes of this attribute is defined. The example shows that these are only the attribute `age` and its pointcut `ageSetter(Customer)`. Another pointcut called `allSetters(Customer)` collects the pointcuts for all affected attributes. Additionally, an advice which is executed after all the defined pointcuts and which checks the invariant is defined.

Tests show that this verification principle causes some trouble at runtime if an instrumented invariant observes more than one attribute for verification.

Let us assume that a `Customer` has an attribute `name` and another attribute `surname`. A defined invariant checks, whether or not the `name` and `surname` are empty. During the construction of a `Customer` instance first the `name` is set. After the setting of the `name` the aspect is executed and the invariant is checked. The `surname` has not been set yet and the verification fails. This problem has not been solved yet and will be ignored in this work.

Another problem is the observation of nested properties and associations. By now, the code generator only supports the pointcut generation for properties and associations which are directly linked to the constrained class. For other properties and associations the pointcut generation is difficult to realize because the properties and associations are linked via other associations which is difficult to describe in *AspectJ*. Future works could investigate this problem more specifically.

6.6.2 Weak Verification

Listing 6.22 shows the instrumentation of the invariant presented in listing 6.20 which is verified after construction of every `Customer` instance and before or after the execution of any public method of any `Customer` instance.

The aspect defines a pointcut `allConstructors(Customer)` which describes

```
1  public privileged aspect InvAspect {
2
3      protected pointcut allConstructors(Customer aClass):
4          execution(Customer.new(..)) && this(aClass);
5
6      protected pointcut allPublicMethods(Customer aClass):
7          execution(public * Customer.*(..))
8          && this(aClass);
9
10     before(Customer aClass) : allPublicMethods(aClass) {
11         if (!(aClass.age >= new Integer(18))) {
12                 throw new RuntimeException(
13                   "Error: Constraint was violated.");
14         }
15         // no else.
16     }
17
18     after(Customer aClass) : allConstructors(aClass)
19       || allPublicMethods(aClass) {
20         if (!(aClass.age >= new Integer(18))) {
21                 throw new RuntimeException(
22                   "Error: Constraint was violated.");
23         }
24         // no else.
25     }
26
27     after(Customer aClass) throwing :
28       allPublicMethods(aClass) {
29         if (!(aClass.age >= new Integer(18))) {
30                 throw new RuntimeException(
31                   "Error: Constraint was violated.");
32         }
33         // no else.
34     }
35 }
```

Listing 6.22: An aspect for an invariant with weak verification.

all constructors of the constrained class `Customer` and a pointcut `allPublicMethods(Customer)` for all public methods of the constrained class. Furthermore, three advices are defined. The first advice checks the invariant before the execution of any public method. The second advice checks the invariant after the execution of any constructor or public method. And the third advice checks the invariant after the execution of any public method which fails by throwing an exception. This is important because invariants must also be true if an executed method fails [BDL05, p. 3].

Tests show that this verification principle causes some troubles as well. Let us assume that a very complex invariant is violated during runtime after the execution of a public method and the *ViolationMacro* defined by the user throws an exception. The throwing of the exception activates the instrumentation aspect again, because the aspect also observes public methods which fail with

```
1  public privileged aspect InvAspect {
2
3      declare parents : Customer extends ExtendedCustomer;
4
5      protected pointcut checkInvariantsCaller(
6        Customer aClass):
7          call(void Customer.checkInvariants())
8          && target(aClass);
9
10     after(Customer aClass) :
11       checkInvariantsCaller(aClass) {
12         if (!(aClass.age >= new Integer(18))) {
13                 throw new RuntimeException(
14                 "Error: Constraint was violated.");
15         }
16         // no else.
17     }
18 }
```

Listing 6.23: An aspect for an invariant with transactional verification.

an exception. The complex invariant is verified again and some maintenance problems can occur.

This problem could only be avoided if the advice which reacts on thrown exceptions checks the type of exception and ignores any exception which was thrown by the *ViolationMacro*. This can only be realized by the user who defines the *ViolationMacro* and will be ignored during this work.

6.6.3 Transactional Verification

Listing 6.23 shows the instrumentation of the presented invariant which is verified after the invokation of the newly defined method `checkInvariants()` only.

At first, the aspect defines a new super class for the constraint class `Customer` using the *ExtendedClass Pattern* which was explained in section 6.3. The super class `ExtendedCustomer` defines the method `checkInvariants()` which can be called by the user to check the instrumented invariants. Then the aspect defines a pointcut `checkInvariantsCaller(Customer)` which observes the method `checkInvariants()` and an advice which is executed after any execution of this method.

This verification strategy would be very efficient if the user wants to check his defined invariants only at specific points during runtime.

6.7 The Special Operation AllInstances

OCL defines a special operation called `allInstances()` which returns a Set containing all instances of a model type [OMG06, p. 139]. Listing 6.24 shows a constraint which uses `allInstances()` [WK04, p. 184].

The presented solution to implement `allInstances()` is similar to the solution presented for `oclIsNew()` in section 6.5.2. Listing 6.25 shows an *AspectJ*

```
1 context Transaction
2 inv: self.allInstances()->size() > 0
```

Listing 6.24: A simple invariant using allInstances().

```
1  public privileged aspect InvAspect {
2
3      protected Map<String, Map> allInstances =
4        new HashMap<String, Map>();
5
6      after(Transaction aClass) :
7        execution(Transaction.new(..)) && this(aClass) {
8
9          Map<Transaction, Object> instanceMap;
10
11         instanceMap = (Map<Transaction, Object>)
12           this.allInstances.get(aClass.getClass()
13           .getCanonicalName());
14
15         if (instanceMap == null) {
16             instanceMap =
17             new WeakHashMap<Transaction, Object>();
18         }
19         // no else.
20
21         instanceMap.put(aClass, null);
22
23         this.allInstances.put(aClass.getClass()
24           .getCanonicalName(), instanceMap);
25     }
26
27     protected pointcut
28       allConstructors(Transaction aClass):
29         execution(Transaction.new(..)) && this(aClass);
30
31     after(Transaction aClass) : allConstructors(aClass) {
32         if (!((new OclSet<Transaction>((Set<Transaction>)
33           this.allInstances.get(aClass.getClass()
34           .getCanonicalName()).keySet())).size()
35           > new Integer(0))) {
36                 throw new RuntimeException(
37                   "Error: Constraint was violated.");
38         }
39         // no else.
40     }
41 }
```

Listing 6.25: An aspect instrumenting an invariant using allInstances().

solution which was developed during the *OCL2J* approach [BDL04, p. 41ff] for the constraint of Listing 6.24. The aspect creates a map which collects maps of all instances of all types on which the operation `allInstances()` is called using the *Canonical Name* of the stored types as keys. An advice observes all constructors of any of these types and collects all instances which were created. Here again, the special Java class `WeakHashMap` is used to store the instances to avoid problems during garbage collection.

The presented solution for `allInstances()` works in Java. However, Jos Warmer and Anneke Kleppe already mentioned in [WK04] the use of the operation `allInstances()` should be avoided. The use of `allInstances()` could cause a lot of overhead if the operation is used on a type which possesses a lot of instances during runtime.

Chapter 7

GUI Implementation and Test

This chapter illustrates briefly, how the graphical user interface (GUI) of the developed code generator has been implemented and how the code generator has been tested.

7.1 The Code Generation Wizard

As mentioned in section 5.1, the code generator has been implemented as a set of Eclipse plug-ins. The central class of the code generator is the class `Ocl2Java` which was also introduced in section 5.1.

In addition to this implementation, a wizard which provides a GUI to select a model and constraints for the code generation and a target directory for the generated code has been developed. It also provides some settings like whether or not inheritance will be enforced for instrumented pre-, postconditions and invariants. The wizard was implemented using the GUI elements of the *Eclipse Plug-in Development Environment (PDE)*. A screenshot of the wizard is shown in figure 7.1.

Figure 7.2 shows a sequence diagram which illustrates how a user can use the GUI to generate code. In the beginning, the GUI and the code generator are created. After that, the user has to select a model and constraints for the code generation before he can specify a target folder for the generated code and other settings to configure the code generation and the features which were pointed out by Katrin Eisenreich (see section 4.1.3). Afterwards the code generation is started. The GUI delegates this request to the `Ocl2Java` instance. This recursively creates the code.

7.2 Tests on the Implementation

The implemented Java code generator has been tested on two different levels: At the first level the fragment code generation was tested with many difference (diff) tests, comparing the generated code with text files containing the expected code. At the second level the code instrumentation was tested using a *jUnit* test suite which tested if the instrumented constraints were indeed enforced and evaluated at runtime. Both test suites were implemented in a test plug-in

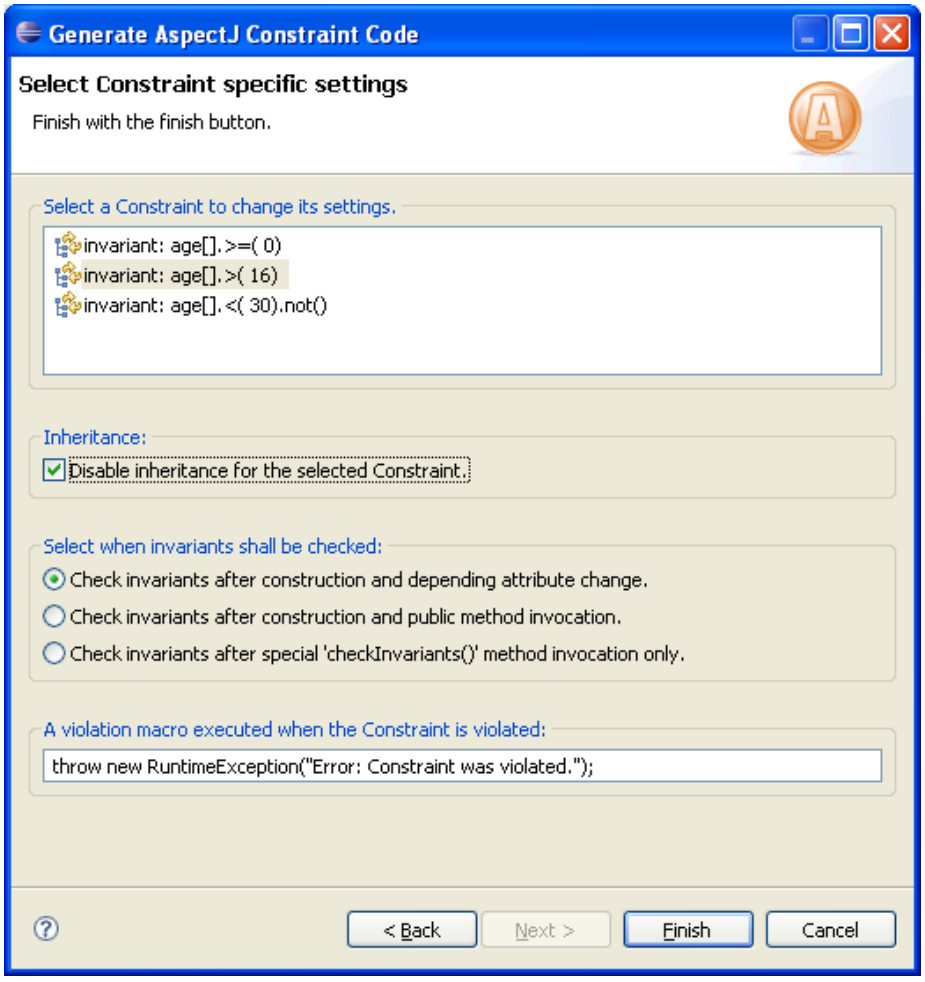

Figure 7.1: The code generation wizard of the Ocl2Java implementation.

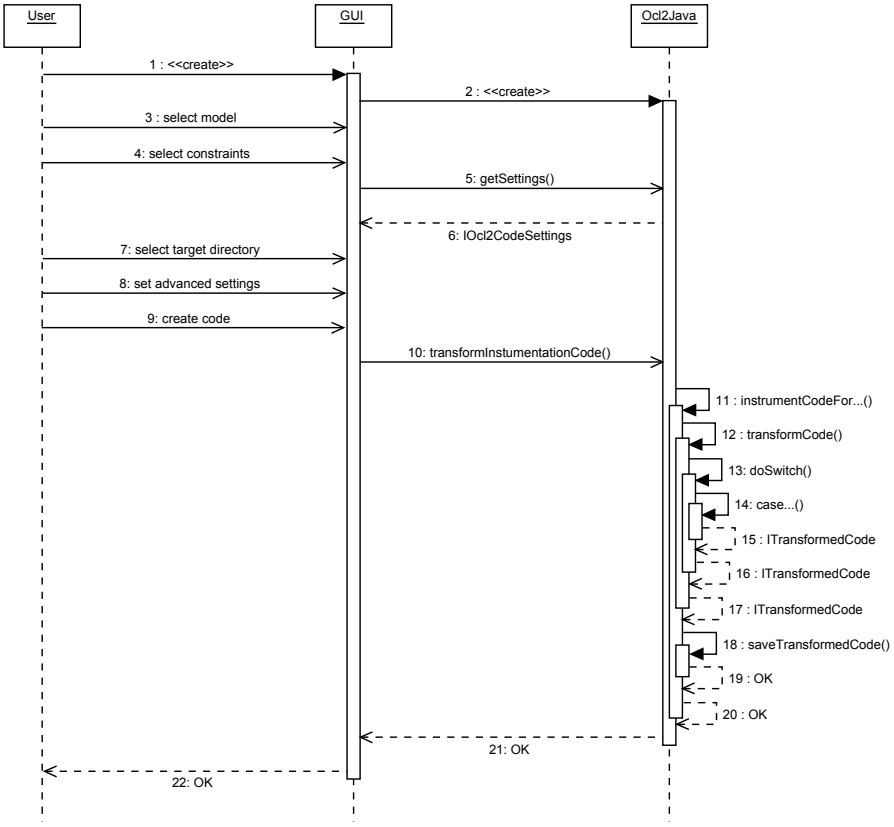

Figure 7.2: Code generation using the Ocl2Java GUI.

called `tudresden.ocl20.pivot.ocl2java.test` and will be explained in the following.

7.2.1 Fragment Generation

To test the fragment generation of the implemented code generator, a *jUnit* test suite which compares the generated fragment code for given constraints with expected constraint code provided in text files has been implemented. The constraints used for the test suite are based on the *royal and loyal* example by Warmer and Kleppe [WK04] which can be found in appendix D.

The test suite has been implemented in two classes called `FragmentTest` and `InstrumentationTest`. The first class compares generated code fragments, the second class compares generated instrumentation code. Both classes together contain 138 test cases which provided good support during the code generator development and can be used in the future for refactoring of the code generator as well.

7.2.2 Fragment Instrumentation

To test the generated instrumentation code (or aspects), another test suite which is based again on the *royal and loyal* example by Warmer and Kleppe [WK04] (which can be found in appendix D) has been developed. The *royal and loyal* example and its constraints have been implemented in a plug-in called `tudresden.ocl20.pivot.examples.royalandloyal`. Another plug-in called `tudresden.ocl20.pivot.examples.royalandloyal.constraints` contains a test suite with 36 test cases which check the generated *AspectJ* code for 52 constraints (the constraints which have been instrumented for this test suite are also available in appendix D). The implemented test suite provided good help to test all the instrumentation variants of the different kinds of OCL constraints.

7.2.3 Performance Test

To test the performance of the newly developed code generator, the generated code has been compared with generated code of the Java code generator of the *DOT2*. Both code generators were used to generate code for nine constraints which were provided as an example with the code generator of the *DOT2*. For this test only constraints which were accepted by the OCL parsers of both toolkit versions without errors were selected.

A simple test suite which tests the instrumented constraints for both packages of generated code has been implemented. Although such a small test suite can not be used as a stable case study, this simple test shows the improvement of the new code generator. The results of the performance test are shown in table 7.1.

This performance test shows that the new code generator improves both, performance and code length. The old code generator created code with a total length of 2251 lines of code (including the source code of the instrumented model). The new Ocl2Java code generator created 303 lines of aspect code or 1244 lines of code including the source code of the instrumented model. That is a decrease of 54%!

	DOT2	**DOT4Eclipse**
Lines of Code (incl. model code)	2251 LOC	941 + 303 = 1244 LOC (-54.7 %)
Avg. Test Suite Execution Time	157 ms	64 ms (-59.2 %)

Table 7.1: The results of the performance test with the code generators of different toolkit releases.

For the performance test, the developed test suite has been executed for both instrumentation versions ten times and an average execution time has been calculated. The old code generator caused an average execution time of 157 milliseconds, the new Ocl2Java code generator caused an average execution time of 64 milliseconds, which denotes a decrease of 59%!

More case studies or benchmark could be developed to investigate the improvements more specifically. For example, the *Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency* developed at the University of Bremen [GKB08] could be used to examine the improvements in more detail.

Chapter 8

Evaluation and Outlook on Future Works

This chapter will evaluate if the developed code generator fulfills the tasks of this work and if the analyzed requirements were realized. At first, the major tasks of this work will be discussed briefly. Afterwards the provided features of the code generation will be evaluated. Finally, this chapter will point out some tasks and features which were not realized and could be tasks of future works.

8.1 The Task of this Work

The tasks of this work have been the evaluation of the work of Katrin Eisenreich regarding constraint code generation [Eis06] and the development of a Java code generator for *Dresden OCL2 for Eclipse*. Aspect-oriented programming techniques should be used to realize the code fragment instrumentation and the OCL sub set supported by the *Essential OCL* meta model should be completely implemented. The implementation should be tested and evaluated.

The result of this work is the implementation of a Java code generator which can be released with the next release of *DOT4Eclipse*. The new code generator is more efficient than the code generator of the *DOT2*. The code is easier to read, shorter and has been tested intensively.

8.2 The Provided Features

This section will evaluate if the developed code generator fulfills the requirements pointed out by the minor thesis of Katrin Eisenreich [Eis06] which was discussed in section 4.1. Here again, three feature groups will be separated: The variation of fragment generation, the variation of fragment instrumentation and the parameterization of the whole code generation.

8.2.1 Variation of the Fragment Generation

One feature point in the variation of fragment generation was the set of supported constraints. The code generator of the *DOT2* and the *OCL2J approach*

only support pre-, postconditions and invariants. The newly developed code generator supports pre-, postconditions and invariants as well, but also body and let expressions, definitions and derived values. Only messages and states are not supported because the *Pivot Model* and *Essential OCL* do not provide any meta model elements for OCL messages and states.

Another feature are coding conventions. Like the old code generator, the new one does not support special settings for coding conventions. However, Eclipse supports features to format code very easily. Thus the users of the new code generator could use Eclipse to adapt the generated code to their preferred structure. Although the new code generator does not support features for code formatting, the code is easier to read than the generated code of the *DOT2* code generator. Short experiments about efficiency showed that the code seems to be more efficient and shorter than the code of the *DOT2* code generator also (see section 7.2.3).

The feature point of target languages is realized as in the old code generator. Only the generation of Java code is supported. Future works could try to adapt the code generator to other target languages like *C++* or *SQL*.

The type representation of the new code generator is based on a type mapping such as in the *OCL2J approach*. The old code generator of the *DOT2* used a standard library. The resulted code of this design decision of the new code generator seems to be both, easier to read and more efficient (see section 7.2.3).

The last feature of the fragment generation feature group is the technique used for fragment generation. The old code generator contained the code for the fragment generation directly in its traversal mechanism. The new code generator uses *StringTemplates* to generate code fragments. Thus the new code generator is more flexible and more adaptable than the old code generator.

8.2.2 Variation of the Fragment Instrumentation

The features of this feature group are closely linked, because many of them depend on the used instrumentation technology. The code generator of the *DOT2* used a parser which instrumented the instrumentation code directly into the constrained Java classes. The newly developed code generator uses *aspect-oriented programming* instead, such as the code generator of the *OCL2J approach*. This decision improves efficiency, reversibility of the code instrumentation, and supports easy and independent refactoring of constraint and constrained code. Source code and byte code can both be instrumented which was not possible using the old code generator. Also the support of special OCL operations such as `oclIsNew()` and `allInstances()` can be provided.

Another feature in this group is the reaction on constraint violations. The new code generator uses *violation macros* like the old code generator to configure the violation reaction. The new code generator supports constraint specific configurations of violation macros. Thus, some constraints can throw a runtime exception and others can simply print a warning on the console. By now, the new code generator does not provide possibilities to configure the violation macro with parameters such as the name of the violated constraint or the constrained class. Such parameterization could be approached in future works.

8.2.3 Parameterization of the Code Generation

The last feature group describes all features provided to configure the code generation. The new code generator provides good support to select constraints from a model for which constraint code will be generated and instrumented. Constraints can be selected manually or by group (for example all invariants). Such possibilities were not supported by the old code generator of the *DOT2*.

Furthermore, the new code generator supports both general and constraint specific settings for inheritances and invariant verification which were both not possible in the old code generator. For pre-, postconditions and invariants the inheritance can be enabled or disabled. Future work could examine if better mechanisms could be implemented to support *Liskov's substitution principle* [WK04, p. 145].

To evaluate invariants during runtime, the new code generator provides three different strategies which were explained in section 4.1.3. The first two strategies showed some minor problems during testing (see section 6.6). These problems could be solved in future works.

8.3 Outlook on Future Works

During the evaluation of the provided features some disadvantages and deficits of the new code generator were pointed out. They could be solved in future works. Some of these tasks are:

- The type mapping from primitive types in OCL to primitive and/or wrapper types in Java could be reconsidered. The actual strategy to map all types to wrapper types causes some problems which were mentioned in section 5.2.1.
- Other languages than Java could be implemented as target languages of the code generation. A good test for the code generator architecture would be an adaption to *C++* for which aspect-oriented techniques are also provided, for example by the language extension *AspectC++* [Asp09a]. Declarative implementations would be interesting as well. For example the tool *Ocl22Sql* of the *DOT2* could be adapted to *DOT4Eclipse* by adapting the developed code generator.
- The violation macro technique could be improved by providing the user the possibility to use some variables in its own defined violation macros. Such variables could contain the name of the violated constraint or the name of the constrained class.
- The support of *Liskov's substitution principle* for constraints could be evaluated and eventually implemented.
- The problems pointed out using the different verification strategies for invariants could be solved or avoided (see section 6.6).
- The *Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency* developed at the University of Bremen [GKB08] could be used to examine the improvements of the new Java code generator in more detail.

Appendix A

Type Mapping

This appendix presents an overview over all OCL types and their mapped correspondences in Java. The type mapping is explained in section 5.2.

Type in OCL	**Type in Java**
`Bag<T>`	`tudresden.ocl20.pivot.ocl2java.types.OclBag<T>`
`Boolean`	`java.lang.Boolean`
`Enumeration`	`java.lang.Enum<T>`
`Integer`	`java.lang.Integer`
`Real`	`java.lang.Float`
`OclAny`	`java.lang.Object`
`OclInvalid`	*no mapping provided*
`OclType`	`java.lang.Class`
`OclUndefined`	*no type just* `'null'`
`OclVoid`	*an empty piece of code*
`OrderedSet<T>`	`tudresden.ocl20.pivot.ocl2java.types.OclOrderedSet<T>`
`Sequence<T>`	`tudresden.ocl20.pivot.ocl2java.types.OclSequence<T>`
`Set<T>`	`tudresden.ocl20.pivot.ocl2java.types.OclSet<T>`
`String`	`java.lang.String`
`TupleType(<attribute1>: <type1>, ...)`	`java.util.HashMap<String, Object>`
`UnlimitedNatural`	`java.lang.Long`

Appendix B

Operation Mapping

This appendix presents an overview over all OCL operations and their mapped correspondences in Java. The operation mapping is explained in section 5.3.2.

```
1 java.lang.Math.abs(<numericLiteral>);
```

Listing B.1: <numericLiteral>.abs()

```
1 (new OclSet<objectType>((java.util.Set<objectType>)
2     this.allInstances.get(<object>.getClass()
3         .getCanonicalName()).keySet()));
```

Listing B.2: <object>.allInstances()

```
1 (<booleanLiteral1> && <booleanLiteral2>);
```

Listing B.3: <booleanLiteral1> and <booleanLiteral2>

```
1 <collectionLiteral>.append(<collectionItem>);
```

Listing B.4: <collectionLiteral>.append(<collectionItem>)

```
1 <collectionLiteral>.asBag();
```

Listing B.5: <collectionLiteral>.asBag()

```
1 <collectionLiteral>.asOrderedSet();
```

Listing B.6: <collectionLiteral>.asOrderedSet()

```
1 <collectionLiteral>.asSequence();
```

Listing B.7: <collectionLiteral>.asSequence()

```
1 <collectionLiteral>.asSet();
```

Listing B.8: <collectionLiteral>.asSet()

```
1 <collectionLiteral>.get(<integerLiteral>);
```

Listing B.9: <collectionLiteral>.at(<integerLiteral>)

```
1 <stringLiteral2>.concat(<stringLiteral2>);
```

Listing B.10: <stringLiteral2>.concat(<stringLiteral2>)

```
1 <collectionLiteral>.count(<collectionItem>);
```

Listing B.11: <collectionLiteral>.count(<collectionItem>)

```
1 (<integerLiteral1> / <integerLiteral2>);
```

Listing B.12: <integerLiteral1>.div(<integerLiteral2>)

```
1 ((Float) <integerLiteral1> / (Float) <integerLiteral2>);
```

Listing B.13: <integerLiteral1> / <integerLiteral2>

```
1 (<realLiteral1> / <realLiteral1>);
```

Listing B.14: <realLiteral1> / (<realLiteral2>)

```
1 <object1>.equals(<object2>);
```

Listing B.15: <object1> = <object2>

```
1 (((Object) <numericLiteral1>) == <numericLiteral2>);
```

Listing B.16: <numericLiteral1> = <numericLiteral2>

```
1 <collectionLiteral>.excludes(<collectionItem>);
```

Listing B.17: <collectionLiteral>.excludes(<collectionItem>)

```
1 <collectionLiteral1>.excludesAll(<collectionLiteral2>);
```

Listing B.18: <collectionLiteral1>.excludesAll(<collectionLiteral2>)

```
1 <collectionLiteral>.excluding(<collectionItem>);
```

Listing B.19: <collectionLiteral>.excluding(<collectionItem>)

```
1 <collectionLiteral>.first();
```

Listing B.20: <collectionLiteral>.first()

```
1 <collectionLiteral>.flatten();
```

Listing B.21: <collectionLiteral>.flatten()

```
((Integer) java.lang.Math.floor(<numericLiteral>));
```

Listing B.22: <numericLiteral>.floor()

```
(<numericLiteral1> > <numericLiteral2>);
```

Listing B.23: <numericLiteral1> > <numericLiteral2>

```
(<numericLiteral1> >= <numericLiteral2>);
```

Listing B.24: <numericLiteral1> >= <numericLiteral2>

```
(!<booleanLiteral1> || <booleanLiteral2>);
```

Listing B.25: <booleanLiteral1>.implies(<booleanLiteral2>)

```
<collectionLiteral>.contains(<collectionItem>);
```

Listing B.26: <collectionLiteral>.includes(<collectionItem>)

```
<collectionLiteral1>.containsAll(<collectionLiteral2>);
```

Listing B.27: <collectionLiteral1>.includesAll(<collectionLiteral2>)

```
<collectionLiteral>.including(<collectionItem>);
```

Listing B.28: <collectionLiteral>.including(<collectionItem>)

```
<collectionLiteral>.indexOf(<collectionItem>);
```

Listing B.29: <collectionLiteral>.indexOf(<collectionItem>)

```
<collectionLit>.insertAt(<numericLit>, <collectionItem>);
```

Listing B.30: <collectionLit>.insertAt(<numericLit>, <collectionItem>)

```
<collectionLiteral1>.intersection(<collectionLiteral2>);
```

Listing B.31: <collectionLiteral1>.intersection(<collectionLiteral2>)

```
<collectionLiteral1>.isEmpty();
```

Listing B.32: <collectionLiteral1>.isEmpty()

```
<collectionLiteral1>.last();
```

Listing B.33: <collectionLiteral1>.last()

```
(<numericLiteral1> < <numericLiteral2>);
```

Listing B.34: <numericLiteral1> < <numericLiteral2>

```
1 (<numericLiteral1> <= <numericLiteral2>);
```

Listing B.35: <numericLiteral1> <= <numericLiteral2>

```
1 java.lang.Math.max(<numericLiteral1>, <numericLiteral2>);
```

Listing B.36: <numericLiteral1>.max(<numericLiteral2>)

```
1 java.lang.Math.min(<numericLiteral1>, <numericLiteral2>);
```

Listing B.37: <numericLiteral1>.min(<numericLiteral2>)

```
1 (<numericLiteral1> - <numericLiteral2>);
```

Listing B.38: <numericLiteral1> - <numericLiteral2>

```
1 <collectionLiteral1>.minus(<collectionLiteral2>);
```

Listing B.39: <collectionLiteral1> - <collectionLiteral2>

```
1 (<numericLiteral1> * <numericLiteral2>);
```

Listing B.40: <numericLiteral1> * <numericLiteral2>

```
1 (<numericLiteral1> && <numericLiteral2>);
```

Listing B.41: <numericLiteral1>.mod(<numericLiteral2>)

```
1 -<numericLiteral1>;
```

Listing B.42: -<numericLiteral>

```
1 !<booleanLiteral>;
```

Listing B.43: not <booleanLiteral>

```
1 <collectionLiteral1>.notEmpty();
```

Listing B.44: <collectionLiteral1>.notEmpty()

```
1 !<object1>.equals(<object2>);
```

Listing B.45: <object1> <> <object2>

```
1 !((Object) <numericLiteral1>).equals(<numericLiteral2>);
```

Listing B.46: <numericLiteral1> <> <numericLiteral2>

```
1 ((<type>) <object>);
```

Listing B.47: <object>.oclAsType(<type>)

```
1 (<object> == null);
```

Listing B.48: <object>.oclIsInvalid()

```
1 this.newInstances.containsKey(<object>);
```

Listing B.49: <object>.oclIsNew()

```
1 (<object> instanceof <type>);
```

Listing B.50: <object>.oclIsKindOf(<type>)

```
1 (<object>.getClass().getCanonicalName().equals("<type>");
```

Listing B.51: <object>.oclIsTypeOf(<type>)

```
1 (<object> == null);
```

Listing B.52: <object>.oclIsUndefined()

```
1 (<booleanLiteral1> || <booleanLiteral2>);
```

Listing B.53: <booleanLiteral1> or <booleanLiteral2>

```
1 (<numericLiteral1> + <numericLiteral2>);
```

Listing B.54: <numericLiteral1> + <numericLiteral2>

```
1 <collectionLiteral>.prepend(<collectionItem>);
```

Listing B.55: <collectionLiteral>.prepend(<collectionItem>)

```
1 <collectionLiteral1>.product(<collectionLiteral2>);
```

Listing B.56: <collectionLiteral1>.product(<collectionLiteral2>)

```
1 java.lang.Math.round(<numericLiteral>);
```

Listing B.57: <numericLiteral>.round()

```
1 <collectionLiteral>.size();
```

Listing B.58: <collectionLiteral>.size()

```
1 <stringLiteral>.length();
```

Listing B.59: <stringLiteral>.size()

```
1 <collectionLiteral>.subOrderedSet();
```

Listing B.60: <collectionLiteral>.subOrderedSet();

```
<collectionLiteral>.subSequence();
```

Listing B.61: <collectionLiteral>.subSequence();

```
<stringLit>.substring(<integerLit1> - 1, <integerLit2>);
```

Listing B.62: <stringLit>.substring(<integerLit1>, <integerLit2>)

```
<genericType> <resultVar>;
<resultVar> = new <genericType>(0);

/* Compute the result of a sum operation. */
for (<genericType> <elementName> : <sourceExp>) {
    <resultVar> += <elementName>;
}
```

Listing B.63: <collectionLiteral>.sum()

```
<setLiteral>.symmetricDifference(<collectionLiteral>);
```

Listing B.64: <setLiteral>.symmetricDifference(<collectionLiteral>)

```
Integer.parseInt(<numericLiteral>);
```

Listing B.65: <numericLiteral>.toInteger()

```
Float.parseFloat(<numericLiteral>);
```

Listing B.66: <numericLiteral>.toReal()

```
<collectionLiteral1>.union(<collectionLiteral2>);
```

Listing B.67: <collectionLiteral1>.union(<collectionLiteral2>)

```
(<booleanLiteral1> ^ <booleanLiteral2>);
```

Listing B.68: <booleanLiteral1> xor <booleanLiteral2>

Appendix C

Code Fragment Templates

These appendix lists all templates used to generate fragment code from OCL constraints. The templates are defined using the template language *StringTemplate* which is introduced briefly in section 3.2. The fragment code generation is explained in section 5.3.

```
1  collectionLiteralExp(collectionName, collectionType,
2    elementCodes, elementExps) ::= <<
3      $collectionType$ $collectionName$;
4      $collectionName$ = new $collectionType$();
5
6      $if(elementExps)$
7        $elementCodes, elementExps:{code, exp |
8          $if(code)$$code$$endif$
9          $exp$
10       }; separator = "\n"$
11     $endif$
12 >>
```

Listing C.1: A template for collection literal expressions.

```
1  collectionLiteralExp_collectionItem(collectionName,
2    itemExp) ::= <<
3      $collectionName$.add($itemExp$);
4  >>
```

Listing C.2: A template for collection items in collection literal expressions.

```
1  collectionLiteralExp_collectionRange(collectionName,
2    indexVar, indexType, firstExp, lastExp) ::= <<
3      /* TODO: Auto-generated initialization
4         does only work for numeric values. */
5      for ($indexType$ $indexVar$ = $firstExp$;
6        $indexVar$ <= $lastExp$; $indexVar$++) {
7          $collectionName$.add($indexVar$);
8      }
9  >>
```

Listing C.3: A template for collection initialization by collection range expressions.

```
enumLiteralExp(enumerationName, literalName) ::= <<
    $enumerationName$.$literalName$
>>
```

Listing C.4: A template for enumeration literal expressions.

```
ifExp(ifCode, ifExp, thenCode, thenExp, elseCode,
  elseExp, resultVar, resultType) ::= <<
    $resultType$ $resultVar$;

    $ifCode$

    if ($ifExp$) {
      $thenCode$
      $resultVar$ = $thenExp$;
    } else {
      $elseCode$
      $resultVar$ = $elseExp$;
    }
>>
```

Listing C.5: A template for if expressions.

```
invalidLiteralExp() ::= <<
null
>>
```

Listing C.6: A template for invalid literal expressions.

```
iterateExp(sourceCode, sourceExp, sourceGenericType,
  itVar, bodyCode, bodyExp, resultType, resultVar,
  resultVarInitCode, resultVarInitExp) ::= <<
    $sourceCode$
    $resultVarInitCode$

    $resultType$ $resultVar$;
    $resultVar$ = $resultVarInitExp$;

    /* IterateExp: Iterate through all elements
       and perform an operation on them. */
    for ($sourceGenericType$ $itVar$ : $sourceExp$) {
      $bodyCode$$resultVar$ = $bodyExp$;
    }
>>
```

Listing C.7: A template for iterate expressions.

```
1  iteratorExp_any(sourceCode, sourceExp, itVar, itType,
2    bodyCode, bodyExp, resultVar) ::= <<
3      $sourceCode$
4      $itType$ $resultVar$;
5      $resultVar$ = null;
6
7      /* Iterator Any: Iterate through the elements and
8          return one element that fulfills the condition. */
9      for ($itType$ $itVar$ : $sourceExp$) {
10       $bodyCode$
11       if ($bodyExp$) {
12         $resultVar$ = $itVar$;
13         break;
14       }
15       // no else
16     }
17 >>
```

Listing C.8: A template for 'any' iterator expressions.

```
1  iteratorExp_collect(sourceCode, sourceExp, itVar, itType,
2    bodyCode, bodyExp, resultVar, resultType, addOp) ::= <<
3      $sourceCode$
4      $resultType$ $resultVar$;
5      $resultVar$ = new $resultType$();
6
7      /* Iterator Collect: Iterate through all elements and
8         collect them. Elements which are collections are
9         flattened. */
10     for ($itType$ $itVar$ : $sourceExp$) {
11       $bodyCode$
12       $resultVar$.$addOp$($bodyExp$);
13     }
14 >>
```

Listing C.9: A template for 'collect' iterator expressions.

```
1  iteratorExp_collectNested(sourceCode, sourceExp, itVar,
2    itType, bodyCode, bodyExp, resultVar, resultType) ::= <<
3      $sourceCode$
4      $resultType$ resultVar;
5      $resultVar$ = new $resultType$();
6
7      /* Iterator CollectNested: Iterate through all
8          elements and collect them. */
9      for ($itType$ $itVar$ : $sourceExp$) {
10       $bodyCode$
11       $resultVar$.add($bodyExp$);
12     }
13 >>
```

Listing C.10: A template for 'collectNested' iterator expressions.

```
1  iteratorExp_exists(sourceCode, sourceExp, itVar, itType,
2    bodyCode, bodyExp, resultVar) ::= <<
3      $sourceCode$
4      $booleanType()$ $resultVar$;
5      $resultVar$ = false;
6
7      /* Iterator Exists: Iterate and check, if any element
8         fulfills the condition. */
9      for ($itType$ $itVar$ : $sourceExp$) {
10       $bodyCode$
11       if ($bodyExp$) {
12         $resultVar$ = true;
13         break;
14       }
15       // no else
16     }
17 >>
```

Listing C.11: A template for 'exists' iterator expressions.

```
1  iteratorExp_forAll(sourceCode, sourceExp, itVar, itType,
2    bodyCode, bodyExp, resultVar) ::= <<
3      $sourceCode$
4      $booleanType()$ $resultVar$;
5      $resultVar$ = true;
6
7      /* Iterator ForAll: Iterate and check, if all elements
8         fulfill the condition. */
9      for ($itType$ $itVar$ : $sourceExp$) {
10       $bodyCode$
11       if (!$bodyExp$) {
12         $resultVar$ = false;
13         break;
14       }
15       // no else
16     }
17 >>
```

Listing C.12: A template for 'forAll' iterator expressions.

```
iteratorExp_isUnique(sourceCode, sourceExp, itVar, itType,
  bodyCode, bodyExp, collectionVar, resultVar) ::= <<
    $sourceCode$
    $setType()$<$itType$> $collectionVar$;
    $booleanType()$ $resultVar$;

    $collectionVar$ = new $setType()$<$itType$>();
    $resultVar$ = true;

    /* Iterator IsUnique: Iterate and check, if all
       elements are unique. */
    for ($itType$ $itVar$ : $sourceExp$) {
      $bodyCode$
      if ($collectionVar$.includes($bodyExp$)) {
        $resultVar$ = false;
        break;
      } else {
        $collectionVar$.add($bodyExp$);
      }
    }
>>
```

Listing C.13: A template for 'isUnique' iteraotor expressions.

```
// ——— IteratorExp for Iterator One ———
iteratorExp_one(sourceCode, sourceExp, itVar, itType,
  bodyCode, bodyExp, resultVar) ::= <<
    $sourceCode$
    $booleanType()$ $resultVar$;
    $resultVar$ = false;

    /* Iterator One: Iterate and check, if exactly
       one element fulfills the condition. */
    for ($itType$ $itVar$ : $sourceExp$) {
      $bodyCode$
      if ($bodyExp$) {
        if ($resultVar$) {
          // Found a second element.
          $resultVar$ = false;
          break;
        } else
          // Found a first element.
          $resultVar$ = true;
              }
      }
      // no else
    }
>>
```

Listing C.14: A template for 'one' iterator expressions.

```
1  iteratorExp_reject(sourceCode, sourceExp, itVar, itType,
2    bodyCode, bodyExp, resultVar, resultType) ::= <<
3      $sourceCode$
4      $resultType$ $resultVar$;
5      $resultVar$ = new $resultType$();
6
7      /* Iterator Reject: Select all elements which do
8         not fulfill the condition. */
9      for ($itType$ $itVar$ : $sourceExp$) {
10       $bodyCode$
11       if (!$bodyExp$) {
12         $resultVar$.add($itVar$);
13       }
14       // no else
15     }
16 >>
```

Listing C.15: A template for 'reject' iterator expressions.

```
1  iteratorExp_select(sourceCode, sourceExp, itVar, itType,
2    bodyCode, bodyExp, resultVar, resultType) ::= <<
3      $sourceCode$
4      $resultType$ $resultVar$;
5      $resultVar$ = new $resultType$();
6
7      /* Iterator Select: Select all elements
8         which fulfill the condition. */
9      for ($itType$ $itVar$ : $sourceExp$) {
10       $bodyCode$
11       if ($bodyExp$) {
12         $resultVar$.add($itVar$);
13       }
14       // no else
15     }
16 >>
```

Listing C.16: A template for 'select' iterator expressions.

```
1  iteratorExp_sortedBy(sourceCode, sourceExp, itVar, itVar2,
2    itType, bodyCode, bodyExp, bodyCode2, bodyExp2,
3    comparatorName, compareResult, resultVar, resultType)
       ::= <<
4      $sourceCode$
5      $resultType$ $resultVar$;
6      java.util.Comparator<$itType$> $comparatorName$;
7
8      $resultVar$ = $sourceExp$;
9
10     $comparatorName$ = new java.util.Comparator<$itType$
         >() {
11
12       /** Method which compares two elements of the
             collection. */
13       public int compare($itType$ $itVar$, $itType$ $
             itVar2$) {
14         int $compareResult$;
15
16         $bodyCode$$bodyCode2$$compareResult$ = 0;
17
18         if ($bodyExp$ < $bodyExp2$) {
19           $compareResult$ = -1;
20         } else if ($bodyExp$ > $bodyExp2$) {
21           $compareResult$ = 1;
22         }
23
24         return $compareResult$;
25       }
26     };
27
28     $resultVar$ = java.util.Collections.sort($resultVar$,
         $comparatorName$);
29 >>
```

Listing C.17: A template for 'sortedBy' iterator expressions.

```
1  letExp(varType, varName, initCode, initExp, inCode) ::= <<
2      $varType$ $varName$;
3      $if(initCode)$
4        $initCode$
5      $endif$
6
7      $if(initExp)$
8        $varName$ = $initExp$;
9      $endif$
10
11     $inCode$
12 >>
```

Listing C.18: A template for let expressions.

```
1 literalExp(type, value) ::= <<
2     new $type$($value$)
3 >>
```

Listing C.19: A template for literal expressions.

```
1 propertyCallExp(sourceExp, propertyName) ::= <<
2     $sourceExp$.$propertyName$
3 >>
```

Listing C.20: A template for property call expressions.

```
1 propertyCallExpOnTuple(sourceExp, propertyName) ::= <<
2    $sourceExp$.get("$propertyName$")
3 >>
```

Listing C.21: A template for property call expressions on tuple types.

```
1 stringLiteralExp(value) ::= <<
2     "$value$"
3 >>
```

Listing C.22: A template for string literal expressions.

```
1  tupleLiteralExp(tupleName, argNames, argCodes, argExps)
       ::= <<
2      $tupleType()$ $tupleName$;
3      $tupleName$ = new $tupleType()$();
4
5      $if(argNames)$
6        $argNames, argCodes, argExps:{name, code, exp |
7          $if(code)$
8            $code$
9          $endif$
10
11         $tupleName$.put($name$, $exp$);
12       }; separator = "\n"$
13    $endif$
14 >>
```

Listing C.23: A template for tuple literal expressions.

```
1 // ——— UndefinedLiteralExp ———
2 undefinedLiteralExp() ::= <<
3     null
4 >>
```

Listing C.24: A template for undefined literal expressions.

Appendix D

The Royal and Loyal Example

The *Royal and Loyal example* was developed by Jos Warmer and Anneke Kleppe to explain the different features of OCL. The model was published in [WK04]. An adapted version which is used in the OCL examples of this work is shown in figure D.1. In the following some constraints are listed which were also used to test the developed Java code generator.

```
1  -- Body Expression 1:
2  context LoyaltyProgram::getServices(): Set
3  body: partners.deliveredServices
4
5  -- Body Expression 2:
6  context LoyaltyAccount::getCustomerName() : String
7  body: membership.card.owner.name
8
9  -- Definition 1:
10 context LoyaltyAccount
11 def: turnover : Real = transactions.amount->sum()
12
13 -- Definition 2:
14 context LoyaltyProgram
15 def: getServicesByLevel(levelName: String): Set(Service)
16      = levels->select(name = levelName).availableServices
            ->asSet()
17
18 -- Definition 3:
19 context Membership
20 def: getCurrentLevelName() : String = currentLevel.name
21
22 -- Definition 4:
23 context LoyaltyAccount
24 def: usedServices: Set(Service) = transactions.service->
       asSet()
25
26 -- Definition 5:
27 context Customer
28 def: initial: String = name.substring(1, 1)
```

Listing D.1: Constraints defined on the Royal and Loyal Example (part 1).

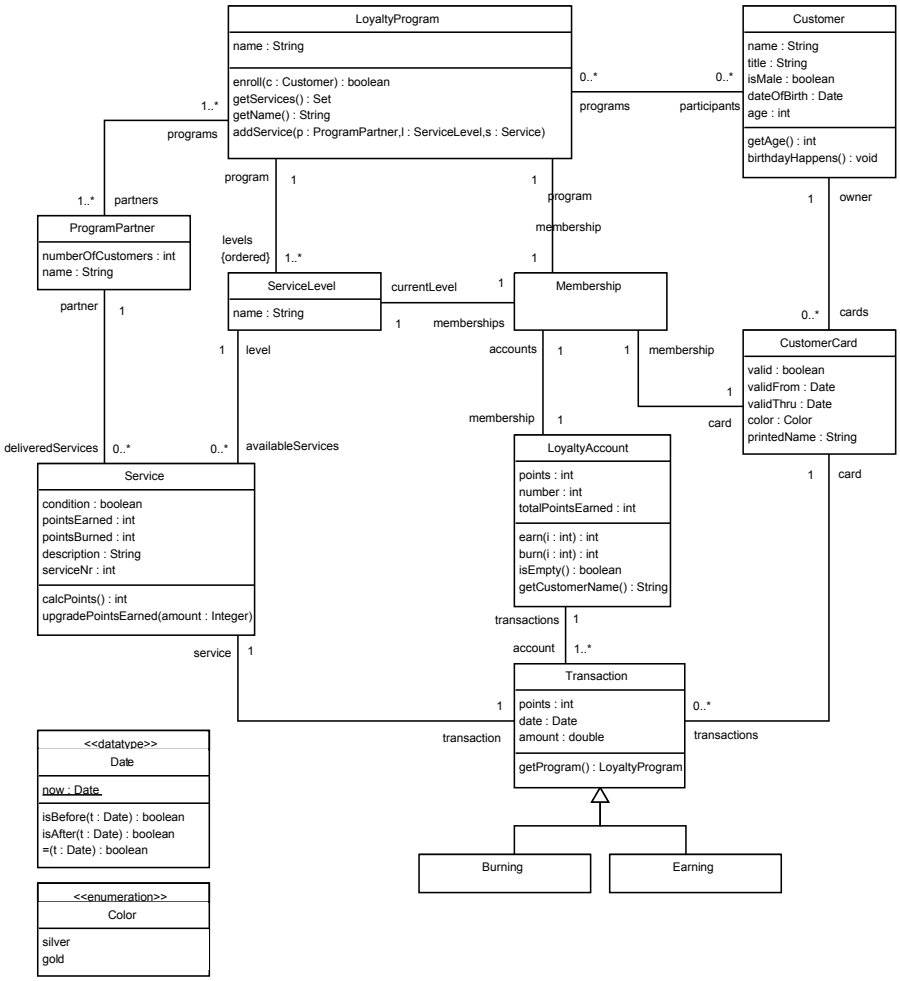

Figure D.1: The Royal and Loyal model by Jos Warmer et al. [WK04].

```
1  -- Definition 6:
2  context CustomerCard
3  def: getTotalPoints(d: Date) : Integer = transactions->
       select(date.isAfter(d)).points->sum()
4
5  -- Definition 7:
6  context CustomerCard
7  def: getAllInstances() : Set(CustomerCard) = self.
       allInstances()
8
9  -- Definition 8:
10 context ProgramPartner
11 def: getBurningTransactions(): Set(Transaction) =
12   self.deliveredServices.transaction->iterate(
13     t          : Transaction;
14     resultSet : Set(Transaction) = Set{} |
15
16     if (t.oclIsTypeOf( Burning )) then
17       resultSet->including( t )
18     else
19       resultSet
20     endif
21   )
22
23 -- Derived Value 1:
24 context CustomerCard::printedName
25 derive: owner.title.concat(' ').concat(owner.name)
26
27 -- Derived Value 2:
28 context LoyaltyAccount::totalPointsEarned
29 derive: transactions->select(oclIsTypeOf(Earning)).points
       ->sum()
30
31 -- Initial Expression 1:
32 context LoyaltyAccount::points
33 init: 0
34
35 -- Initial Expression 2:
36 context CustomerCard::valid
37 init: true
38
39 -- Initial Expression 3:
40 context LoyaltyAccount::transactions : Set(Transaction)
41 init: Set{}
42
43 -- Invariant 1:
44 context Customer
45 inv ofAge: age >= 18
```

Listing D.2: Constraints defined on the Royal and Loyal Example (part 2).

```
1  -- Invariant 2:
2  context CustomerCard
3  inv checkDates: validFrom.isBefore(validThru)
4
5  -- Invariant 3:
6  context LoyaltyProgram
7  inv knownServiceLevel: levels->includes(membership.
       currentLevel)
8
9  -- Invariant 4:
10 context Membership
11 inv correctCard: program.participants.cards->includes(self
       .card)
12
13 -- Invariant 5:
14 context Membership
15 inv levelAnColor:
16   currentLevel.name = 'Silver' implies card.color = Color
         ::silver
17   and
18   currentLevel.name = 'Gold' implies card.color = Color::
         gold
19
20 -- Invariant 6:
21 context LoyaltyProgram
22 inv minServices: partners->forAll(deliveredServices->size
       () >= 1)
23
24 -- Invariant 7:
25 context Customer
26 inv sizesAgree:
27     programs->size() = cards->select( valid = true )->size
           ()
28
29 -- Invariant 8:
30 context LoyaltyProgram
31 inv noAccounts:
32   partners.deliveredServices
33     ->forAll(pointsEarned = 0 and pointsBurned = 0)
34     implies membership.accounts->isEmpty()
35
36 -- Invariant 9:
37 context ProgramPartner
38 inv nrOfParticipants: numberOfCustomers = programs.
       participants->size()
39
40 -- Invariant 10:
41 context LoyaltyProgram
42 inv firstLevel: levels->first().name = 'Silver'
```

Listing D.3: Constraints defined on the Royal and Loyal Example (part 3).

```
1  -- Invariant 11:
2  context ProgramPartner
3  inv totalPoints: deliveredServices.transaction.points->sum
       () < 10000
4
5  -- Invariant 12:
6  context ProgramPartner
7  inv totalPointsEarning:
8    deliveredServices.transaction->select(oclIsTypeOf(
         Earning)).points->sum() < 10000
9
10 -- Invariant 13:
11 context CustomerCard
12 inv:
13 let correctDate : Boolean =
14   self.validFrom.isBefore(Date::now()) and
15   self.validThru.isAfter(Date::now())
16 in
17   if valid then
18     correctDate = false
19   else
20     correctDate = true
21   endif
22
23 -- Invariant 14:
24 context LoyaltyAccount
25 inv oneOwner: transactions.card.owner->asSet()->size() = 1
26
27 -- Invariant 15:
28 context LoyaltyAccount
29 inv: points > 0 implies transactions->exists(t | t.points
       > 0)
30
31 -- Invariant 16:
32 context Service
33 inv: self.oclIsUndefined() = false
34
35 -- Invariant 17:
36 context Service
37 inv: self.oclIsInvalid() = false
38
39 -- Invariant 18:
40 context Burning
41 inv: self.points = self.oclAsType(Transaction).points
42
43 -- Postcondition 1:
44 context LoyaltyProgram::enroll(c : Customer)
45 post: participants = participants@pre->including(c)
```

Listing D.4: Constraints defined on the Royal and Loyal Example (part 4).

```
1  -- Postcondition 2:
2  context LoyaltyProgram::addService(aPartner:
       ProgramPartner, aLevel: ServiceLevel, aService:
       Service)
3  post: partners.deliveredServices->includes(aService)
4
5  -- Postcondition 3:
6  context LoyaltyProgram::addService(aPartner:
       ProgramPartner, aLevel: ServiceLevel, aService:
       Service)
7  post: levels.availableServices->includes(aService)
8
9  -- Postcondition 4:
10 context LoyaltyProgram::addService(aPartner:
       ProgramPartner, aLevel: ServiceLevel, aService:
       Service)
11 post: partners.deliveredServices->includes(aService) and
       levels.availableServices->includes(aService)
12
13 -- Postcondition 5:
14 context LoyaltyAccount::isEmpty():Boolean
15 post: result = (points = 0)
16
17 -- Postcondition 6:
18 context Customer::birthdayHappens()
19 post: age = age@pre + 1
20
21 -- Postcondition 7:
22 context Service::upgradePointsEarned(amount: Integer)
23 post: calcPoints() = calcPoints@pre() + amount
24
25 -- Postcondition 8:
26 context Transaction::getProgram(): LoyaltyProgram
27 post: not result.oclIsNew()
28
29 -- Postcondition 9:
30 context Transaction::getProgram(): LoyaltyProgram
31 post: result = self.card.membership.program
32
33 -- Postcondition 10:
34 context Transaction::getProgram(): LoyaltyProgram
35 post: self.oclIsTypeOf(Transaction)
36
37 -- Postcondition 11:
38 context LoyaltyProgram::enroll(c : Customer)
39 post: membership = membership@pre
40
41 -- Precondition 1:
42 context LoyaltyProgram::enroll(c : Customer)
43 pre: c.name <> ''
```

Listing D.5: Constraints defined on the Royal and Loyal Example (part 5).

```
1  -- Precondition 2:
2  context LoyaltyProgram::addService(aPartner:
       ProgramPartner, aLevel: ServiceLevel, aService:
       Service)
3  pre: partners->includes(aPartner)
4
5  -- Precondition 3:
6  context LoyaltyProgram::addService(aPartner:
       ProgramPartner, aLevel: ServiceLevel, aService:
       Service)
7  pre: levels->includes(aLevel)
8
9  -- Precondition 4:
10 context LoyaltyProgram::addService(aPartner:
       ProgramPartner, aLevel: ServiceLevel, aService:
       Service)
11 pre:  partners->includes(aPartner) and levels -> includes(
       aLevel)
12
13 -- Precondition 5:
14 context Transaction::getProgram(): LoyaltyProgram
15 pre: self.oclIsTypeOf(Transaction)
```

Listing D.6: Constraints defined on the Royal and Loyal Example (part 6).

Bibliography

[ABR06] Apel, S. ; Batory, D. ; Rosenmüller, M.: On the Structure of Crosscutting Concerns: Using Aspects or Collaborations? In: *Proceedings of the 1st Workshop on Aspect-Oriented Produt Line Engineering (AOPLE'06) co-located with the 5th Int'l Conf. on Generative Programming and Component Engineering (GPCE'06)*, 2006, S. 20–24. – Available at http://www.softeng.ox.ac.uk/aople/aople1/AOPLE1-Proceedings.pdf

[AK07] Apel, S. ; Kästner, C.: Pointcuts, advice, refinements, and collaborations: similarities, differences, and synergies. In: *Innovations in Systems and Software Engineering* Bd. 3, Springer London, 2007, S. 281–289. – More information available at http://www.springerlink.com/content/08m600873g3044t4/

[Aßm03] Assmann, Uwe: *Invasive Software Composition*. Edition 2. Springer-Verlag Berlin, Heidelberg, New York, 2003. – ISBN 3–540–44385–1

[Asp09a] *The AspectC++ Project*. AspectC++ Project Website, 2009. – Available at http://www.aspectc.org/

[Asp09b] *The AspectJ Project*. Eclipse Project Website, 2009. – Available at http://www.eclipse.org/aspectj/

[BDF+04] Barnett, M. ; DeLine, R. ; Fähndrich, M. ; Leino, K. Rustan M. ; Schulte, Wolfram: Verification of object-oriented programs with invariants. In: *Special issue ECOOP 2003 Workshop on FTfJP* Bd. 3, ETH Zurich, Chair of Software Engineering, June 2004 (Journal of Object Technology), S. 27–56. – Available at http://www.jot.fm/issues/issue_2004_06/article2

[BDL04] Briand, L. C. ; Dzidek, W. ; Labiche, Y.: Using Aspect-Oriented Programming to Instrument OCL Constracts in Java / Carleton University Ottawa, Canada. 2004 (SCE-04-03). – Technical Report. – Available at http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-04-03.pdf

[BDL05] Briand, L. C. ; Dzidek, W. J. ; Labiche, Y.: Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In: Society, IEEE C. (Hrsg.): *21st*

IEEE International Conference on Software Maintenance (ICSM), Budapest, Hungary, September 25-30, 2005, S. 687–690. – Available at http://portal.acm.org/citation.cfm?id=1091907

[Böh06] Böhm, Oliver: *Aspektorientierte Programmierung mit AspectJ 5.* Edition 1. dpunkt.verlag GmbH, 2006. – ISBN 3–89864–330–1

[Brä07] Bräuer, Matthias: *Models and Metamodels in a QVT/OCL Development Environment.* Großer Beleg, May 2007. – Available at http://dresden-ocl.sourceforge.net/gbbraeuer/index.html

[Bra06] Brandt, Ronny: *Java-Codegenerierung und Instrumentierung von Java-Programmen in der metamodellbasierten Architektur des Dresden OCL Toolkit.* Großer Beleg, September 2006

[Bra07] Brandt, Ronny: *Ein OCL-Interpreter für das Dresden OCL2Toolkit basierend auf dem Pivotmodell.* Diploma Thesis, August 2007

[BSM+03] Budinsky, Frank ; Steinberg, David ; Merks, Ed ; Ellersick, Raymond ; Grose, Timothy J.: *Eclipse Modeling Framework: A Developer's Guide.* Edition 2. Addison-Wesley Professional, 2003. – ISBN 0–13–142542–0

[DBL06] Dzidek, W. J. ; Briand, L. C. ; Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Revised Selected Papers, Jean-Michel ed. by B. ed. by Bruel (Hrsg.): *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005* Bd. 3844, Springer-Verlag GmbH, 2006 (Lecture Notes in Computer Science ISBN: 3-540-31780-5), S. 10–19. – Available at http://simula.no/research/engineering/publications/Dzidek.2006.1/simula_pdf_file

[Eis06] Eisenreich, Katrin: *Varianzanalyse zur Generierung imperativen Codes aus OCL-Ausdrücken.* Großer Beleg, October 2006

[Fin99] Finger, Frank: *Java-Implementierung der OCL-Basisbibliothek.* Großer Beleg, July 1999. – Available at http://www-st.inf.tu-dresden.de/ocl/ff3/beleg.pdf

[Fin00] Finger, Frank: *Design and Implementation of a Modular OCL Compiler.* Diploma Thesis, March 2000. – Available at http://www-st.inf.tu-dresden.de/ocl/ff3/diplom.pdf

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns - Elements of Reusable Object-Oriented Software.* Edition 2. Addison-Wesley Professional, 1995. – ISBN 0–20–163361–2

[GJSB05] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *Java(TM) Language Specification*. Edition 3. Addison Wesley, 2005. – ISBN 0–32124–678–0

[GKB08] Gogolla, Martin ; Kuhlmann, Mirco ; Büttner, Fabian: A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In: *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'2008)* Bd. 5301, Springer, Berlin, 2008 (LNCS), S. 446–459. – Available at http://db.informatik.uni-bremen.de/publications/

[Hei05] Heidenreich, Florian: *SQL-Codegenerierung in der metamodellbasierten Architektur des Dresden OCL Toolkit*. Großer Beleg, May 2005

[Hei06] Heidenreich, Florian: *OCL-Codegenerierung für deklarative Sprachen*. Diploma Thesis, April 2006

[HH01] Heinrich Hussmann, Prof. D. h.: *Formal Specification of Software Systems*. Slides from lectures at the Technische Universität Dresden, Dezember 2001. – Available at http://st.inf.tu-dresden.de/fs/slides/fss5a-sl.pdf

[MDT09] *Eclipse Model Development Tools Project*. Eclipse Project Website, 2009. – Available at http://www.eclipse.org/modeling/mdt/

[Ock03] Ocke, Stefan: *Entwurf und Implementierung eines metamodellbasierten OCL-Compilers*. Diploma Thesis, June 2003. – Available at http://www-st.inf.tu-dresden.de/home/html/de/diplomarbeiten/DAOcke.pdf

[OMG97] Object Management Group – OMG: *Object Constraint Language Specification*. Version 1.1. September 1997. – Available at http://www.omg.org/docs/ad/97-08-08.pdf

[OMG06] Object Management Group – OMG: *Object Constraint Language, OMG Available Specification*. Version 2.0. May 2006. – Available at http://www.omg.org/docs/formal/06-05-01.pdf

[Sch98] Schmidt, Andreas: *Untersuchungen zur Abbildung von OCL-Ausdrücken auf SQL*. Diploma Thesis, September 1998

[Str09] *StringTemplate Website*. ANT Project Website, 2009. – Available at http://www.stringtemplate.org/

[Thi07] Thieme, Nils: *Reengineering des OCL2-Parsers*. Großer Beleg, November 2007. – Available at http://dresden-ocl.sourceforge.net/downloads/pdfs/gb_nils_thieme.pdf

[Ull06] Ullenboom, C.: *Java ist auch eine Insel - Programmieren für die Java 2-Plattform in der Version 5*. Edition 5. Galileo Computing, 2006. – ISBN 978–3–89842–747–0. – Available at http://openbook.galileodesign.de/javainsel5/

[Wie00] WIEBICKE, Ralf: *Utility Support for Checking OCL Business Rules in Java Programs*. Diploma Thesis, December 2000. – Available at http://rw7.de/ralf/diplom00/intro.html

[Wik09] *Unified Modeling Language (UML)*. Wikipedia - The Free Encyclopedia, February 2009. – Available at http://de.wikipedia.org/wiki/Unified_Modeling_Language

[WK04] WARMER, Jos ; KLEPPE, Anneke: *Object Constraint Language 2.0*. mitp-Verlag/Bonn, 2004. – ISBN 3–8266–1445–3. – Translated from the English publication. Original published at Pearson Education, Inc. 2003.

Confirmation

I confirm that I independently prepared the thesis and that I only used the references and auxiliary means indicated in the thesis.

Dresden, February 19, 2009