TUHH
Technische Universität Hamburg-Harburg

# Prototypes for Solving Consistency Problems in Model Engineering

**Master Thesis**

Submitted by:

Meng Xue

Informatik-Ingenieurwesen

Matriculation Number: 17263

Supervised by:

Prof. Dr. Ralf Möller (STS)
Prof. Dr. Rolf-Rainer Grigat (TI1)
M.Sc. Miguel Garcia (STS)

Hamburg, Germany
7th January 2006

# Declaration

Hereby I declare that this work has been prepared by me, all literally or content-related quotations from other sources are clearly pointed out, and no other sources or aids than the ones that are declared are used.

Meng Xue

Hamburg, Germany
7th Jan. 2006

# Acknowledgement

I would like to thank Prof. Dr. Ralf Möller and Prof. Dr. Rolf-Rainer Grigat for giving me the opportunity to work on this thesis project under their supervision.

I would also like to thank M. Sc. Miguel Garcia, who was very helpful in providing advice and direction on this thesis.

This project could not have been as fruitful as it was, without the support, interest and inputs from individuals and their willingness to share their experience and knowledge, and the time given to discuss related topics.

# Table of contents

**Abstract** 5

**Chapter 1 Overview** 6

**Chapter 2 Case Study** 7

2.1 Using ATL for Checking Models 7
2.2 Summary 10

**Chapter 3 Introduction to ILOG Business Rules** 11

3.1 The Challenge and Solution 11
3.2 ILOG Business Rule Studio Overview 12
3.3 Main Features of ILOG Business Rule Studio 13
3.4 Concepts 13
3.5 Summary 17

**Chapter 4 Prototype Based on Business Rules** 18

4.1 Advantages of Applying Business Rules 18
4.2 Transformation from OCL invariants to Business Rules 19
4.2.1 ILOG Rule Language Structure 19
4.2.2 Octopus Java Code Generation 21
4.2.3 Mapping from OCL to IRL 24
4.2.4 Changes Made to Octopus 29
4.3 Runtime Execution 32
4.4 Summary 35

**Chapter 5 Prototype Based on AOP and Back Navigation Algorithm** 36

5.1 General AOP Approach 36
5.2 AOP and Back Navigation Approach 40
5.2.1 The noAccounts Invariant Example 40
5.2.2 General Octopus Generated Non Side-effect-free Method Pattern 42
5.2.3 Pointcut for Association Class 44
5.2.4 Pointcut and Advice for allInstances 48
5.2.5 Back Navigation Algorithm 50
5.2.6 Drawback of allInstances and Weak Reference 56
5.2.7 Other Issues 58
5.2.8 Consistency Verification Infrastructure 59
5.2.9 AspectJ Code Templates 59
5.2.10 Code Generation Process 61
5.2.11 Implementation 61
5.3 Summary 63

**Chapter 6 Conclusions** 64

**Appendix A** 65

**Appendix B** 66

**References** 69

# Abstract

Today, the analysis and design of software systems rely heavily on modeling. In model engineering, a number of models are produced during the software development process and MDA in particular. These models are usually expressed in UML and should be related in different way to each other. One of the most important relationships between them is that they should remain consistent during the life time. In object oriented world, the Object Constraint Language (OCL) is introduced to define constraints on UML models, namely, a means to define consistent models. The maintenance and verification of consistency should also be performed efficiently at runtime. Prototypes for such consistency verification infrastructure are to be designed and implemented. Putting these questions into the discussion and trying to find answers are the main topics to be addressed in this master thesis.

# Chapter 1

# Overview

Nowadays, object oriented modeling is mature enough and applied widely to provide a normalized way of designing software systems. In this engineering process, models can be viewed as an abstraction of artifacts and are expressed by using a suitable modeling language. However, this is usually not enough in practice. Additional consistency rules need be specified. For instance, imagine that we are designing a human resource management system for a company and have defined a new class named `Employee` with the attribute `age` of type integer. Any employee with negative age should be considered as an invalid entity. Actually the attribute `age` should be constrained to stay inside some range. If the system still accepts employees with invalid age, the consistency is broken. So besides by applying modeling language to define the models, one needs other approaches to specify the consistency rules of the whole system. In this way, the model is said to be consistent.

Here one can formulate the precise definition of consistent model. A model is said to be consistent when it conforms to the semantics of all the domains involved in the development process. Normally the consistency rules can be expressed in form of constraints enforced on models. In response to this requirement, the OMG (Object Management Group) specified OCL to define constraints on models.

However, the maintenance of consistency always becomes more and more uncontrollable as the system grows and expands. At runtime, an update may break predefined constraints and introduce inconsistency. In this case, we prefer to get informed about the situation and take proper actions to handle it. Manual consistency management is not applicable. Because it is error-prone in such a complicated engineering process. So prototypes for reliable consistency management system are expected.

# Chapter 2

# Case Study

In this chapter, we will go through a case study which also deals with model consistency problems. It is a good beginning to start from here. Because by learning the case study, we can gain some inspirations and a few useful tips to implement our own infrastructures.

## 2.1 Using ATL for Checking Models

This project deals with model consistency problems based on extending the standard object constraint language (OCL) and ATLAS transformation language (ATL) [3]. The end solution has been implemented as an Eclipse based plug-in.

Currently, some tools are available to check OCL invariants on UML models. However, there are very few tools able to do the same for any metamodel. This is quite inconvenient for the DSL (Domain Specific Language) approach to model engineering. The DSL approach promotes the definition of a large number of small domain-specific metamodels, rather than using a single and large metamodel. At the present time, few DSL tools are able to evaluate constraints on models. The objective of this project is to show how existing model transformation tools, such as ATL, can be used for this purpose.

A very simple DSL called CD for Class Diagrams is used as a motivating example. A metamodel of CD is given in figure 2.1. CD can be considered as a simplified subset of UML. Every element of a class diagram has a name. Classes can have `supertypes` and `StructuralFeatures`, which are `References` to other `Classes` or `Attributes`. `StructuralFeatures` have a multiplicity and are typed by a `Class` or a `DataType`. `Packages` are used to structure diagrams by grouping related elements.

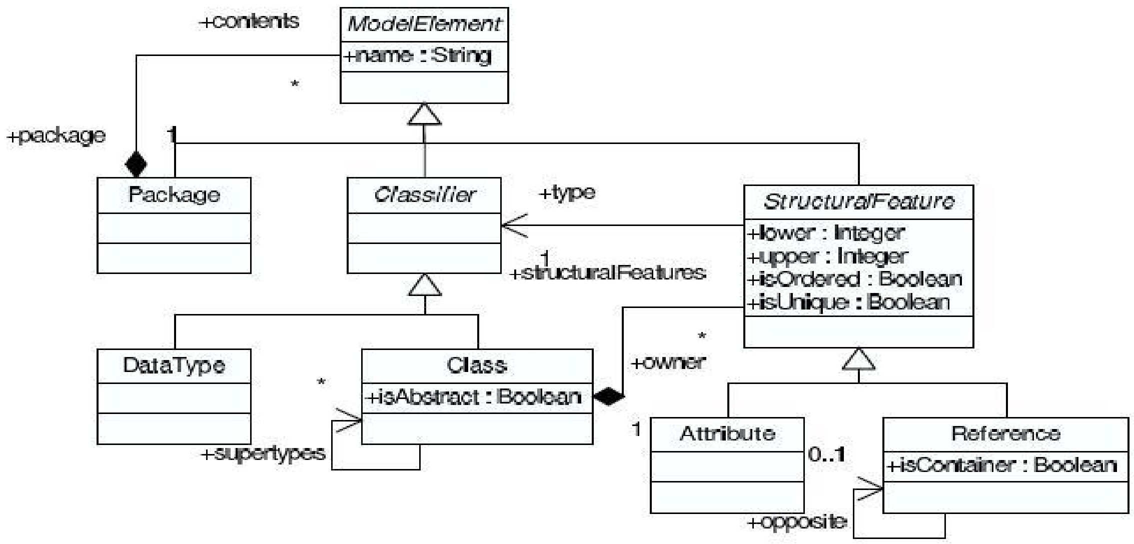


**Figure 2.1 Simple Class Diagram Metamodel**

The definition of the metamodel is however not complete. One can create models conforming to CD that are still not valid class diagrams. Additionally, constraints should be defined to complete the specification. To simplify the complete definition, just two constraints (C1) and (C2) are added.

(C1) `Classifier` names must be unique within a `Package`
(C2) The name of a `Classifier` should begin with an upper case letter

In order to be automatically verified, constraints must be written in an executable language. The well-known OCL solution is used here. Invariants must be verified at all times so that a model is supposed to be in a consistent state. An invariant is defined in the context of a metamodel type. It is composed of a Boolean expression, which must evaluate to true for every element of this type. It is however not always easy to understand the issue with the Boolean expression associated to the invariant. Sometimes it is even useful to specify constraints that **should** not be violated. When such constraints are not verified, the consistency of the model is not provably wrong. For example, if (C2) is violated, in this case, it should not be considered as an error and does not impact the structure of the metamodel at all. One can treat it just as a style convention. On the other hand, it is critical if (C1) is broken. This is quite similar to the way that compilers traditionally tag messages as error or warning. An error is fatal while a warning indicates a potential problem. The term “severity” is used here to indicate the failure degree of the problem.

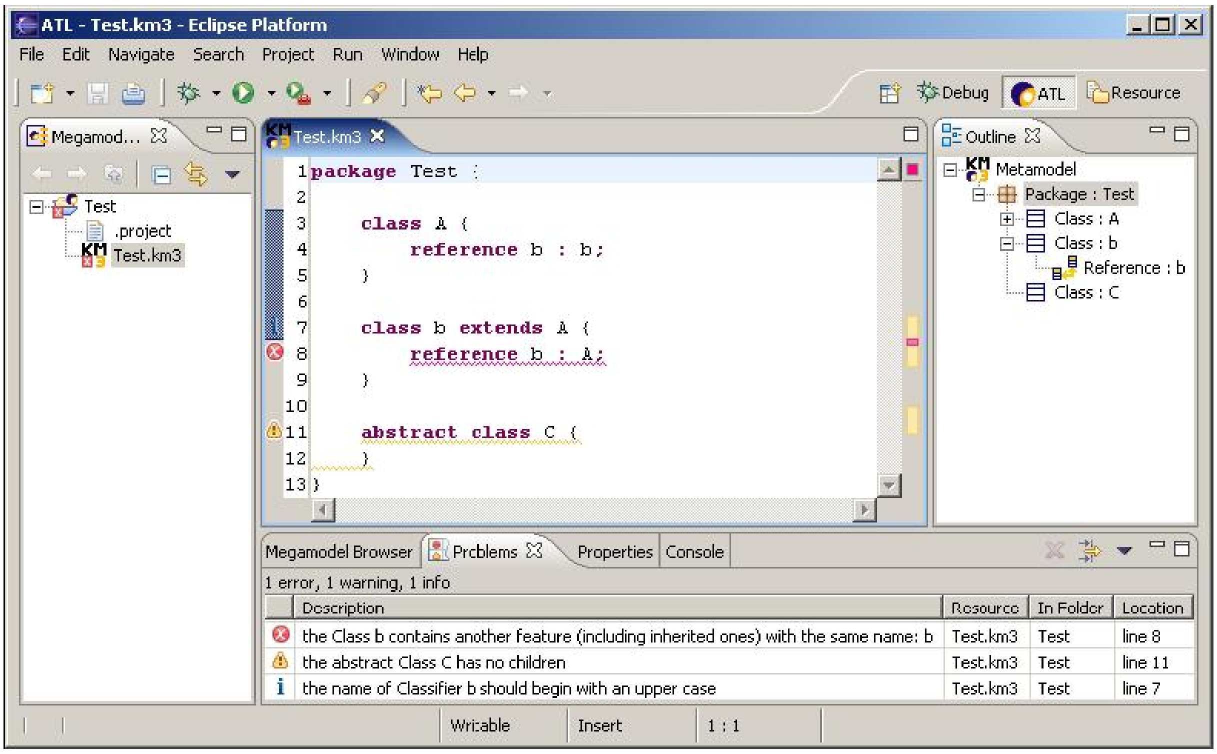
So far, the violation of an invariant can only be associated to the constraint itself, its severity and the violating model element. In order to better inform the user, “description” is used to state the problem in a human understandable manner. At the end, “location” is also reported which states the location of the problem in a computer readable format.

Generally speaking, people name the result of verification as a “diagnostic”. The simplest form of diagnostic is a Boolean. The true value means the model satisfies all the constraints whereas false means the model fails to satisfy all of them. If the diagnostic is represented as an integer, it can be used to encode the failure degree. The above mentioned extension is also a form of diagnostic. Since the diagnostic is a model, any transformation can be performed on it. Within an IDE, the diagnostic model can be mapped to the native representation (e.g. IMarkers in Eclipse). The problems then show up at the corresponding location in the editor and in the "Problems" view, as shown in figure 2.2.

ATL is a QVT-like (Query/View/Transformation) model transformation language. An execution engine for ATL is available as an open-source Eclipse plug-in. The ATL engine can be used to verify constraints. The algorithm to create a checking program from a set of constraints is the following: for each constraint, create an ATL transformation rule so that:

- The source pattern type of the rule is the context of the constraint.
- The guard of the rule contains the negation of the Boolean expression associated to the constraint.
- The target pattern of the rule specifies a single type: Problem, which is to be created on a match (i.e. on a violation of the invariant).
- It will be initialized using three bindings for: the severity of the invariant, a description of the issue and its location. Since the target element is initialized from OCL expressions navigating the source model, the implementation of a description as a constructed string is straightforward.

They implemented the verification of a slightly larger set of constraints on CD models in an Eclipse plug-in using the Eclipse Modeling Framework. They chose to represent class diagrams textually, using a simple syntax. In this case, the location of a Problem is therefore composed of a line and a column number. Figure 2.2 shows what is actually presented to the user when a class diagram contains problems.



```
 1 package Test ;
 2
 3     class A {
 4         reference b : b;
 5     }
 6
 7     class b extends A {
 8         reference b : A;
 9     }
10
11     abstract class C {
12     }
13 }
```

1 error, 1 warning, 1 info

| | Description | Resource | In Folder | Location |
|---|---|---|---|---|
| ⊗ | the Class b contains another feature (including inherited ones) with the same name: b | Test.km3 | Test | line 8 |
| ⚠ | the abstract Class C has no children | Test.km3 | Test | line 11 |
| i | the name of Classifier b should begin with an upper case | Test.km3 | Test | line 7 |

**Figure 2.2 Screenshot of the Eclipse Prototype [Jouault05]**

The source of an ATL transformation implementing this solution for (C1) is shown in figure 2.3.

```
-- (C1) Error: the name of a Classifier must
-- be unique within its package.
rule ClassifierNameUniqueInPackage {
  from
    i : CD!Classifier (
      i.package.contents->exists(e |
       (e <> i) and (e.name = i.name)))
  to
    o : Problem!Problem (
      severity <- Severity::error,
      description <- 'a Classifier of the same name ' +
          'already exists in the same package: ' + i.name,
      location <- i.location
    )
}
```

**Figure 2.3 ATL Transformation of (C1) with OCL Extensions [Jouault05]**

## 2.2 Summary

In this chapter we have reviewed a case study which improves the constraint checking in model engineering. In this approach, constraints are associated to additional information. This can be, for instance, severity, description and location. The diagnostic resulting from the verification of constraints is considered as a model, which can then be transformed into any representation. The ATL language can be used to express constraints on models and the ATL engine performs the verification in a batch-manner, i.e. not immediately upon an update is performed, but once the checks are run one after the other. Mechanisms to improve the efficiency of constraint checking are necessary.

# Chapter 3

# Introduction to ILOG Business Rules

Before we start to discuss our prototype, ILOG Business Rules [4] is first introduced, which is a key component used in our first consistency verification infrastructure. Generally speaking, business rules are precise statements, which describe the enterprise operations, constraints or definitions applied to business issues. These rules help the companies achieve better goals, operate more efficiently and facility the communication between different parties. It is also the same case when business rules are applied to information technology field. A typical product in this field is ILOG Business Rules, which allows business rules to be quickly changed and redeployed without changing the application code, thus reducing maintenance costs and extending the life of business applications.

## 3.1 The Challenge and Solution

Nowadays, more and more enterprise applications are developed with the increase in complexity and the pace of updating different aspects during the implementation time. By applying the traditional software architectures and mixing different aspects in the same code, one is not able to handle the complex and voluminous process as the system grows and expands. It is also difficult to map enterprise requirements to the implementation and then trace the implementation back to requirements. The ability to respond quickly to changing requirements and environmental conditions is rapidly becoming the key to solving this issue.

Besides that, forward-thinking IT architects consider present and as well as potential future requirements. Failing to take into account the potential future requirements may eventually lead to changing many parts of the system. On the other hand, over design results in a difficult-to-control, bloated system. In order to satisfy requirements in business domain, some architects make the business logic an integral part of the application code. The long term impact of developing your application in this way is that changing the business logic becomes impossible without having to redevelop large parts of the application logic. So the architects encourage their companies to realize the value of managing business rules as assets separate from data and code.

A new methodology is specifically targeted the management of this issue, by applying business rules. A business rule is a precise statement that describes, constrains, or controls some aspect of your business process. The strategic importance of business rules is to implement a company's objectives and accomplish the company's vision and goals. With business rules, one is able to distinguish the business logic from the application logic. In addition one adds agility and flexibility to the business application by allowing the business logic to be changed dynamically with very little overhead. Companies who have adopted business rule management need an enterprise-class Business Rule Management System (BRMS) to make the running of their business more practical.

## 3.2 ILOG Business Rule Studio Overview

We want to focus on learning business rule technology, instead of another development environment. With ILOG Business Rule Studio and ILOG JRules (J stands for Java here), one uses the familiar Eclipse IDE to embed rules into equally familiar Java/J2EE applications. ILOG Business Rule Studio is the first business rule authoring, testing and debugging environment for Eclipse. It enables the development of applications that evolve with changing business requirements. Powerful reasoning engines execute these business rules in real time to develop the best possible operating decisions, either as operator recommendations or as automated actions.

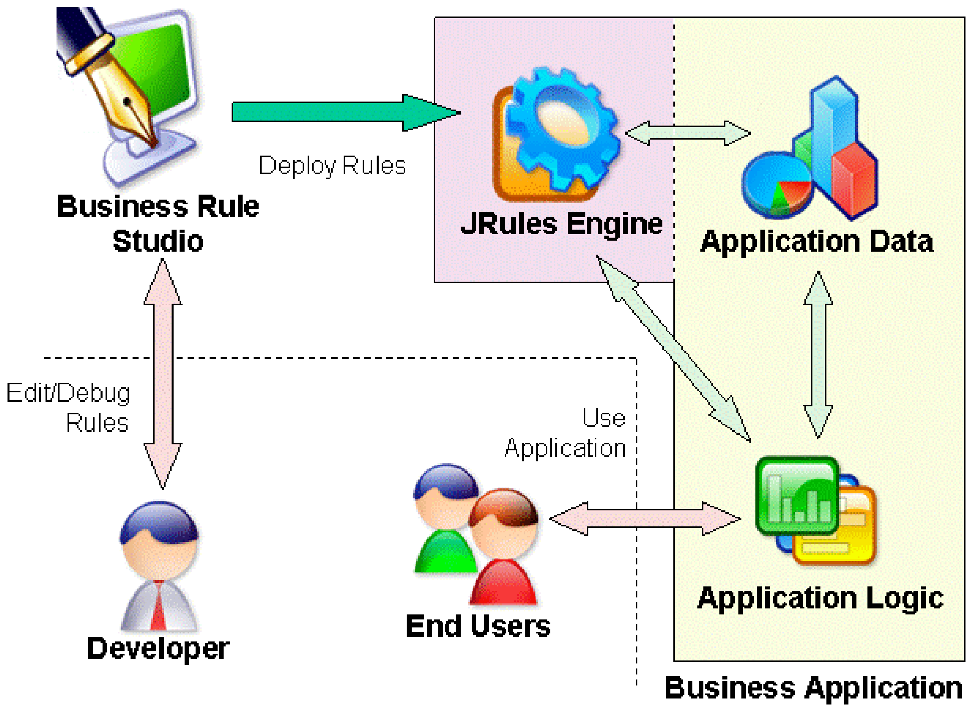


**Figure 3.1 Scenario of Using the Business Rule System [4]**

A typical scenario of using ILOG Business Rule Studio and the JRules rule engine is the following:

- The JRules rule engine executes business rules which implement the business logic. It handles application data through the application logic.

- The application logic is not aware of the business logic. It presents the business data to the users and calls the rule engine when the business logic is applied.
- The developer uses Business Rule Studio to edit, debug and test business rules. When new rules are ready, they can be deployed in the JRules rule engine to change the business logic on the fly.

## 3.3 Main Features of ILOG Business Rule Studio

ILOG Business Rule Studio offers a rich, developer-centric environment to author, test and deploy business rules. The developer can write and debug Java code and business rules from the same environment without interfering with each other. The Business Rule Studio is integrated with any Eclipse-based IDE including IBM's WebSphere Studio Application Developer.

With Business Rule Studio's embedded rule engine, one can test and debug rules locally without deploying them to an external rule engine. Business Rule Studio provides a new Eclipse project type called rule project. It stores resources like rules, packages and ruleflows. Two perspectives support rule editing and debugging (similar to Eclipse Java editing and debugging). Text editor is used for rule authoring. The Business Rule Studio text editor includes all the features of Eclipse's Java code editors. It supports basic editing capabilities. Advanced capabilities include code completion, syntax checking, auto-indentation and syntax-coloring. Source code control integration like the most popular CVS is also provided to facility the working process. Last but not the least, the engine API integrates and controls the business rule engine using an extensive and comprehensive Java library, which is also delivered.

## 3.4 Concepts

### Rule Engine

The basic functionality of the rule engine can be described as follows; it:

- can read its rules dynamically (at runtime)
- reasons on objects it knows
- can keep track of changes to the objects it knows
- can invoke the firing of rules

The rule engine is service oriented, meaning that it responds to explicit invocations. The rule engine enables business behavior to be managed separately from the core, code-based architecture of an application, which also means that it can evolve more rapidly than the code. The rule engine in ILOG Business Rules operates with rules expressed in the ILOG Rule Language (IRL), geared specifically for the expression of business rules directly translatable to rule engine execution.

The rule engine is provided as a set of class libraries, which enable it to be integrated into any Java application. Architecture is not imposed, which allows the engine to be integrated into an application without constraining the application or technical architecture. The rule engine can be deployed in this way on the Java 2 Standard Edition (J2SE) platform or on the Java 2 Enterprise Edition (J2EE) platform. The rule engine must be provided with rules, usually in the form of a ruleset (.irl) file, and a set of classes. When the ruleset has been passed to the rule engine, it is then possible to interact with the rule engine object using the API. When application objects are inserted into the rule engine object, two things happen. First, references to the native Java application objects are added to the rule engine. These references enable the rule engine to monitor the application objects. Second, the conditions of all rules in the ruleset are evaluated. If the conditions of a rule are met, the rule is declared eligible to execute, or fire.

In ILOG JRules, the rule engine is an instance of the `IlrContext` class, so the rule engine is simply a Java object. This class contains all the methods required to control the rule engine. An `IlrContext` is always attached to an `IlrRuleset`. The constructor for an `IlrContext` may take the form `IlrContext(IlrRuleset)` or `IlrContext()`. If the `IlrContext` object is created without a ruleset passed as an argument, it will create its own ruleset. The `IlrRuleset` class is used for the management of the rules and `IlrContext` is used for the execution of the rules. An `IlrContext` associates a ruleset with application objects and implements the rule engine that controls the relationship between the rule part of the application and the application data. An application can contain several rule engine objects. These can be direct instances of the `IlrContext` class or instances of derived classes. A rule in ILOG JRules is represented as an object, which is an instance of the `IlrRule` class. The `IlrRule` class is responsible for the management of rules and is always attached to a rule engine object.

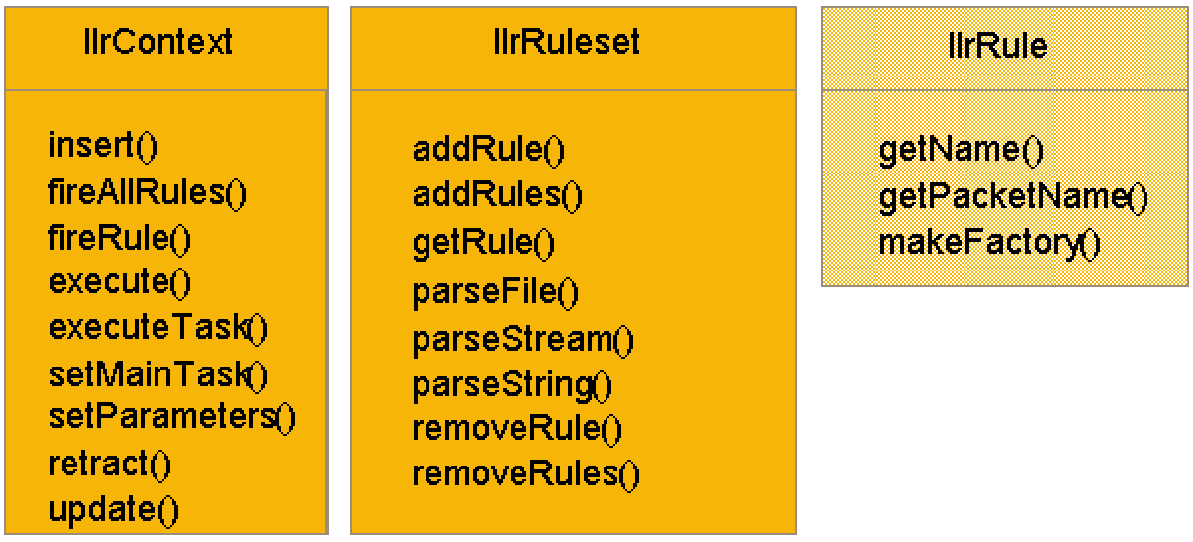


**Figure 3.2 APIs of Business Rules**

A rule can be executed (fired) by the method `IlrContext.fireRule()`. One can use the methods of the class `IlrRuleset` to dynamically add/remove rules to/from the ruleset, which is the file that contains all defined rules. One can add a rule or several rules to the ruleset by using the methods `addRule` and `addRules` of the class `IlrRuleset`. One can remove a rule or several rules from the ruleset by using the methods `removeRule` and `removeRules` of the class `IlrRuleset`. Rules may be dynamically added, modified or removed from the rule engine, enumerated and inspected, packaged into sets, and executed individually or as sets.

**Rules**

Business rules are implemented in ILOG Business Rules by expressing them as execution rules in the ILOG Rule Language (IRL), where they can be executed by the ILOG rule engine. Like other programming languages, IRL has a number of keywords, or reserved words. A rule is defined by means of the rule keyword.

```
rule ruleName {
   priority = priorityValue;
   property propertyName = value; ...
   when { conditions ... }
   then { actions ... }
   else { actions ... }
}
```

**Figure 3.3 Skeleton of Rule**

An execution rule has an IRL structure composed of a header, a condition part, and an action part.

- The **header** part defines the name of the rule with the `rule` keyword statement, the properties, and its priority.
- The **condition** part, which begins with the keyword `when`, is also referred to as the left-hand side (LHS) of the rule. It utilizes the object-oriented structure of Java to carry out pattern matching on objects. This pattern matching binds variables to objects and field values. Rule conditions are also used to test field values. This provides a filtering mechanism for objects.
- The **action** part, composed of one or two parts, is also referred to as the right-hand side (RHS) of the rule. The first part begins with the keyword `then`. The optional second part begins with the keyword `else`. The action part of the rule specifies actions to be taken if the rule is executed.

Figure 3.4 shows a typical example:

```
rule FindFilm {
   priority = 1;
   when {
      Film(language == English; ProductionYear > 1980 & < 1990; ?t:title);
      Cinema(location == Paris; filmTitle == ?t;
             showingTime > 13.00 & < 16.00; ?c:name);
   }
   then {
      System.out.println("The film: " + ?t + " is showing at the cinema: "
                          + ?c);
   }
};
```

**Figure 3.4 Example Rule FindFilm**

After the `rule` keyword in the first line we find the name of the rule, FindFilm. The second line expresses its priority by means of the `priority` keyword. In this example there are two conditions. These conditions use the classes `Film` and `Cinema`. In ILOG JRules, a rule can have any number of conditions, but a rule without any conditions is not allowed. The `then` keyword marks the end of the conditions and the start of the actions. In the example, there is only one action: an ILOG JRules instruction that prints the title of a film showing at a cinema in Paris.

**Execution Object Model**

The eXecution Object Model (XOM) provides classes for a rule engine that can be used by rules written in the ILOG Rule Language (IRL). The XOM can be accessed through classes of the Factory API. This package provides reflective requests on a ruleset and supports persistence and sharing through an object representation of rules.

**ILOG Rule Language**

The ILOG Rule Language (IRL) is the executable rule language for ILOG Business Rules. All business rules are translated into this language before parsing by the rule engine. IRL provides a rich set of constructs which includes collections, support for relations between objects, and temporal reasoning. The ILOG JRules rule engine uses various optimization techniques to improve efficiency in rule processing. An IRL program can be integrated into multithreaded applications and deployed in environments including J2EE. It also provides support for XML-based reporting. One of the important features of ILOG Business Rules is its Business Rule Language support which uses a framework to ensure the translation from the business rule language to the execution rule language: IRL.

**Algorithms**

One has a choice of algorithms used during rule engine operation. An algorithm called the Rete algorithm operates efficiently in the domain of pattern matching. The sequential algorithm is designed for speed of execution.

The Rete network is used by the rule engine to minimize the number of rules and conditions that need to be evaluated, compute which rules should be executed, and identify in which order these rules should be fired. The Rete network includes a working memory and an agenda for containing and manipulating application objects.

The ILOG JRules sequential algorithm utilizes a dynamic rule compilation that can significantly improve the speed of rule processing. The performance of an engine will improve using the sequential algorithm if it is provided with a large ruleset made of basic but test-intensive rules with static priorities.

**Working memory**

Under the Rete algorithm, each rule engine in ILOG JRules is paired with a working memory. The working memory contains all the objects that need be treated by the rules. Objects can be added to, updated in and removed from the working memory. In other words, the engine is aware of the objects that are in the working memory and those which are linked to them. If an object is not accessible from the working memory, it cannot be used by the rule engine.

**Agenda**

The agenda is where ILOG JRules stores the rules whose patterns are all matched. Any rule that enters the agenda is said to be instantiated. The agenda stores rule instances that are eligible to be fired.

## 3.5 Summary

In this chapter we learned what Business Rules are and what advantages they bring to us compared to the other traditional methodologies.    Then, ILOG Business Rule Studio is introduced as the first business rule authoring, testing and debugging environment for Eclipse. Features and concepts are listed for the reader to gain an overview of the ILOG Business Rule Studio.

# Chapter 4

# Prototype Based on Business Rules

In the previous chapter we learned the concept of Business Rules and got familiar with the structure of the rule and rule engine. Due to the separation of the business logic and application code, we can treat the consistency requirements as part of the business logic in our case. And with the support of the business rule engine and provided APIs, we can conceive our first prototype based on Business Rules and manage the consistency problems efficiently. The infrastructure is implemented as an extension of the Eclipse plug-in Octopus [1], which supports the use of both UML and OCL for modeling.

## 4.1 Advantages of Applying Business Rules

Model engineering is based on the definition of models. Besides models, additional constraints are necessary to be specified to improve consistency management. In response to these requirements, the OMG (Object Management Group) specified OCL to express constraints on models. OCL usually returns a Boolean value to indicate whether the invariant is violated or not. However, it is not always desirable and easy to understand the issue with only the Boolean expression associated to the constraint. One needs another mechanism or extension to detect the inconsistency and take the corresponding action.

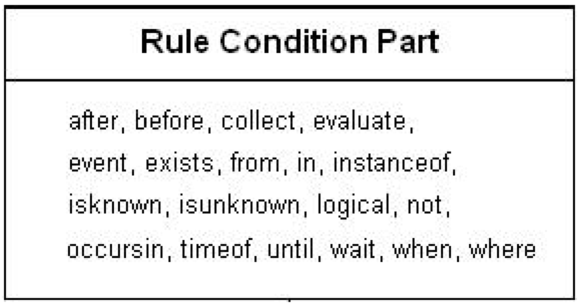
With the help of Business Rules, one is able to extend the ability of OCL. Not only can inconsistency be detected as usual, but also proper actions can be carried out if the consistency is broken. The consistency requirements can be specified in the condition part of the Business Rules and the problem handling can be done in the action part. How to handle the issue if the consistency is violated is left to the designer. The infrastructure can inform the user about the inconsistency by stating the problem in a human understandable manner. It is also desirable if the system is able to automatically repair the inconsistency in some cases. All these can be specified by Business Rules.

## 4.2 Transformation from OCL invariants to Business Rules

Since OCL is still the standard and most famous language used to define model constraints, the first step for us to implement our infrastructure is to define a mapping from OCL invariants to ILOG rule language. One may ask how well OCL invariants can be mapped to ILOG rule language due to the different language semantics and structures, and whether the mapping can be carried out without any loss. Before answering those questions, let us take a deep look at the structure of the ILOG rule language again.

### 4.2.1 ILOG Rule Language Structure

A business rule has the following form: IF conditions THEN actions. The keyword `when` is used to specify the condition part of a rule. The condition part of a rule is composed of a set of conditions, or patterns, that refer to Java objects. Each pattern is matched, if possible, with one or more application objects. More precisely, a pattern comprises tests that are applied to each object in the working memory, and an object is said to match the pattern when it passes these tests successfully. The pattern is tested by evaluating public attributes and/or public methods of Java objects in the working memory. The ILOG rule language also offers a few keywords to facilitate the pattern matching test. They are listed in figure 4.1.



| Rule Condition Part |
|---|
| after, before, collect, evaluate,<br>event, exists, from, in, instanceof,<br>isknown, isunknown, logical, not,<br>occursin, timeof, until, wait, when, where |

**Figure 4.1 Keywords in Rule Condition Part**

Here only some of those keywords are of interest to our case.

- **collect**

[`?variable`:] **collect** [(`expression`)] `collectionTarget`
[where (`collectionTest`$_1$ `...` `collectionTest`$_n$)];

The `collect` statement is used in the condition part of a rule to create a collection object. The collection object stores instances of the class `collectionTarget` that match the condition. This condition may contain tests on the class fields. The collection object may be bound to a variable for the scope of the rule. The `collect` statement may contain a list of tests on the collection object in the `where` part of the statement.

- **evaluate**

**evaluate** (`expression`);

The evaluate statement is used in the condition part of a rule to test objects of the working memory. An evaluate statement must have a simple condition preceding it that binds a variable to an object or a value. Any such variable bound to an object or a value may be tested. Note that the statements not, exists, and collect are not simple conditions. An evaluate statement is true if all the tests carried out in the expression are true. The expression may be multiple tests enclosed by braces ({}).

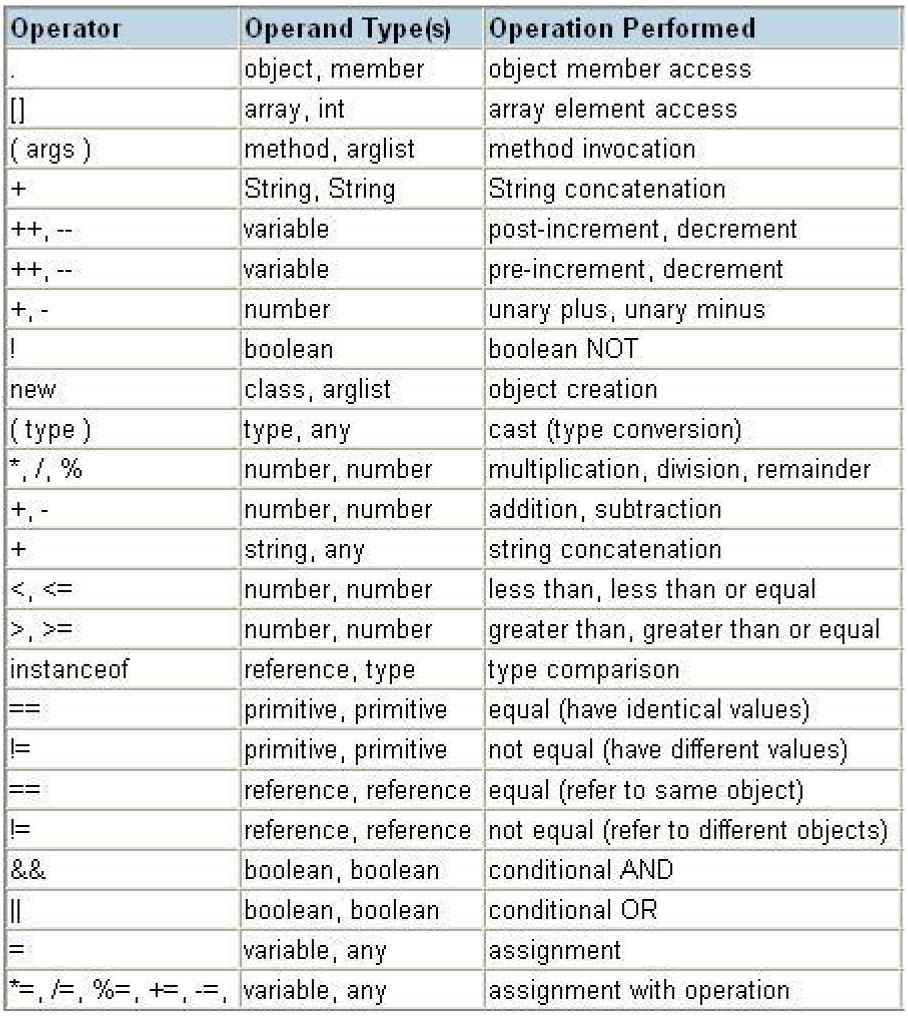
In the case where the condition part of a rule ends with an evaluate statement, the action part can have an else part, executed if the evaluate statement returns false. If it returns true, the then part is executed.

- **where**

[?variable:] collect [(expression)] collectionTarget
[**where** (collectionTest$_1$ ... collectionTest$_n$)];

The collect statement is used in the condition part of a rule to create a collection object. The where part of the statement may contain tests that the collection object must fulfill. It may be left empty.

Besides those keywords, ILOG rule language offers also operators, which are a subset of the operators provided in the Java programming language. Figure 4.2 displays all those operators.



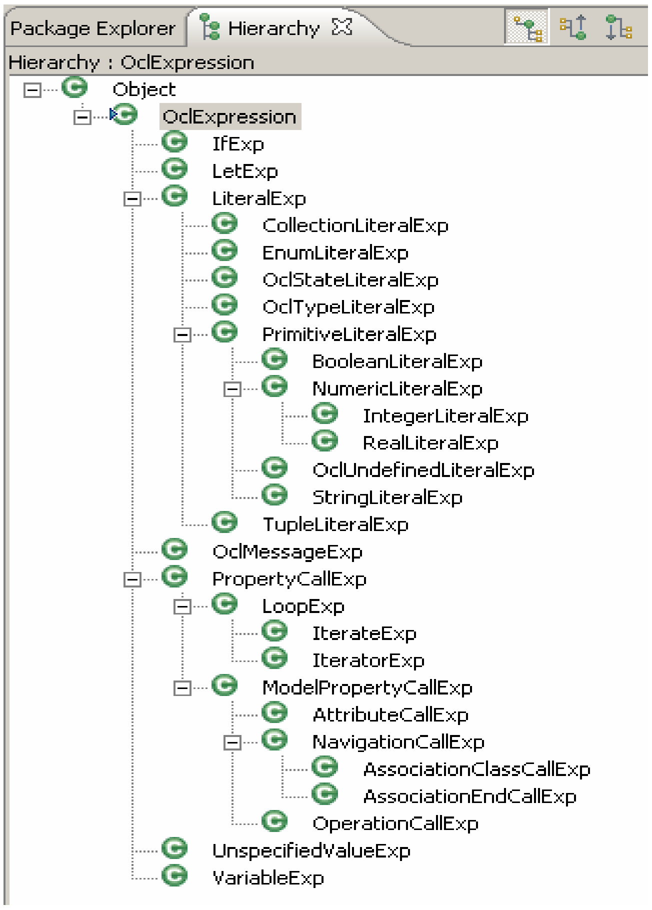
| Operator | Operand Type(s) | Operation Performed |
|---|---|---|
| . | object, member | object member access |
| [] | array, int | array element access |
| ( args ) | method, arglist | method invocation |
| + | String, String | String concatenation |
| ++, -- | variable | post-increment, decrement |
| ++, -- | variable | pre-increment, decrement |
| +, - | number | unary plus, unary minus |
| ! | boolean | boolean NOT |
| new | class, arglist | object creation |
| ( type ) | type, any | cast (type conversion) |
| *, /, % | number, number | multiplication, division, remainder |
| +, - | number, number | addition, subtraction |
| + | string, any | string concatenation |
| <, <= | number, number | less than, less than or equal |
| >, >= | number, number | greater than, greater than or equal |
| instanceof | reference, type | type comparison |
| == | primitive, primitive | equal (have identical values) |
| != | primitive, primitive | not equal (have different values) |
| == | reference, reference | equal (refer to same object) |
| != | reference, reference | not equal (refer to different objects) |
| && | boolean, boolean | conditional AND |
| \|\| | boolean, boolean | conditional OR |
| = | variable, any | assignment |
| *=, /=, %=, +=, -=, | variable, any | assignment with operation |

**Figure 4.2 Operators in ILOG Rule Language**

As one can see, the structure of the ILOG rule language is heavily based on Java language. Given that Octopus plug-in already provides the functionality of transforming OCL to Java code, one can take advantage of this available feature. In order to continue our work, one needs to know how Octopus transforms OCL invariants to Java code first.

## 4.2.2 Octopus Java Code Generation

Octopus defines its own OCL metamodel. Figure 4.3 shows the hierarchy of Octopus OCL metamodel in a listed view. Every OCL expression is of type `OclExpression`. The invariant can be displayed as an AST (Abstract Syntax Tree). It contains different kinds of sub expressions. During the code generation process, invariant is analyzed and split into pieces. The corresponding Java code is generated for each sub expression. Depending on the complexity of each sub expression, simply Java clause or complete Java method may be created. For example, `VariableExp` results in only a simple Java attribute access, which can be expressed by stating the variable name. On the other hand, `IterateExp` causes iteration over a Set, which is much more complicated than the first case. Octopus handles such cases by generating private methods which return the intermediate results. For instance, a private method which returns the iteration result will be created for `IterateExp`. Those intermediate results may be used by the public invariant checking method or other intermediate private method resulted from the parent node. Finally, a public method used for checking invariant is generated, which is expected to be invoked by the end user.



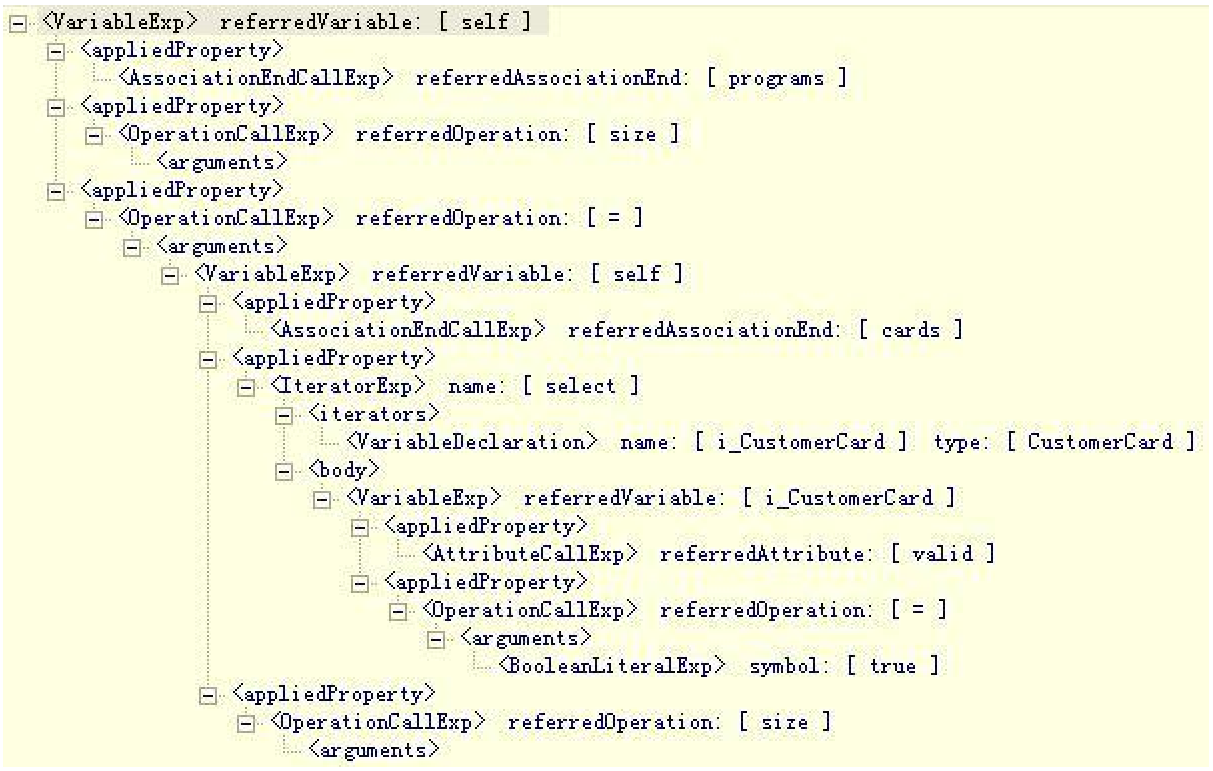
```
Hierarchy : OclExpression
Object
    OclExpression
        IfExp
        LetExp
        LiteralExp
            CollectionLiteralExp
            EnumLiteralExp
            OclStateLiteralExp
            OclTypeLiteralExp
            PrimitiveLiteralExp
                BooleanLiteralExp
                NumericLiteralExp
                    IntegerLiteralExp
                    RealLiteralExp
                OclUndefinedLiteralExp
                StringLiteralExp
            TupleLiteralExp
        OclMessageExp
        PropertyCallExp
            LoopExp
                IterateExp
                IteratorExp
            ModelPropertyCallExp
                AttributeCallExp
                NavigationCallExp
                    AssociationClassCallExp
                    AssociationEndCallExp
                OperationCallExp
        UnspecifiedValueExp
        VariableExp
```

**Figure 4.3 Metamodel of OCL in Octopus**

In order to better understand the generation process, we take the following OCL invariant from the famous Loyal and Royal project (UML diagram given in appendix A) as an example:

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

This invariant states the consistency rule that in class `Customer` the size of associated `programs` should be equal to the size of those associated `cards`, whose attributes `valid` are set to `true`. The AST of the above OCL invariant is displayed in figure 4.4.



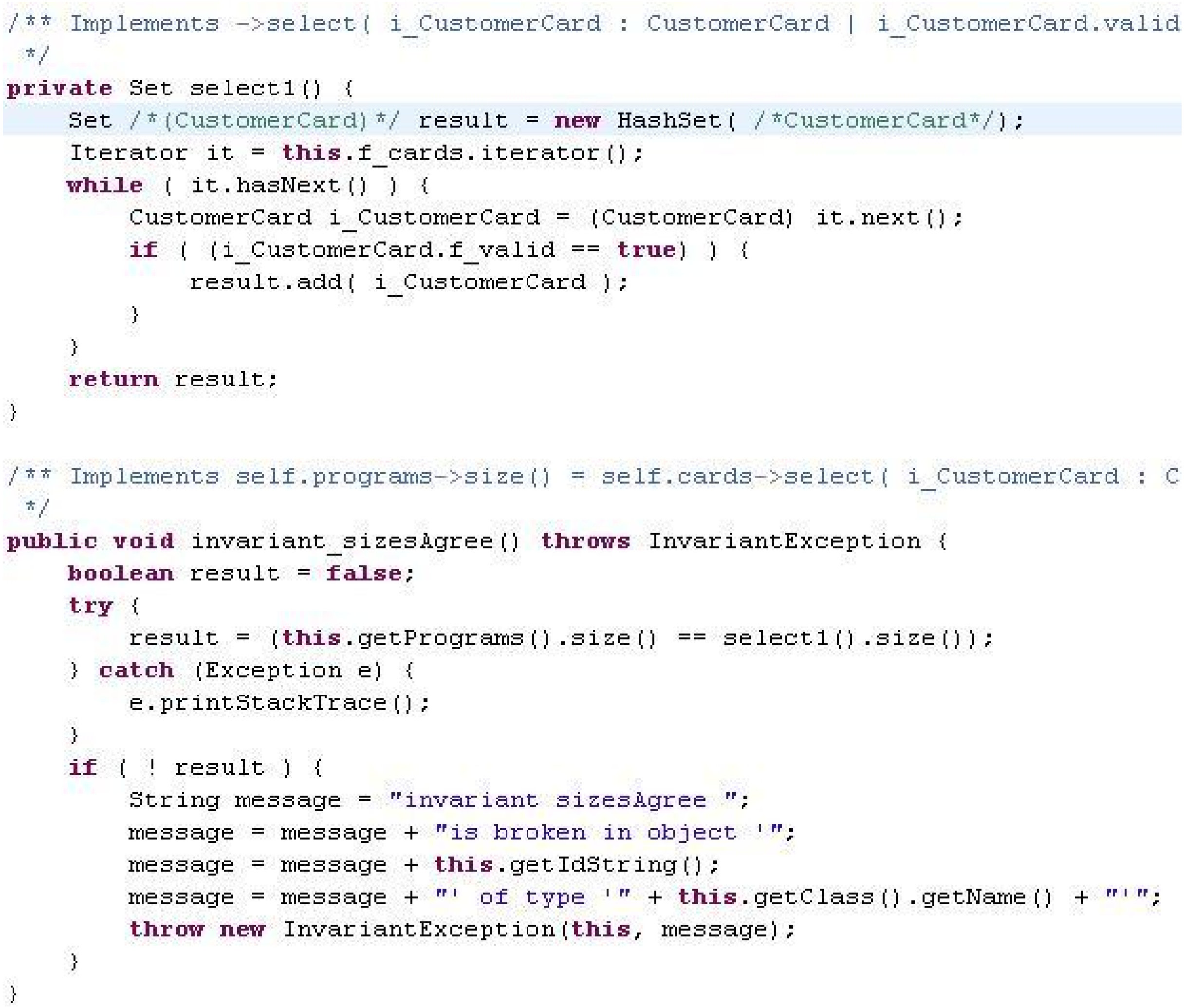
```
⊟ <VariableExp>  referredVariable: [ self ]
   ⊟ <appliedProperty>
        <AssociationEndCallExp>  referredAssociationEnd: [ programs ]
   ⊟ <appliedProperty>
      ⊟ <OperationCallExp>  referredOperation: [ size ]
           <arguments>
   ⊟ <appliedProperty>
      ⊟ <OperationCallExp>  referredOperation: [ = ]
         ⊟ <arguments>
            ⊟ <VariableExp>  referredVariable: [ self ]
               ⊟ <appliedProperty>
                    <AssociationEndCallExp>  referredAssociationEnd: [ cards ]
               ⊟ <appliedProperty>
                  ⊟ <IteratorExp>  name: [ select ]
                     ⊟ <iterators>
                          <VariableDeclaration>  name: [ i_CustomerCard ]  type: [ CustomerCard ]
                     ⊟ <body>
                        ⊟ <VariableExp>  referredVariable: [ i_CustomerCard ]
                           ⊟ <appliedProperty>
                                <AttributeCallExp>  referredAttribute: [ valid ]
                           ⊟ <appliedProperty>
                              ⊟ <OperationCallExp>  referredOperation: [ = ]
                                 ⊟ <arguments>
                                      <BooleanLiteralExp>  symbol: [ true ]
               ⊟ <appliedProperty>
                  ⊟ <OperationCallExp>  referredOperation: [ size ]
                       <arguments>
```

**Figure 4.4 AST View of Invariant `sizesAgree`**

The AST view shows that this invariant consists of different OCL sub expressions, e.g., `VariableExps`, `AssociationEndCallExps`, `OperationCallExps`, `AttributeCallExps` and `BooleanLiteralExp`. The analysis sequence is top down as shown in the AST view. That means if the example OCL invariant is given, one starts from the root of the AST, in this case, `VariableExp self`. By invoking the `getAppliedProperty`, the next sub expression is returned, namely, `AssociationEndCallExp programs`. Following the same way, all the sub expressions will be traversed and handled. Each kind of OCL expression has its own Java code pattern. For example, the generated code for `AssociationEndCallExp` returns just the corresponding `getter` method of that association end. In our case, `this.getPrograms()` is the generated Java code for `AssociationEndCallExp programs`. Complicated case is for example `IteratorExp select`, it deals with a selection over a Set, and the selection result is again a Set. In order to make the code more readable, Octopus creates a private method to calculate the selection over that Set. And this private method will be invoked in the public invariant checking method. The name of the invariant checking method begins with `invariant_` and is followed by the same string as the invariant name if the name is defined in OCL. If not given, the Java method name begins with `invariant_` and is followed by a number, which is incremented automatically for each unnamed invariant. Intermediate private methods follow the similar naming convention, only the name begins with what it actually performs.

In the final invariant checking method, a `boolean` variable is defined. If the invariant is broken, the `boolean` variable is set to false and an `InvariantException` is thrown.

Figure 4.5 shows the generated Java code by Octopus:



```
/** Implements ->select( i_CustomerCard : CustomerCard | i_CustomerCard.valid
 */
private Set select1() {
    Set /*(CustomerCard)*/ result = new HashSet( /*CustomerCard*/);
    Iterator it = this.f_cards.iterator();
    while ( it.hasNext() ) {
        CustomerCard i_CustomerCard = (CustomerCard) it.next();
        if ( (i_CustomerCard.f_valid == true) ) {
            result.add( i_CustomerCard );
        }
    }
    return result;
}

/** Implements self.programs->size() = self.cards->select( i_CustomerCard : C
 */
public void invariant_sizesAgree() throws InvariantException {
    boolean result = false;
    try {
        result = (this.getPrograms().size() == select1().size());
    } catch (Exception e) {
        e.printStackTrace();
    }
    if ( ! result ) {
        String message = "invariant sizesAgree ";
        message = message + "is broken in object '";
        message = message + this.getIdString();
        message = message + "' of type '" + this.getClass().getName() + "'";
        throw new InvariantException(this, message);
    }
}
```

**Figure 4.5 Generated Java Code of Invariant `sizeAgree`**

One may ask why not just simply evaluate the generated public invariant checking methods in the condition part of the ILOG rule to test whether the consistency is broken or not. The key is that public variables should be accessed instead of method calls in IRL condition part as much as possible in order to improve the performance. Otherwise, the rule engine needs to reevaluate that method whenever we update the object in the working memory, and will not detect on its own that a rule instance has to be added to the agenda.

## 4.2.3 Mapping from OCL to IRL

The mapping should start from the OCL metamodels, which are exactly what the figure 4.3 displays. Next step, we will explain the mapping for each of these metamodels. Sometimes, a direct mapping is not possible. In this case, the generated Java method should be invoked in the IRL condition part instead of formulating the counterpart in the business rules. Although performing a direct method invocation instead of attribute access may lose some performance at run time, it guarantees that no information is missing after this mapping happens.

- *IfExp*
  It has the `If – then – else` pattern. `If` corresponds to `when` in condition part of IRL language. The other two correspond to the action part. Depending on whether the condition part is fulfilled, the proper action is carried out.

- *LetExp*
  OCL: *let definedVariable : type = defExp*
  IRL: Not possible to define new variable in IRL condition part.
  Direct method invocation in IRL condition part.

- *LiteralExp*

  1. *CollectionLiteralExp*
     OCL: *Set {Integer* | Float* | String*}*
     IRL: Not possible to collect primitive types as a Set in IRL condition part.
     Direct method invocation in IRL condition part.

  2. *EnumLiteralExp*
     OCL: *Type::EmunName* (e.g. *Gender::male*)
     IRL: *JavaType.EnumNameToUpperCase* (e.g. *Gender.MALE*)

  3. *OCLStateLiteralExp*
     No change.

  4. *OCLTypeLiteralExp*
     No change.

  5. *BooleanLiteralExp*
     No change.

  6. *IntegerLiteralExp*
     No change.

  7. *RealLiteralExp*
     OCL: *RealLiteralExp*
     IRL: *(float) RealLiteralExp*

  8. *OclUndefinedLiteralExp*
     OCL: *OclUndefinedLiteralExp*
     IRL: *null*

9. *StringLiteralExp*
No change.

10. *TupleLiteralExp*
No change.

- *OclMessageExp*
Not handled by Octupus.

- *VariableExp*
OCL: *self*
IRL: *?self*

- *AttributeCallExp*
OCL: *attributeName* (e.g. *name*)
IRL: *javaFieldName* (e.g. *f_name*)

- *AssociationClassCallExp*
OCL: *AssociationClassName* (e.g. *Membership*)
IRL: *javaFieldName* (e.g. *f_membership*)

- *AssociationEndCallExp*
OCL: *AssociationEndName* (e.g. *cards*)
IRL: *javaFieldName* (e.g. *f_cards*)

- *OperationCallExp*

OCL: *allInstances*
IRL: *className.allInstances()*

OCL: *Class operation*
IRL: *className.opName.(args)*

CollectionOper:

OCL: *source.**count**(obj)*
IRL: *?countCollection: **collect** objType(source.contains(this))*
*…?countCollection.size()…*

OCL: *source.**excludes**(obj)*
IRL: *!source.contains(obj)*

OCL: *source.**excludesAll**(coll)*
IRL: *Stdlib.excludesAll(source, coll)*

OCL: *source.**includes**(obj)*
IRL: *source.contains(obj)*

OCL: *source.**includesAll**(coll)*
IRL: *source.containsAll(coll)*

OCL: *source.**isEmpty**()*
IRL: *source.isEmpty()*

OCL: *source.**notEmpty**()*
IRL: *!source.isEmpty()*

OCL: *source.**size**()*
IRL: *source.size()*

OCL:*source.**sum**()*
IRL: It needs iterator to travel through every element to sum them up and is not supported directly.
Direct method invocation in IRL condition part.

OCL: *source = arg*
IRL: *Stdlib.( setEquals | bagEquals | sequenceEquals | orderedsetEquals).(source, arg)*

OCL: *source <> arg*
IRL:*! Stdlib.( setEquals | bagEquals | sequenceEquals | orderedsetEquals).(source, arg)*

OCL: *source - coll*
IRL: The sequence can not be guaranteed.
Direct method invocation in IRL condition part.

OCL: *source.**append**(obj)*
IRL: The sequence can not be guaranteed.
Direct method invocation in IRL condition part.

OCL: *source.**at**(Int)*
IRL: *source.get(int – 1)*

OCL: *source.**excluding**(obj)*
IRL: *?excludingColl **collect** objType(source.contains(this); !obj.equals(this))*

OCL: *source.**first**()*
IRL: *source.get(0)*

OCL: *source.**flatten**()*
IRL: *Stdlib.( setFlatten | bagFlatten | bagFlatten | orderedsetFlatten ).(source)*

OCL: *source.**including**(obj)*
IRL: *?excludingColl **collect** objType(source.contains(this) || obj.equals(this))*

OCL: *source. **indexOf**(obj)*
IRL: *source.indexOf(obj) + 1*

OCL: *source.**insertAt**(int, obj)*
IRL: *Stblib.insertAt(source, int – 1, obj)*

OCL: *source.**intersection**(set)*
IRL: *?intersectionColl **collect** objType(source.contains(this); set.contains(this))*

OCL: *source.**last**()*
IRL: *source.get(source.size() – 1)*

OCL: *source.**prepend**(obj)*
IRL: The sequence can not be guaranteed.
Direct method invocation in IRL condition part.

OCL: *source.**subOrderedSet**(int, int)*
IRL: *source.sublist(int, int)*

OCL: *source.**subsequence**(int, int)*
IRL: *source.sublist(int, int)*

OCL: *source.**symmetricDifference**(set)*
IRL: *? symmetricDiffColl **collect** objType(!source.contains(this); set.contains(this))*

OCL: *source.**union**(sequence)*
Depending on the source type, if the type is `OrderedSet`, no duplicated objects are allowed in the result set.
IRL: *? unionColl **collect** objType(source.contains(this) || sequence.contains(this))*
Otherwise, duplicated case is considered, not supported in IRL.
Direct method invocation in IRL condition part.

OCL: *source.**asBag**()*
IRL: *Stdlib.collectionAsBag(source)*

OCL: *source.**asSequence** ()*
IRL: *Stdlib.collectionAsSequence (source)*

OCL: *source.**asOrderedSet** ()*
IRL: *Stdlib.collectionAsOrderedSet (source)*

OCL: *source.**asSet** ()*
IRL: *Stdlib.collectionAsSet (source)*

OCL: *source.**oclIsUndefined**()*
IRL: *source == null*

- *LoopExp -> iteratorExp -> Exists*
  OCL: *source->exists(expr)*
  IRL: *?existsColl: **collect** objType(source.contains(this); expr)*
  ***evaluate**(... (?existsColl.size() > 0) ...)*
  Source code in Octopus:
  *package com.klasse.octopus.codegen.umlToIRL.expgenerators.creators;*
  *LoopExpIRLCreator*
  *createExists*

- *LoopExp -> iteratorExp -> ForAll*
  OCL: *source->forAll(expr)*
  IRL: *? forAllColl:* **collect** *objType(source.contains(this); !expr)*
  **evaluate***(… (?forAllColl.size() == 0) …)*
  Source code in Octopus:
  *package com.klasse.octopus.codegen.umlToIRL.expgenerators.creators;*
  *LoopExpIRLCreator*
  *createForAll*

- *LoopExp -> iteratorExp -> IsUnique*
  OCL: *source->isUnique(expr)*
  IRL: not supported in IRL.
  Direct method invocation in IRL condition part.

- *LoopExp -> iteratorExp -> any*
  OCL: *source->any(expr)*
  IRL: *? any: objType(source.contains(this); expr)*
  Drawback: The rule could be fired more than once.

- *LoopExp -> iteratorExp -> one*
  OCL: *source->one(expr)*
  IRL: *? oneColl:* **collect** *objType(source.contains(this); expr)*
  **evaluate***(…(?oneColl.size() == 1)…)*

- *LoopExp -> iteratorExp -> collect*
  OCL: *source.collect(expr)*
  Depending on the argType, if the type is CollectionType
  IRL: Direct method invocation in IRL condition part.
  Otherwise
  IRL: *? coll:* **collect** *objType(this.assoEnd.equals(source); expr != null)*

- *LoopExp -> iteratorExp -> collectNested*
  OCL: *source.collectNested(expr)*
  IRL: *? collNested:* **collect** *objType(this.assoEnd.equals(source); expr != null)*

- *LoopExp -> iteratorExp -> select*
  OCL: *source.select(expr)*
  IRL: *?selectColl:* **collect** *objType(source.contains(this); expr)*

- *LoopExp -> iteratorExp -> reject*
  OCL: *source.reject(expr)*
  IRL: *? rejectColl:* **collect** *objType(source.contains(this); !expr)*

- *LoopExp -> iteratorExp -> sortedBy*
  OCL: *source.sortedBy(expr)*
  IRL: Not possible to sort objects in a Set in IRL condition part.
  Direct method invocation in IRL condition part.

| *LoopExp -> iterateExp*<br>OCL: *source->iterate(variable declariation, expression)*<br>IRL: Not possible to define new variable in IRL condition part.<br>Direct method invocation in IRL condition part. |
|---|

**IRL Template**

The next step to implement this extension is to design the general IRL code template. The key is to build the correct skeleton and put the generated condition code and action code in their right places. In order to let the reader put more attention to the mapping, we just simplify the action code to make it report an error message in the console if the consistency is broken. Later, if necessary, one can extend and generate more complicated user defined action template by himself.

```
rule RuleName {
       when {
          ?self: SomeClass();
          iRLIntermediateConditionContent // if any
          evaluate ( !iRLFinalCondition);
       }
       then {
          out.println("invariant ... is broken in object" + ?self.getIdString()
           + " of type " + ?self.getClass().getName());
       }
    };
```

The dynamically generated contents are `RuleName`, `SomeClass`, `iRLIntermediateconditionContent` and `iRLFinalCondition`. In order to assure the uniqueness of the rule names, they are denominated the same as their corresponding Java invariant checking method names. `SomeClass` represents the class that this invariant is referred to. `iRLIntermediateconditionContent` may contain intermediate defined collection variables depending on the complexity of the invariant. `iRLFinalCondition` is the final condition that should be evaluated by the rule engine. If the evaluation fails, the error message about the content of the invariant, ID and name of the class is printed in the console.

### 4.2.4 Changes Made to Octopus

**Public Attributes:**

First of all, the generated Java attributes should have public visibility. In this way, the attributes can be directly accessed in IRL condition part instead of invoking the corresponding `getter` and `setter` methods.

The change is made in:

```
package com.klasse.octopus.codegen.umlToJava.modelgenerators.creators;
public class AttributeCreator
field1.setVisibility(OJVisibilityKind.PUBLIC);
```

Effect:

e.g. Royal and Loyal `Customer` Class

```
public String f_name = "";
public String f_title = "";
public boolean f_isMale = false;
public Date f_dateOfBirth = null;
public int f_age = 0;
```

**Public Navigation Fields:**

It is also reasonable to set navigation fields to public and access them directly in IRL condition part. The change is made in:

```
package com.klasse.octopus.codegen.umlToJava.modelgenerators.creators;
public class NavigationCreator
field1.setVisibility(OJVisibilityKind.PUBLIC);
```

Effect:
e.g. Royal and Loyal `Customer` Class

```
public Set /*(Transaction)*/ f_cards = new HashSet( /*CustomerCard*/);
```

**Public Association End Fields:**

For the same reason, we make association end fields public.

```
package com.klasse.octopus.codegen.umlToJava.modelgenerators.creators;
public class AssocClassCreator
//set association end fields in association class to public
private void commonStuff(OJClass owner, IAssociationClass asscls) {
…
field4.setVisibility(OJVisibilityKind.PUBLIC);
field5.setVisibility(OJVisibilityKind.PUBLIC);
…
}
//set association end field in base class to public
private void addToBaseType(…){
…
field1.setVisibility(OJVisibilityKind.PUBLIC);
…
}

private void addMultToBaseType(…){
…
field2.setVisibility(OJVisibilityKind.PUBLIC);
…
}

private void addMultMultToBaseType(…) {
...
field3.setVisibility(OJVisibilityKind.PUBLIC);
…
}
```

Effect:

e.g. Royal and Loyal `Membership` Class

```
public Customer f_participants = null;
public LoyaltyProgram f_programs = null;
```

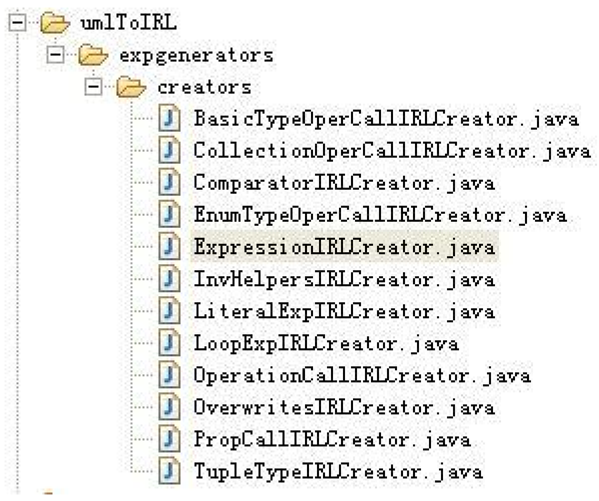
e.g. Royal and Loyal Customer Class

```
public Set /*(Transaction)*/ f_membership = new HashSet( /*LoyaltyProgram*/);
```

**Newly Added Classes**

Besides the above mentioned changes, a few new classes are added to realize the transformation from OCL to ILOG rules. They are all packed under the same package, namely, `com.klasse.octopus.codegen.umlToIRL.expgenerators.creators`. The classes are listed in figure 4.6.

.



```
umlToIRL
  expgenerators
    creators
      BasicTypeOperCallIRLCreator.java
      CollectionOperCallIRLCreator.java
      ComparatorIRLCreator.java
      EnumTypeOperCallIRLCreator.java
      ExpressionIRLCreator.java
      InvHelpersIRLCreator.java
      LiteralExpIRLCreator.java
      LoopExpIRLCreator.java
      OperationCallIRLCreator.java
      OverwritesIRLCreator.java
      PropCallIRLCreator.java
      TupleTypeIRLCreator.java
```

**Figure 4.6 Classes for IRL Generation**

`BasicTypeOperCallIRLCreator` handles the transformation of all basic type operation calls, such as "div", "mod", "size", "concat", etc.

`CollectionOperCallIRLCreator` handles the transformation of all collection operation calls, such as “count”, “excludes”, “includes”, etc.

`ComparatorIRLCreator` generates the code for the comparison result of two objects, which could be of any primitive type.

`EnumTypeOperCallIRLCreator` generates the code for the enumeration type operation.

`ExpressionIRLCreator` is the topmost class in this hierarchy. Here the OCL expression is split and sub expressions are delivered to the corresponding creators to be further processed.

`LiteralExpIRLCreator` generates code for `LiteralExp`.

`LoopExpIRLCreator` generates code either for `iteratorExp` or `iterateExp`.

`OperationCallIRLCreator` generates the code for OCL operations like `oclIsUndefined`, `oclIsTypeOf`, `oclIsKindOf`, etc.

`PropCallIRLCreator` deals with the transformation of `LoopExp`, `attributeCallExp`, `associationEndCallExp` and `associationClassCallExp`.

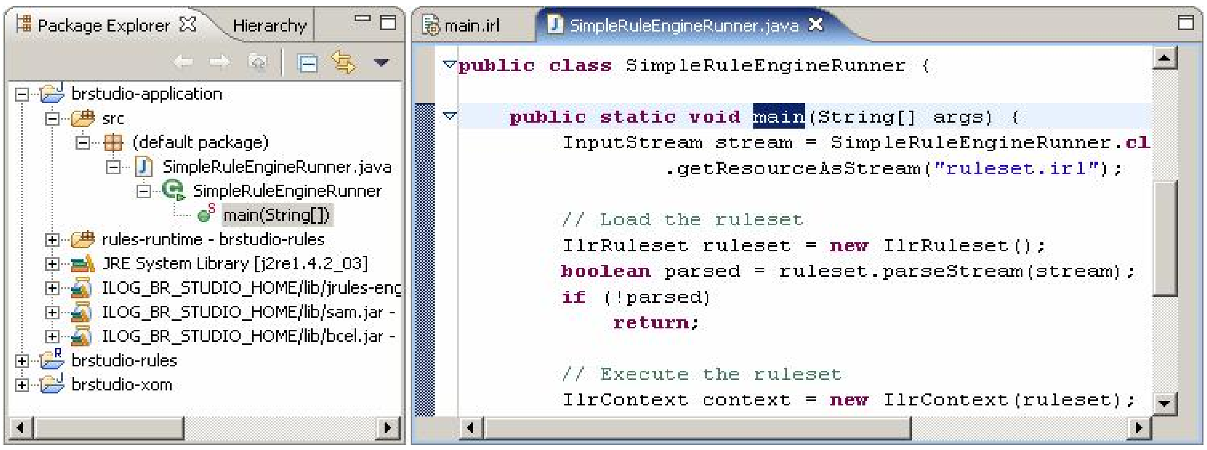
## 4.3 Runtime Execution

With the newly added extension, the user is able to automatically generate not only Java code but associated IRL rules by opening the Octopus context menu on the Java project and selecting "Generate Java Code". After doing that, one can see that the generated IRL rule files are created under the same directories where the corresponding Java files are located. Next step, one has to create a Rule project in Eclipse and define the eXecution Object Model (XOM) as the previously mentioned Octopus project. Then import the generated IRL rules in the source directory. After compilation, a file named `ruleset.irl` is generated, which contains all defined IRL rules in a single file and can be used later by the rule engine.

In realistic application environments, the rule engine is invoked using some Java code. To do this one needs a piece of Java code to:

- Initialize the Java application
- Create an instance of the rule engine
- Load some rules
- Initialize the working memory with some objects to be processed by the rules
- Start the rule engine so that it executes all rules that match the objects in the working memory
- Perform a final Java action such as output analysis, persistence, logging, and shut down the Java application.

ILOG Business Rule Studio provides a wizard that enables one to generate a Java application that carries out the steps described above. This kind of project is called **Java Project for Rules**, which contains a single runnable main class to execute some rules contained in the output folder of a rule project. Figure 4.7 displays the content of this class.


Package Explorer
Hierarchy
brstudio-application
src
(default package)
SimpleRuleEngineRunner.java
SimpleRuleEngineRunner
main(String[])
rules-runtime - brstudio-rules
JRE System Library [j2re1.4.2_03]
ILOG_BR_STUDIO_HOME/lib/jrules-eng
ILOG_BR_STUDIO_HOME/lib/sam.jar -
ILOG_BR_STUDIO_HOME/lib/bcel.jar -
brstudio-rules
brstudio-xom
main.irl
SimpleRuleEngineRunner.java

```
public class SimpleRuleEngineRunner {

    public static void main(String[] args) {
        InputStream stream = SimpleRuleEngineRunner.cl
            .getResourceAsStream("ruleset.irl");

        // Load the ruleset
        IlrRuleset ruleset = new IlrRuleset();
        boolean parsed = ruleset.parseStream(stream);
        if (!parsed)
            return;

        // Execute the ruleset
        IlrContext context = new IlrContext(ruleset);
```

**Figure 4.7 Main Class to Execute Rules**

One can now insert objects in the working memory of the rule engine using the Java API:

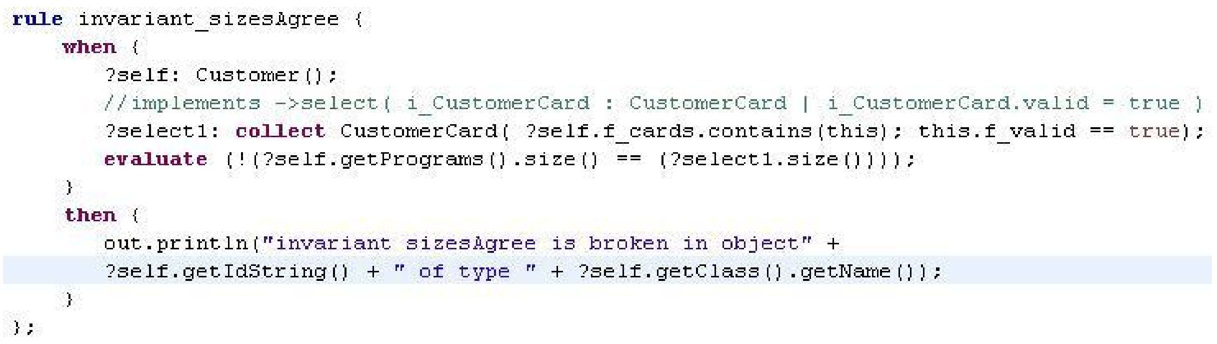
1. In the Java project, locate and edit the file `SimpleRuleEngineRunner.java`
2. In the main method, locate the line that creates the rule engine (class `IlrContext`). This line is:
   `IlrContext context = new IlrContext(ruleset);`
3. Insert a line break below and type the code to instantiate the objects.

Each time a new instance is created, the instance should be inserted into the working memory by calling `context.insert(instance);`.

And each time the object is updated in the Java code, the working memory should be informed. This is done by calling `context.update(updatedInstance, true);`. The Boolean argument is set to true, which indicates that the update will cause the agenda to be refreshed.

At the transaction commit time, `context.execute()` is invoked for consistency checking. This method executes the ruleflow defined in the context's ruleset. Any inconsistency can be caught by calling this method. If inconsistency exists, rules are fired and actions are carried out.

To better understand this process, we take the Royal and Loyal project as example again. First create a Java project, name it “RandL”. Then add Octopus nature to it and define Royal and Loyal models using UML. The previously mentioned invariant `SizesAgree` is defined here in OCL file to specify the consistency. Here we define only one single invariant in order to simplify the whole process. The generated `Customer.irl` file contains the corresponding rule. Figure 4.8 displays the generated rule of `invariant_sizesAgree`.

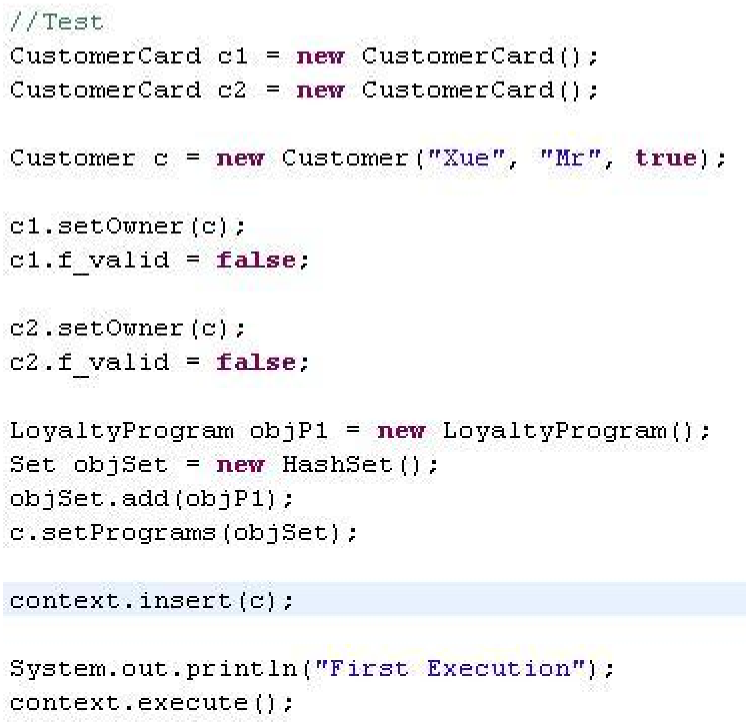


```
rule invariant_sizesAgree {
    when {
       ?self: Customer();
       //implements ->select( i_CustomerCard : CustomerCard | i_CustomerCard.valid = true )
       ?select1: collect CustomerCard( ?self.f_cards.contains(this); this.f_valid == true);
       evaluate (!(?self.getPrograms().size() == (?select1.size())));
    }
    then {
       out.println("invariant sizesAgree is broken in object" +
       ?self.getIdString() + " of type " + ?self.getClass().getName());
    }
};
```

**Figure 4.8 Generated Rule `invariant_sizesAgree`**

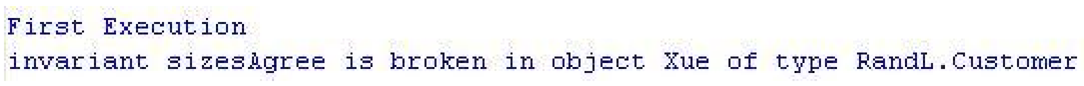
Next, one creates a Rule project called `RL_Rules` and associates the eXecution Object Model (XOM) to the previously mentioned Octopus project, namely, “RandL”. Import the `Customer.irl` file to the `src` directory in the Rule project and compile it, a file named `ruleset.irl` is generated, which contains all defined IRL rules in this project and can be used later by the rule engine, though in this case, just a single rule has been defined.

Last step, one has to create a “Java Project for Rules” to execute the rules. We name the project `RandLJavaProject4Rules`. Locate the file `SimpleRuleEngineRunner.java` and add the following code below `IlrContext context = new IlrContext(ruleset);`.



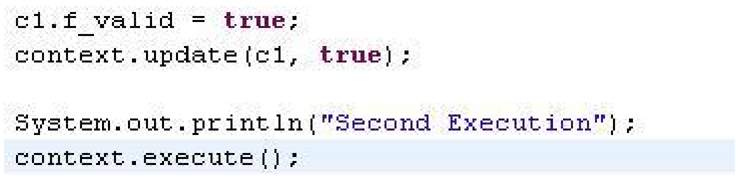
```
//Test
CustomerCard c1 = new CustomerCard();
CustomerCard c2 = new CustomerCard();

Customer c = new Customer("Xue", "Mr", true);

c1.setOwner(c);
c1.f_valid = false;

c2.setOwner(c);
c2.f_valid = false;

LoyaltyProgram objP1 = new LoyaltyProgram();
Set objSet = new HashSet();
objSet.add(objP1);
c.setPrograms(objSet);

context.insert(c);

System.out.println("First Execution");
context.execute();
```

Here two `CustomerCard` instances are created and associated to a `Customer` instance. Initially the two `CustomerCard` instances are all set to invalid (`f_valid = false`). Then a single `LoyaltyProgram` instance `objP1` is created and associated to the same `Customer` instance. Next, the `Customer` instance should be added to the working memory by calling `context.insert(c)`. In order to distinguish the execution sequence, a string is printed in the console before each time the `execute()` is called. According to the definition of invariant `SizesAgree`, the invariant is broken at the first execution time (size of `Programs` equals one, while size of valid `Cards` equals zero). The following error message is printed out in the console.



```
First Execution
invariant sizesAgree is broken in object Xue of type RandL.Customer
```

Next, we continue our work and add the following lines below the first execution.



```
c1.f_valid = true;
context.update(c1, true);

System.out.println("Second Execution");
context.execute();
```

Here one of the `CustomerCard` instances is updated to be valid. After the update, one has to inform the working memory by calling `context.update()`. Without it, the objects in the working memory are unaware of the update and thus remain inconsistent. The second execution causes no action just as one expected. Because the inconsistency is repaired by setting the `f_valid` attribute to true. In that way, the size of `Programs` and the size of valid `Cards` are the same.

## 4.4 Summary

In this chapter we implemented our first infrastructure based on Business Rules for consistency checking. Rule based approach enables the separation of application code and consistency verification. One of the advantages is that the definition of action pattern is flexible, though in this chapter we just demonstrated the simplest action, e.g. an error message is printed in the console. Other complicated and proper action patterns can be considered to improve this infrastructure.

.

# Chapter 5

# Prototype Based on AOP and Back Navigation Algorithm

In this chapter, a new prototype based on Aspect-Oriented Programming (AOP) and back navigation algorithm will be discussed. This infrastructure is also implemented as an extension of Octopus plug-in. The consistency verification should be carried out as a separate concern. In this case, AOP is the best candidate technology which can be applied to handle this kind of issue. Here back navigation algorithm is used to improve the consistency verification performance. The reader is assumed to have a basic knowledge of AspectJ. Reading the paper of my project work [Xue05] is also helpful to better understand this chapter.

## 5.1 General AOP Approach

Needless to mention, Octopus also generates Java methods for consistency checking. One inconvenient thing is that those constraint checking methods have to be invoked explicitly if we want checks (e.g. evaluation of invariants) to be carried out at runtime. We wish things get easier. We wish those checking methods to be invoked automatically. AspectJ helps us in solving this problem. First we can define the pointcut in AspectJ, namely the place where checking method should be invoked. Then we can define the advice, which checks the constraints. If it is broken, in our case an error message is printed in the console.

The general approach without back navigation algorithm is straight forward. Since Octopus generates the `checkAllInvariants` method for the Java class if any invariant is defined for this class, we could in principle call this method after the creation of an instance of this class, before and after most of the public methods. Notice that `checkAllInvariants` calls every single invariant checking method, which is defined as `public` too. These public invariant check methods should be excluded from the pointcut, otherwise the AspectJ code leads to a recursive call. And all `getter` methods defined for attributes are also excluded, since they do not change anything and could be called within invariant checking methods. Octopus generates three extra methods for every class, which are `toString()`, `getIdentifyingString()` and `allInstances()`. They should not be listed in the pointcut list either (`getIdentifyingString()` is always called within invariant checking methods, default by Octopus) .

The starting point of this extension is to design the general AspectJ code templates for enforcing invariants. Figure 5.1 shows the detailed templates.

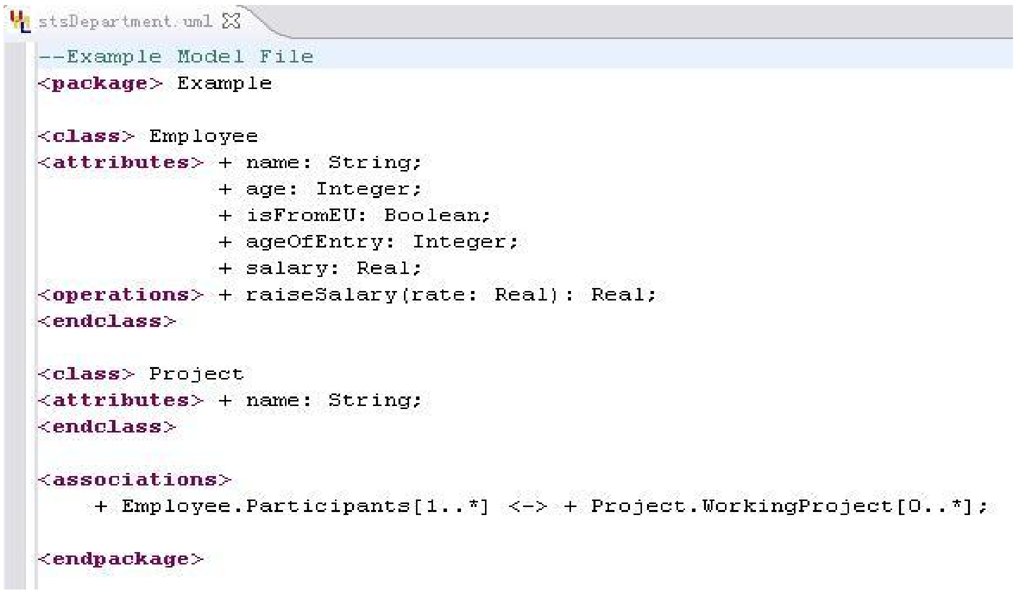
```
//define point cut for invariant check
public pointcut invPointCut(SomeClass self):
(
        (execution(public [method-return-type] SomeClass.someMethod(
        [parameter types of the method – if any]))
        [||any futhur methods]
)
&& target(self);

// Before the execution of every public method in SomeClass
before(SomeClass self) :
        invPointCut(self) {
                java.util.List invErrorList=self.checkAllInvariants();
                  for(int i=0; i<invErrorList.size(); i++){
                            System.out.println("After the execution of method in Class "
        +invErrorList.toArray()[i].toString());        }
}

// After the execution of every public method in SomeClass
after(SomeClass self) :
        invPointCut(self) {
                java.util.List invErrorList=self.checkAllInvariants();
                  for(int i=0; i<invErrorList.size(); i++){
                            System.out.println("After the execution of method in Class "
        +invErrorList.toArray()[i].toString());        }
}

// After the execution of any constructor in SomeClass
after(SomeClass self) :
        execution(public SomeClass.new(..))
        && target(self) {
                java.util.List invErrorList=self.checkAllInvariants();
                  for(int i=0; i<invErrorList.size(); i++){
                            System.out.println("After the execution of any constructor in Class "
        +invErrorList.toArray()[i].toString());        }
}
```

**Figure 5.1 AspectJ Code Templates**

The AspectJ source code will be generated according to our templates. Reviewing the templates, we can find that our main tasks include creating the condition checking method in Java source code, creating the correct pointcut, listing the class type, the method arguments and building the AspectJ skeleton. Most of the information can be extracted in the Java code generation process. Here the implementation detail is skipped.

In order to check this general AOP prototype, we create a new Octopus project and define the UML and OCL files in the following way.

UML file:

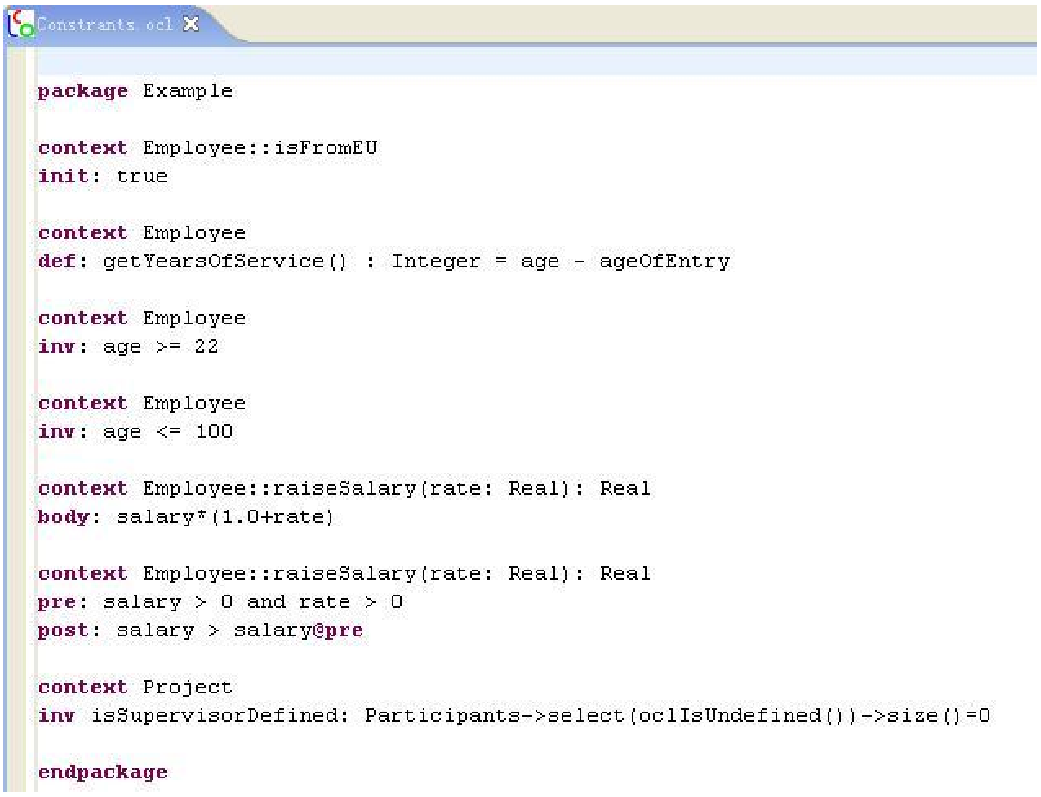


stsDepartment.uml

```
--Example Model File
<package> Example

<class> Employee
<attributes> + name: String;
             + age: Integer;
             + isFromEU: Boolean;
             + ageOfEntry: Integer;
             + salary: Real;
<operations> + raiseSalary(rate: Real): Real;
<endclass>

<class> Project
<attributes> + name: String;
<endclass>

<associations>
    + Employee.Participants[1..*] <-> + Project.WorkingProject[0..*];

<endpackage>
```

**Figure 5.2 stsDepartment.uml File**

OCL file:

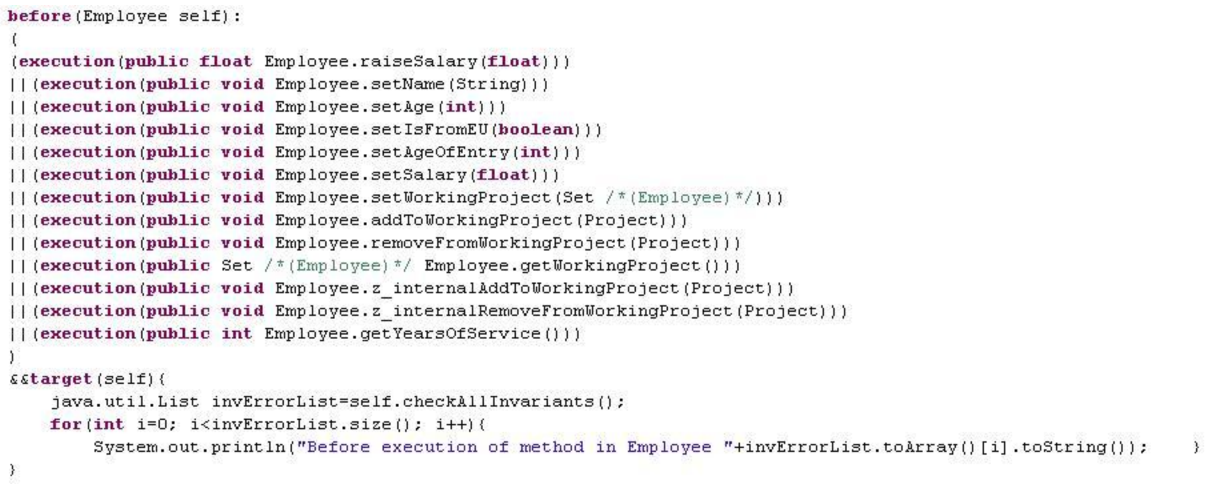


Constraints.ocl

```
package Example

context Employee::isFromEU
init: true

context Employee
def: getYearsOfService() : Integer = age - ageOfEntry

context Employee
inv: age >= 22

context Employee
inv: age <= 100

context Employee::raiseSalary(rate: Real): Real
body: salary*(1.0+rate)

context Employee::raiseSalary(rate: Real): Real
pre: salary > 0 and rate > 0
post: salary > salary@pre

context Project
inv isSupervisorDefined: Participants->select(oclIsUndefined())->size()=0

endpackage
```

**Figure 5.3 Constraints.ocl File**

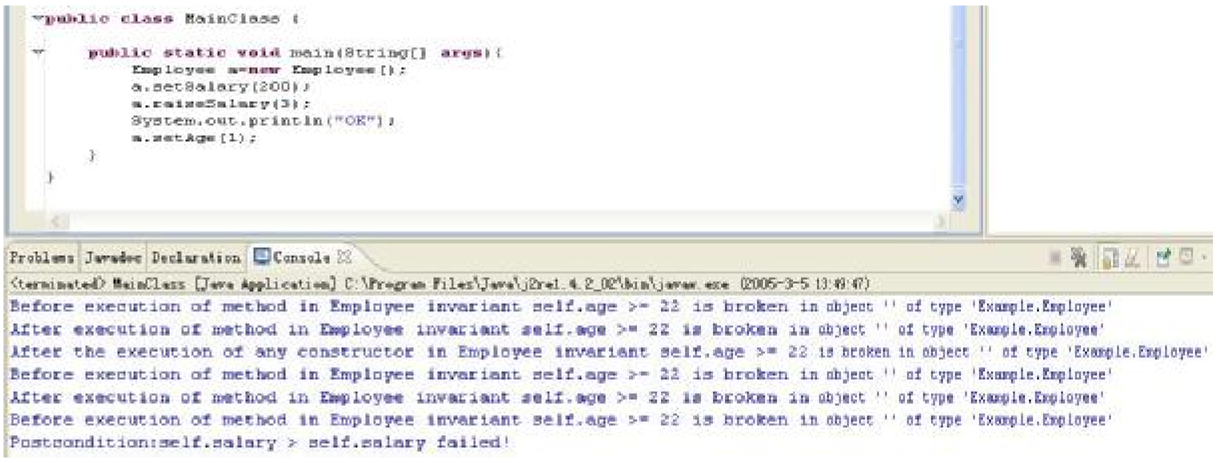
Part of the AspectJ code for enforcing invariant is listed below in figure 5.4:



```
before(Employee self):
(
(execution(public float Employee.raiseSalary(float)))
||(execution(public void Employee.setName(String)))
||(execution(public void Employee.setAge(int)))
||(execution(public void Employee.setIsFromEU(boolean)))
||(execution(public void Employee.setAgeOfEntry(int)))
||(execution(public void Employee.setSalary(float)))
||(execution(public void Employee.setWorkingProject(Set /*(Employee)*/)))
||(execution(public void Employee.addToWorkingProject(Project)))
||(execution(public void Employee.removeFromWorkingProject(Project)))
||(execution(public Set /*(Employee)*/ Employee.getWorkingProject()))
||(execution(public void Employee.z_internalAddToWorkingProject(Project)))
||(execution(public void Employee.z_internalRemoveFromWorkingProject(Project)))
||(execution(public int Employee.getYearsOfService()))
)
&&target(self){
    java.util.List invErrorList=self.checkAllInvariants();
    for(int i=0; i<invErrorList.size(); i++){
        System.out.println("Before execution of method in Employee "+invErrorList.toArray()[i].toString());    }
}
```

**Figure 5.4 AspectJ Code for Enforcing Invariant**

If having installed AspectJ Development Tools [5] (also a plug-in, available at Eclipse homepage) in Eclipse, one can test the code. First write a main class to create an instance and then invoke its methods to deliberately break the invariants. One can see the error messages printed in the console after running the program, as shown in figure 5.5.



```
public class MainClass {

    public static void main(String[] args){
        Employee a=new Employee();
        a.setSalary(200);
        a.raiseSalary(3);
        System.out.println("OK");
        a.setAge(1);
    }
}
```

```
<terminated> MainClass [Java Application] C:\Program Files\Java\j2re1.4.2_02\bin\javaw.exe (2005-3-5 13:49:47)
Before execution of method in Employee invariant self.age >= 22 is broken in object '' of type 'Example.Employee'
After execution of method in Employee invariant self.age >= 22 is broken in object '' of type 'Example.Employee'
After the execution of any constructor in Employee invariant self.age >= 22 is broken in object '' of type 'Example.Employee'
Before execution of method in Employee invariant self.age >= 22 is broken in object '' of type 'Example.Employee'
After execution of method in Employee invariant self.age >= 22 is broken in object '' of type 'Example.Employee'
Before execution of method in Employee invariant self.age >= 22 is broken in object '' of type 'Example.Employee'
Postcondition:self.salary > self.salary failed!
```

**Figure 5.5 Result of the Runtime Invariant Checking**

The readers may already find a few disadvantages of this approach. According to the AspectJ code pattern, the pointcuts are defined too generally. For example, in the above example, every time the public method `Employee.setName` is executed, the invariants must be rechecked, though this method does not influence the consistency at all. In this way, redundancy could be introduced. Another potential problem is the execution performance of `checkAllInvariants`. In the above example, only two invariants are defined for `Employee` and one for `Project`. That means, if the method `checkAllInvariants` is invoked, all private invariant checking methods will have to be executed. This could become an overload if too many invariants are defined for a single class. Considering the redundancy problem and potential poor runtime performance, one needs a more specific approach to define the pointcuts and invoke the invariant checking methods.

## 5.2 AOP and Back Navigation Approach

In order to avoid redundancy, the pointcut and advice should be defined in a more precise manner. So we're interested in defining a calculus that operates on OCL expressions (which in turn refer to a UML class model) to determine the navigation paths of the data elements on which the invariant depends. In this approach, a certain invariant checking method need only be invoked if the data elements on the navigation paths have been updated. Only those updates may potentially cause the violation of the consistency. The invariant checking performance can be improved by applying this approach.

### 5.2.1 The `noAccounts` Invariant Example

To illustrate, let us go back to the Royal and Loyal project (UML diagram is given in appendix A) and begin with an OCL invariant.

In the example an invariant `noAccounts` is given:

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

the forward-navigation links are:

```
self +
     |- partners.deliveredServices +
     |                             |- pointsEarned
     |                             |- pointsBurned
     |- Membership.account
```

Here `self` represents `LoyaltyProgram`, namely, the context class. All the other expressions (association end call, association class call and attribute call expression) in the naviagtion links are named data elements in this OCL expression.

Let us imagine the following case. At runtime, the user updates the `pointsEarned` of certain `Service` instance, all the `LoyaltyProgram` instances which are associated with this `Service` instance should be checked to insure this `noAccounts` invariant is not broken. The attribute `pointsEarned` can be updated by calling the method `setPointsEarned`. In other words, whenever `setPointsEarned` is called, the `noAccounts` invariant should be rechecked. The method `setPointsEarned` should be defined as pointcut, and the advice should be defined to invoke the invariant checking method `invariant_noAccounts()` of all associated `LoyaltyProgram` instances. The next task is to find all the associated `LoyaltyProgram` instances. The starting point is the owner of the attribute `pointsEarned`, namely, the class `Service` (`deliveredServices` is the role name, defined in UML). To find the correct associated `LoyaltyProgram` instances, one should follow the path `self.partner.programs` (here `self` represents `Service`), We denominate this path as back navigation path. The path returns a collection of associated `LoyaltyProgram`. Since Octopus has already generated the invariant checking method `invariant_noAccounts`, we need only to iterate over the collection and invoke `invariant_noAccounts` on each of them to check the consistency. The back navigation paths of all elements in `noAccounts` Invariant are listed in the table 5.1 below.

| **Element** | **back navigation** |
|---|---|
| `partners` | `self.programs` |
| `deliveredServices` | `self.partner.programs` |
| `Membersihp` | `self.programs` |
| `account` | `self.Membership.programs` |

**Table 5.1 Back Navigation Paths of All Elements in `noAccounts`**

Notice that before one decides the pointcut and back navigation path of a certain data element in the invariant expression, first one has to find the parent data element or in other words, the owner of the current data element. Both the pointcut and back link are referred to that parent element but not the current element. So, for example, if the current data element is `deliveredServices`, the corresponding parent data element is `partners`.

In this way, the back navigation paths should be

- for [`partners`] : `self`
- for [`partners.deliveredServices`] : `self.programs`
- for [`partners.deliveredServices.pointsEarned`] : `self.partner.programs`
- for [`partners.deliveredServices.pointsBurned`] : `self.partner.programs`
- for [`Membership`] : `self`
- for [`Membership.account`] : `self.programs`

The two main tasks of this approach consist in definining non side-effect-free methods as pointcut and finding the back navigation path. Here the non side-effect-free methods refer to those methods which are related to certain data element in the OCL invariant expression and invoking them may potentially break the invariant, e.g. insert entity, delete entity, update attribute, insert relationship, etc.

**Octopus generated non side-effect-free methods in invariant `noAccounts`**

Let us consider the invariant `noAccounts` again and analyze which Octopus generated Java methods are non side-effect-free in this case.

1. assignment of the self.`partners` field
   add / remove in `self.partners`
2. assignment of `deliveredServices` field in some item in `self.partners`
   add / remove in `self.partners.deliveredServices`
3. update of the simple-type attribute `Service.pointsEarned`

There may be `Services` which are not associated over `deliveredServices` to a `ProgramPartner` (because it is not a composition). In these cases, the result of the back navigation should return `null`.

4. update of the simple type attribute `Service.pointsBurned` (similar to `Service.pointsEarned`)
5. the cardinality of `self.Membership` is given by the role `participants` on the `Customer` side of the association with `LoyaltyProgram`. It is many in this case. Given that for each `Membership` a `LoyaltyAccount` will be collected, the following affects
   5.1 assignment of the `self.Membership` field
   5.2 add / remove in `self.Membership`
   5.3 assignment of an existing item in `self.Membership`
6. assignment of `account` field in some item in `self.Membership`
   (there is no add / remove in `self.Membership.account`, cardinality is 1. Nor is there assignment of an existing item in `self.Membership.account`, for the same reason)
7. given that no attribute of `LoyaltyAccount` is referred to in an iterator (instead a `->isEmpty()` is called) there is no updated of simple-type attribute in `LoyaltyAccount`.

Generally speaking, from a class the following is reachable:

- an attribute
- Lto1, an Octopus-managed association with cardinality 1 on the other end
- LtoN, an Octopus-managed association with cardinality N on the other end
- LtoA, the other end is an association class and therefore some of the involved calsses may access the tuple which they are related to by using the association class name, e.g. `Membership`, instead of the other end's role name.

### 5.2.2 General Octopus Generated Non Side-effect-free Method Pattern

Table 5.2 displays the general non side-effect-free method pattern. The reason we name it "general" is because that there are certain special cases which differ a little bit from this pattern. Those special cases will be discussed later.

| | |
|---|---|
| attribute | - set_(_) |
| Lto1 | - set_(_)<br>- z_internalRemoveFrom_(_)<br>- z_internalAddTo_(_) |
| LtoN | - set<Role>(collection)<br>- z_internalRemoveFrom_(_)<br>- z_internalAddTo_(_)<br>- addTo_(collection)<br>- addTo_()<br>- removeFrom_(collection)<br>- removeFrom_(_)<br>- removeAllFrom() no args |

**Table 5.2 General Non Side-effect-free Method Pattern**

The first special case is non Java simply type attribute. Unlike those managed by an association end or association class, given such an attribute, there is no field for the other association end (there is no code to manage the association, in fact). Let us consider the following example.

```
<class> Person
<attributes>
name: String;
age: Integer;
<endclass>

<class> Department
<attributes>
name: String;
manager: Person;
employees: Set(Person);
<endclass>
```

```
 context Department
 inv noSameName:
 employees->select( name = manager.name) -> isEmpty()
```

Given that an OCL invariant may refer to the non Java simply type attribute and that we can detect changes on it, how do we navigate back to the instance where the invariant is anchored? Although there is still a value relationship between the instances. In this case, the look-up code for a manager is that

```
Department.allInstances -> select(manager.equals(aManager))
```

and for an employee is that

```
Department.allInstances -> select(employees.contains(aEmployee))
```

In general it takes a lot of computing at runtime to resolve such problems.

Three alternatives:

(a) warn about OCL ASTs containing non Java simply type attribute (it is ok if the .uml contains them and they are not referred to in any OCL invariant)
(b) generate look-up code. Even if this code is functionally correct, its performance will approach (or exceed) for a deep enough level of the first general AOP approach.
(c) check invariant for all instances

Expressions involving association classes are dealt with in the next section.

### 5.2.3 Pointcut for Association Class

Here besides the description of the pointcut for association class, special cases of association end are also explained. They are all listed in table 5.3.

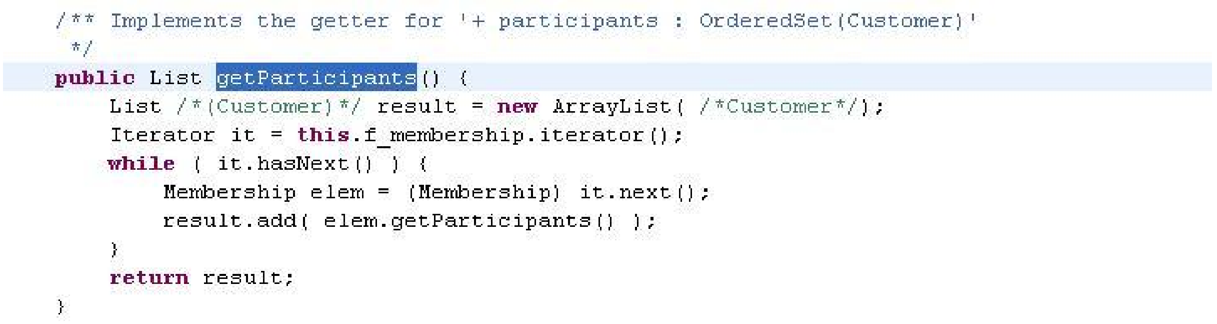
| | |
|---|---|
| LtoA, accessing the association class itself<br><br>e.g. `aLoyaltyProgram.Membership.account->isEmpty()`<br><br>All methods that affect `f_membership` | Almost the same as LtoN, because this LtoA has cardinality N (would be same as Lto1 otherwise). The differences are that external add / remove methods are referred to another association end (in this example, `participants`) and the method `removeAllFrom_()` should be included.<br><br>In `LoyaltyProgram` the following Octopus generated methods affect `f_membership`:<br><br>public void setMembership(List val)<br><br>public void z_internalAddToMembership(Membership assocClass)<br>public void z_internalRemoveFromMembership(Membership assocClass)<br><br>public void removeFromParticipants(Customer par)<br>public void removeFromParticipants(Collection oldElems)<br>public void removeAllFromParticipants()<br><br>public void addToParticipants(Customer par)<br>public void addToParticipants(Collection newElems)<br><br>Last but not the least, `setParticipants` is not included in the pointcut, because it is invoked inside `addToParticipants`. |
| LtoA, accessing an association end of an association class<br><br>e.g. `aLoyaltyProgram.participants->size()`<br><br>All methods that affect `f_membership` | Special case for association end, which is the end of an association class. The effect is the same as the above case. See explanation below the table. |
| LtoA, accessing an association class followed by the assocaition end<br>e.g. | All methods that affect `f_membership` in `LoyaltyProgram` is the same as the first case. Single method that affects `f_participants` in class `Membership` is `clean()`. See explanation below the table. |

| | |
|---|---|
| aLoyaltyProgram.Membership.participants->size() | |

Table 5.3 Pointcuts for Association Class

The first case is a pure access to the association class itself. No further explanations are necessary. Let us consider the second and the third cases. The two queries `aLoyaltyProgram.participants` and `aLoyaltyProgram.Membership.participants` actually return the same items. Below is the generated Java code

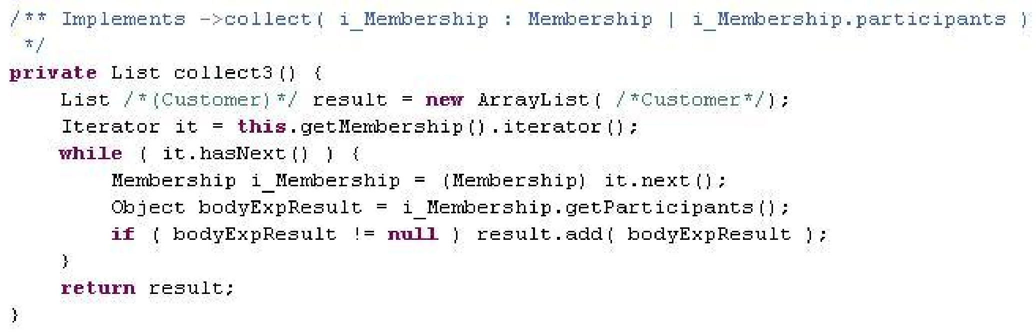
Code for `aLoyaltyProgram.participants` is shown in figure 5.6.



```
/** Implements the getter for '+ participants : OrderedSet(Customer)'
 */
public List getParticipants() {
    List /*(Customer)*/ result = new ArrayList( /*Customer*/);
    Iterator it = this.f_membership.iterator();
    while ( it.hasNext() ) {
        Membership elem = (Membership) it.next();
        result.add( elem.getParticipants() );
    }
    return result;
}
```

**Figure 5.6 Generated Java Method for `aLoyaltyProgram.participants`**

Code for `aLoyaltyProgram.Membership.participants` is shown in figure 5.7:



```
/** Implements ->collect( i_Membership : Membership | i_Membership.participants )
 */
private List collect3() {
    List /*(Customer)*/ result = new ArrayList( /*Customer*/);
    Iterator it = this.getMembership().iterator();
    while ( it.hasNext() ) {
        Membership i_Membership = (Membership) it.next();
        Object bodyExpResult = i_Membership.getParticipants();
        if ( bodyExpResult != null ) result.add( bodyExpResult );
    }
    return result;
}
```

**Figure 5.7 Generated Java Method for `aLoyaltyProgram.Membership.participants`**

In the second case, the element `participants` is normally treated as an association end. But what makes it different from a normal association end is that it has an associaiton class `Membership` defined. In other words, this association end corresponds to the end of an association class. According to our general Octopus generated non side-effect-free method pattern, the following methods should be defined as pointcut.

```
context LoyaltyProgram

public void setParticipants (List par)
public void z_internalRemoveFromParticipants(Customer par)
public void z_internalAddToParticipants(Customer par)
public void removeFromParticipants(Customer par)
public void removeFromParticipants(Collection oldElems)
public void removeAllFromParticipants()
public void addToParticipants(Customer par)
public void addToParticipants(Collection newElems)
```

But the correct result is actually:

```
context LoyaltyProgram

public void setMembership(List val)
public void z_internalAddToMembership(Membership assocClass)
public void z_internalRemoveFromMembership(Membership assocClass)
public void removeFromParticipants(Customer par)
public void removeFromParticipants(Collection oldElems)
public void removeAllFromParticipants()
public void addToParticipants(Customer par)
public void addToParticipants(Collection newElems)
```

There are no `z_internalRemoveFromParticipants` and `z_internalAddToParticipants` in class `LoyaltyProgram`. And instead of observing the method `setParticipants`, `setMembership` should be watched. Because when `setParticipants` is invoked, `addToParticipants` is called internally. This method is already under our observation.

In the third case, the association class is followed by the corresponding assocaition end. The problem is that if one writes `LoyaltyProgram.Membership.participants`, The `participants` is viewed as LtoN association end, and again according to our general pattern, the following methods are defined as pointcut:

```
context LoyaltyProgram

public void setParticipants (List par)
public void z_internalRemoveFromParticipants(Customer par)
public void z_internalAddToParticipants(Customer par)
public void removeFromParticipants(Customer par)
public void removeFromParticipants(Collection oldElems)
public void removeAllFromParticipants()
public void addToParticipants(Customer par)
public void addToParticipants(Collection newElems)
```

which is incorrect again. Actually no methods need be watched here. Because the preceding element `Membership` already generates the correct pointcut. There is a method called `clean()` in the association class `Membership`, which we should pay attention to. It removes both ends of the association. Internal `z_internalRemoveFromMembership` (in `LoyaltyProgram`) is called to catch it. We need to catch it by defining:

```
context Membership

public void clean()
```

So the difficulty lies in checking whether the association end corresponds to the end of an association class or not. This solution will be discussed later in the implementation phase.

Finally, applying the above pattern, one gets the following non side-effect-free methods for the case of `noAccounts` invariant.

```
context LoyaltyProgram

public void setPartners(Set elements)
public void z_internalAddToPartners(ProgramPartner element)
public void z_internalRemoveFromPartners(ProgramPartner element)
public void addToPartners(Collection newElems)
public void addToPartners(ProgramPartner element)
public void removeFromPartners(Collection oldElems)
public void removeFromPartners(ProgramPartner element)
public void removeAllFromPartners()
```

```
context LoyaltyProgram

public void setMembership(List val)
public void z_internalAddToMembership(Membership assocClass)
public void z_internalRemoveFromMembership(Membership assocClass)
public void removeFromParticipants(Customer par)
public void removeFromParticipants(Collection oldElems)
public void removeAllFromParticipants()
public void addToParticipants(Customer par)
public void addToParticipants(Collection newElems)
```

```
context Membership

public void setAccount(LoyaltyAccount element)
public void z_internalAddToAccount(LoyaltyAccount element)
public void z_internalRemoveFromAccount(LoyaltyAccount element)
```

```
context ProgramPartner

public void setDeliveredServices(Set elements)
public void z_internalAddToDeliveredServices(Service element)
public void z_internalRemoveFromDeliveredServices(Service element)
public void addToDeliveredServices(Collection newElems)
public void addToDeliveredServices(Service element)
public void removeFromDeliveredServices(Collection oldElems)
public void removeFromDeliveredServices(Service element)
public void removeAllFromDeliveredServices()
```

```
context Service

public void setPointsEarned(int element)
public void setPointsBurned(int element)
```

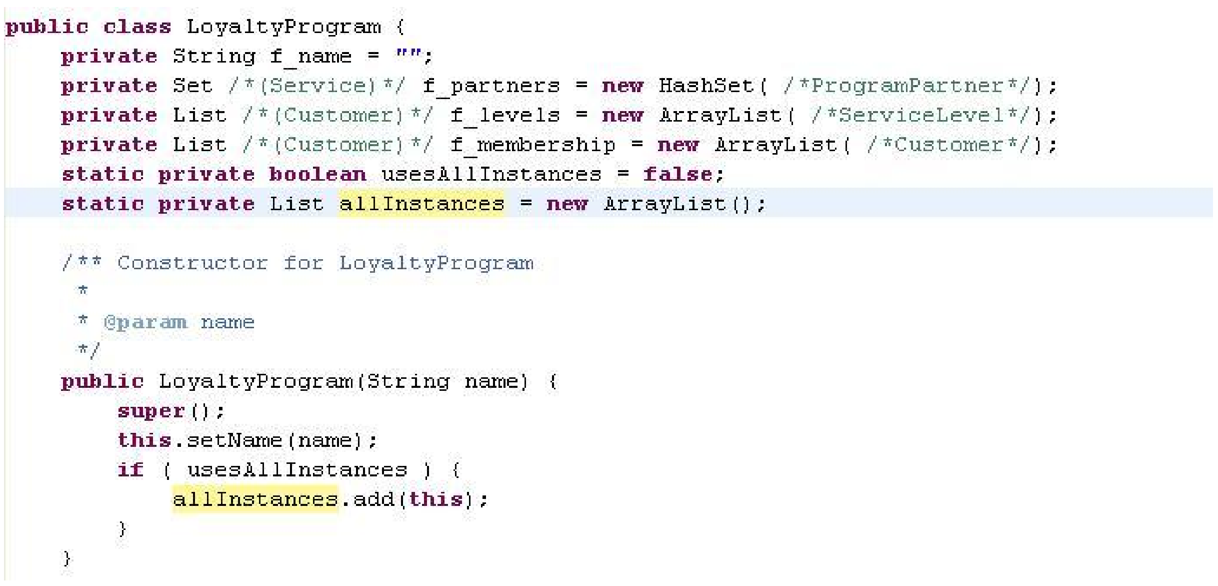
### 5.2.4 Pointcut and Advice for allInstances

The `allInstances` is a class operation that can be applied only to classes. In all other cases, it will result in undefined. For a class, it results in a set of all instances of that class, including all instances of its subclasses. From the OCL reference books, `allInstances` can only be used as `<Type>::allInstances`, however Octopus is more permissive, and therefore a translator would be able to handle the variants in syntax.

The variant is whether the source expression is an instance or a class. Here which specific instance appears is not relevant (only its declared class counts). So the following two expressions are the same.

```
anEmployee.allInstances
Employee.allInstances
```

In order to be able to return all instances of a certain class, in Octopus generated Java code, every instance need be added to a collection held by the class itself, i.e. a static field of type `Collection`. See the following generated code for class `LoyaltyProgram` in figure 5.8.



```
public class LoyaltyProgram {
    private String f_name = "";
    private Set /*(Service)*/ f_partners = new HashSet( /*ProgramPartner*/);
    private List /*(Customer)*/ f_levels = new ArrayList( /*ServiceLevel*/);
    private List /*(Customer)*/ f_membership = new ArrayList( /*Customer*/);
    static private boolean usesAllInstances = false;
    static private List allInstances = new ArrayList();

    /** Constructor for LoyaltyProgram
     *
     * @param name
     */
    public LoyaltyProgram(String name) {
        super();
        this.setName(name);
        if ( usesAllInstances ) {
            allInstances.add(this);
        }
    }
```

**Figure 5.8 `allInstances` in Generated Java Code**

There will be no back navigation to a specific instance because the invariant should be checked in turn not on a subset of instances but on all instances. Let us consider the following example.

```
context LoyaltyProgram
inv atrificialInv1:
partners.allInstances()->size() <10
```

In this case, there is no back navigation for `allInstances`. All we can do is to define a pointcut on the static variable `allInstances` of class `ProgramPartner` (`partners` is the role name of the association between `ProgramPartner` and `LoyaltyProgram`). In the advice, we get the collection of `allInstances` of type `LoyaltyProgram` and invoke the invariant checking method on each of them. Here one should distinguish the two kinds of `allInstances`. One is defined in the pointcut and the other in advice (always refers to the context class). They may belong to different class types.

The next issue we should take into consideration is that either an attribute, an association end or an association class is collected over `allInstances`. Generally speaking, anything appearing after `allInstances` belongs to this case. The previously discussed pointcut pattern is also valid here. But the advice should be defined to invoke the invariant checking method over `allInstances` of the context class. One example of this kind is given below:

```
context LoyaltyProgram
inv atrificialInv2:
partners.allInstances()->select( numberOfCustomers > 100 )->size() <10
```

Table 5.4 categorizes these two different cases.

| | |
|---|---|
| <Class>.allInstances<br>or<br>anInstance.allInstances | Pointcut on the static variable `allInstances`<br>Advice: invoke invariant checking method on all instances of context class |

| | |
|---|---|
| <Class>.allInstances.attri / asso or anInstance.allInstances.attri / asso | Pointcut same as the previously discussed pattern. Advice: invoke invariant checking method on all instances of context class |

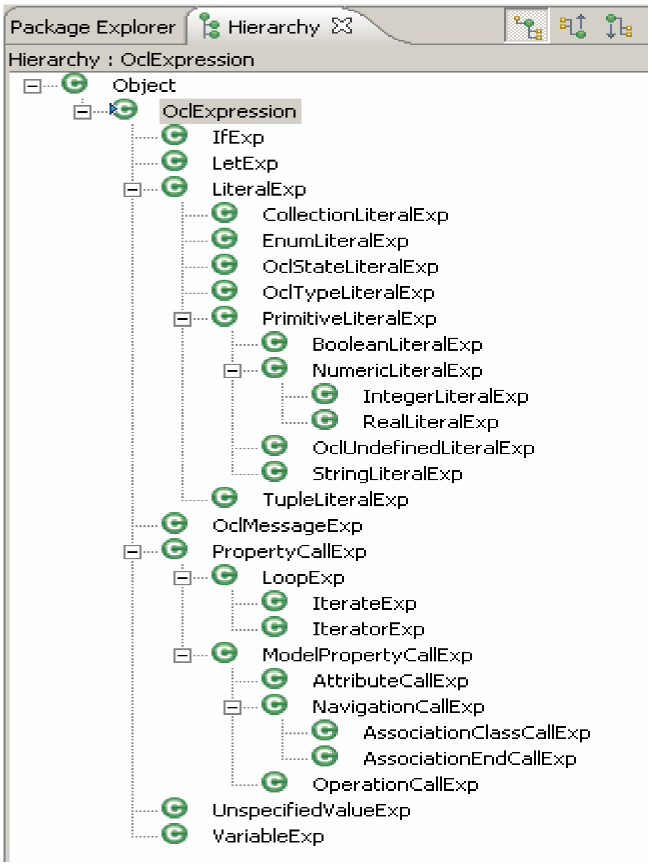
**Table 5.4 Pointcuts and Advices for `allInstances`**

## 5.2.5 Back Navigation Algorithm

In this section we will describe the back navigation algorithm. In order to better understand the algorithm, the reader should have basic knowledge about OCL metamodels, OCL AST and the general AST visitor.

Figure 5.9 displays the OCL metamodels. After exploring OCL AST, some issues relevant for the implementation of the algorithm to determine backlinks become apparent. For example, the AST walker will have visited some arguments to an operation before visiting the operation itself, as a result of OCL syntax (try with the AST of “a + b” and walk it with ASTWalker). This is in contrast to the walkers for other grammars one might be used to.

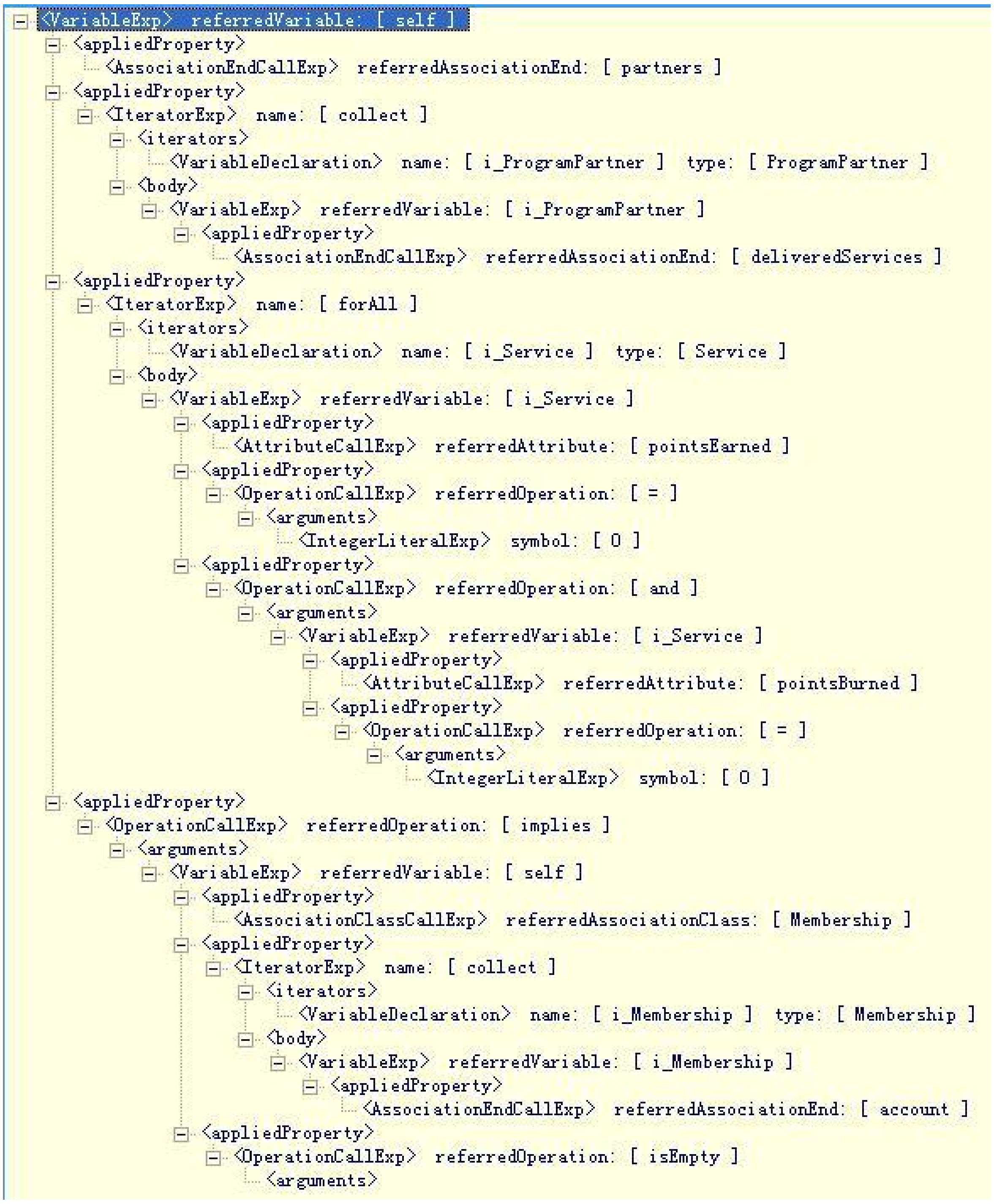
Another particularity has to do with the handling of variable declarations for ‘normal’ usages of attributes and associations vs. their usage inside the body of a collection operation. In the second case, an intermediate iterator variable is declared (which will range over each item of the collection). The usages in the body refer to the iterator variable, not to the ‘actual’ items being iterated.



```
Hierarchy : OclExpression
Object
  OclExpression
    IfExp
    LetExp
    LiteralExp
      CollectionLiteralExp
      EnumLiteralExp
      OclStateLiteralExp
      OclTypeLiteralExp
      PrimitiveLiteralExp
        BooleanLiteralExp
        NumericLiteralExp
          IntegerLiteralExp
          RealLiteralExp
        OclUndefinedLiteralExp
        StringLiteralExp
      TupleLiteralExp
    OclMessageExp
    PropertyCallExp
      LoopExp
        IterateExp
        IteratorExp
      ModelPropertyCallExp
        AttributeCallExp
        NavigationCallExp
          AssociationClassCallExp
          AssociationEndCallExp
        OperationCallExp
    UnspecifiedValueExp
    VariableExp
```

**Figure 5.9 OCL Metamodels**

Let us first take a look at the AST Tree View of previously mentioned invariant `noAccounts`, which is displayed in figure 5.10. Here what also need be mentioned is that `AstWalker` visits the `IOclExpression` in exactly the same sequence as what the whole `IOclExpression` displays in the AST Tree View (namely, from top to bottom). The inner `getAppliedProperty` is called first before the next `appliedProperty` at the same depth.



```
<VariableExp> referredVariable: [ self ]
  <appliedProperty>
    <AssociationEndCallExp> referredAssociationEnd: [ partners ]
  <appliedProperty>
    <IteratorExp> name: [ collect ]
      <iterators>
        <VariableDeclaration> name: [ i_ProgramPartner ] type: [ ProgramPartner ]
      <body>
        <VariableExp> referredVariable: [ i_ProgramPartner ]
          <appliedProperty>
            <AssociationEndCallExp> referredAssociationEnd: [ deliveredServices ]
  <appliedProperty>
    <IteratorExp> name: [ forAll ]
      <iterators>
        <VariableDeclaration> name: [ i_Service ] type: [ Service ]
      <body>
        <VariableExp> referredVariable: [ i_Service ]
          <appliedProperty>
            <AttributeCallExp> referredAttribute: [ pointsEarned ]
          <appliedProperty>
            <OperationCallExp> referredOperation: [ = ]
              <arguments>
                <IntegerLiteralExp> symbol: [ 0 ]
          <appliedProperty>
            <OperationCallExp> referredOperation: [ and ]
              <arguments>
                <VariableExp> referredVariable: [ i_Service ]
                  <appliedProperty>
                    <AttributeCallExp> referredAttribute: [ pointsBurned ]
                  <appliedProperty>
                    <OperationCallExp> referredOperation: [ = ]
                      <arguments>
                        <IntegerLiteralExp> symbol: [ 0 ]
  <appliedProperty>
    <OperationCallExp> referredOperation: [ implies ]
      <arguments>
        <VariableExp> referredVariable: [ self ]
          <appliedProperty>
            <AssociationClassCallExp> referredAssociationClass: [ Membership ]
          <appliedProperty>
            <IteratorExp> name: [ collect ]
              <iterators>
                <VariableDeclaration> name: [ i_Membership ] type: [ Membership ]
              <body>
                <VariableExp> referredVariable: [ i_Membership ]
                  <appliedProperty>
                    <AssociationEndCallExp> referredAssociationEnd: [ account ]
          <appliedProperty>
            <OperationCallExp> referredOperation: [ isEmpty ]
              <arguments>
```

**Figure 5.10 AST Tree View of Invariant `noAccounts`**

Since we are interested in navigation path, in other words, association, only `associationEndCallExp` and `associationClassCallExp` are of the most importance. Both of these two expressions have a method called `getSource()`, which returns the referred variable of the current expression. We know that the attribute or association will in fact range over a collection, and that the iterator variable stands for one element of that collection at a time. For example, `getSource()` returns `IVariableExp` "`i_ProgramPartner`" for the `AssociationEndCallExp` "`deliveredServices`". But an `IVariableExp` has no method `getSource()`. Neither does it have its `getReferredVariable()`. Therefore, `IAssociationEndCallExp` and `IAssociationClassCallExp` (once visited) can only trace their anchor back to the `IVariableExp`. But what this `IVariableExp` iterates over, cannot be determined unless intermediate infomation has been kept. Only in this way one can know to whom this `IVariableExp` should be anchored. Lists `f_Vari` and `f_ForwardLink` are defined to record intermediate information. The former records all visited `IVariableDeclaration`, the latter saves the forward navigation path, which will be used to build the back navigation path. These two lists are always assumed to have the same size.

Figure 5.11 displays the content of the back navigtion algorithm.

```
AspectJInvariantVisitor implements IAstVisitor{

  List f_gatheredStuffs
  -- save all visted IAssociationEndCall, IAssociationClass or IAttribute, without
  -- repetitation. In the case that the same IAssociationEndCall, IAssociationClass or
  -- IAttribute appears more than once, in order to avoid creating the pointcut more than
  -- once, not part of the algorithm
  List <IvariableDeclaration> f_Vari
  -- save all visited IVariableDeclaration
  List <String> f_ForwardLink
  -- save the back navigation forwardly, later one can build the correct back naviation
  -- by traversing the list reversely
  Boolean f_AllInstances_visited = false;
  -- a flag to indicate if allInstances has been visited in this particilar OCL branch

  variableExp {
      Check the name of the referred variable, if it equals "self"{
          Clean f_Vari, f_ForwardLink
          Set f_AllInstances_visited to false
      }
  }

  variableDeclaration{
      Add the name of variable declaration to f_Vari
  }

  operationCallExp{
      if the name of referred operation equals "allInstances"{
        f_AllInstances_visited = true;
        No back navigation, all instances should be checked
      }
  }
```

```
associationEndCallExp{
     If name of referred variable does not equal "self"{
          Get the index of the name in f_Vari
          If the index does not equal the size of f_Vari – 1 {
             Only keep the first index + 1 items in
              f_Vari, f_ForwardLink
          }
    }

    If for the case that
    this associationEnd is an end role of an associationClass
    -- e.g. AssoEnd participants is the end role of AssoClass Membership
    and
    this associationEnd is preceded by exactly that corresponding associationClass
    -- e.g. LoyaltyProgram.Membership.participants,
    {
      back navigation returns the name of the association class -- e.g. Membership
      add the back navigation of this associationEnd to f_ForwardLink
    }
    else
    {
      back navigation returns the name of the other association end
      add the back navigation of this associationEnd to f_ForwardLink
    }

    If f_AllInstances_visited equals true
    {
         No back navigation, all instances should be checked
    }
    else
    {
          Build the correct back navigation by traversing all items in the f_ForwardLink
          reversely
    }
}

associationClassCallExp{
    If name of referred variable does not equal "self"{
      Get the index of the name in f_Vari
      If the index does not equal the size of f_Vari – 1 {
           Only keep the first index + 1 items in
           f_Vari, f_ForwardLink
        }
    }

    add the back navigation of this associationEnd to f_ForwardLink

    If f_AllInstances_visited equals true
    {
         No back navigation, all instances should be checked
    }
```

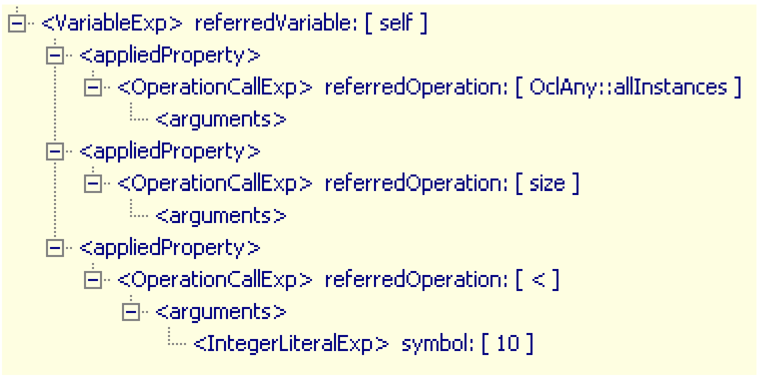
```
            else
            {
                Build the correct back navigation by traversing all items in the f_ForwardLink
                reversely
            }
        }
}
```

**Figure 5.11 Back Navigation Algorithm**

In the AST, the referred variable "`self`" represents the context class itself. In a single OCL invariant expression, "`self`" can appear more than once. Every time "`self`" is visited, it means everything starts from the beginning (for example, in the `noAccounts` invariant example, "`self`" occurs twice). The previously recorded information, e.g. the forward navigation path or the names of visited variable, becomes useless. All information in the algorithm must be reset to the initial conditions at this point. It is done in `variableExp`.

The `allInstances` in an AST is actually a parameterless operation. See the following example:

```
context Employee
  inv whatIsAllInstances :
     allInstances()->size() < 10
```



```
<VariableExp> referredVariable: [ self ]
    <appliedProperty>
        <OperationCallExp> referredOperation: [ OclAny::allInstances ]
            <arguments>
    <appliedProperty>
        <OperationCallExp> referredOperation: [ size ]
            <arguments>
    <appliedProperty>
        <OperationCallExp> referredOperation: [ < ]
            <arguments>
                <IntegerLiteralExp> symbol: [ 10 ]
```

**Figure 5.12 `allInstances` in AST**

So one needs to check each `operationCallExp` and catch the case that the name of referred operation equals "`allInstances`". No back navigation path needs to be computed here and neither do those elements afterwards on the same branch.

From `associationEndCallExp` or `associationClassCallExp` one can get the name of corresponding referred variable. The index (depth) of that referred variable is calculated and the first index + 1 items in `f_ForwardLink` bulid the forward back navigation, one only needs to rebuild the correct result by traversing all items in the `f_ForwardLink` reversely.

Special case is the pattern `associationClass.associationEndOfThatAssociationClass` should be treated differently from normal association end. In this case the back navigation of the `associationEnd` should be the name of the association class itself but not the name of the other association end. Table 5.5 lists the case for `participants`.

| **OCL Expression** | **Back Navigation Path of `participants`** |
|---|---|
| `LoyaltyProgram.participants` | `programs` |
| `LoyaltyProgram.Membership.participants` | `Membership.programs` |

**Table 5.5 Back Navigation Paths of Two Different `participants`**

This algorithm has been intensively tested on all the OCL invariants provided in the Royal and Loyal project and proved to work correctly. However, there are some cases where the current algorithm does not return the expected back navigations.

Let's assume the following UML:

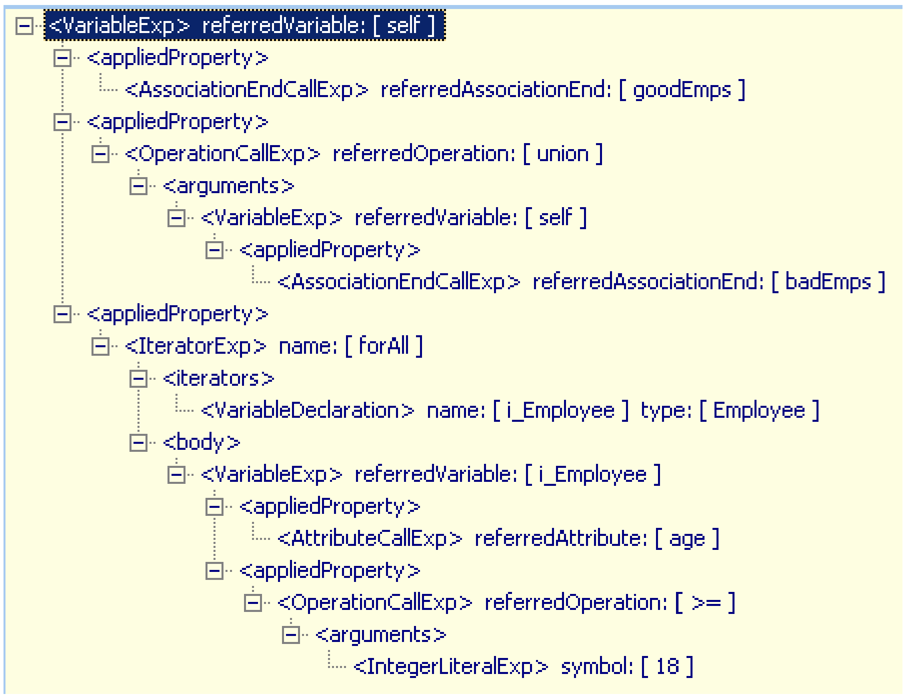
```
<package> myocl

<class> Department
<attributes>
   name : String;
<endclass>

<class> Employee
<attributes>
   age : Integer;
   addrAsAttr : Address;
<endclass>

<class> Address
<attributes>
   number : Integer;
<endclass>

<associations>
  Department.deptOfGoodEmp [0..*] <-> Employee.goodEmps [0..*];
  Department.deptOfBadEmp [0..*] <-> Employee.badEmps [0..*];
  Employee.empRoleWork <-> Address.workAddress [1];
  Employee.empRoleHome <-> Address.homeAddress [1];

<endpackage>
```

and the following OCL invariant:

```
package myocl

context Department
  inv problemWithUnion :
    goodEmps->union(badEmps)->forAll(age >= 18)

endpackage
```

The invariant `problemWithUnion` shows that the usage of `age` in the body of the `forAll` has an `i_Employee` as `referredVariable()` (see figure 5.13). In general, after a `->union()` the iterator variable will be typed with the nearest common ancestor in the type hierarchy. In this case, both sets under `union()` have the same type.

If fragments of the backlink are collected in the visit order supported by `ASTWalker`, the partial chain at the time of visiting `age` will be `self.badEmps`. The fact that in addition to `self.badEmps`, the backlink over the `goodEmps` association need also be followed at runtime is overlooked (the same `Employee` could be reachable over `goodEmps` from one `Department` and over `badEmps` from another, when an update occurs in `Employee.age` all relevant instances of `Department` should be returned).



```
<VariableExp> referredVariable: [ self ]
    <appliedProperty>
        <AssociationEndCallExp> referredAssociationEnd: [ goodEmps ]
    <appliedProperty>
        <OperationCallExp> referredOperation: [ union ]
            <arguments>
                <VariableExp> referredVariable: [ self ]
                    <appliedProperty>
                        <AssociationEndCallExp> referredAssociationEnd: [ badEmps ]
    <appliedProperty>
        <IteratorExp> name: [ forAll ]
            <iterators>
                <VariableDeclaration> name: [ i_Employee ] type: [ Employee ]
            <body>
                <VariableExp> referredVariable: [ i_Employee ]
                    <appliedProperty>
                        <AttributeCallExp> referredAttribute: [ age ]
                    <appliedProperty>
                        <OperationCallExp> referredOperation: [ >= ]
                            <arguments>
                                <IntegerLiteralExp> symbol: [ 18 ]
```

**Figure 5.13 AST of invariant `problemWithUnion`**

### 5.2.6 Drawback of allInstances and Weak Reference

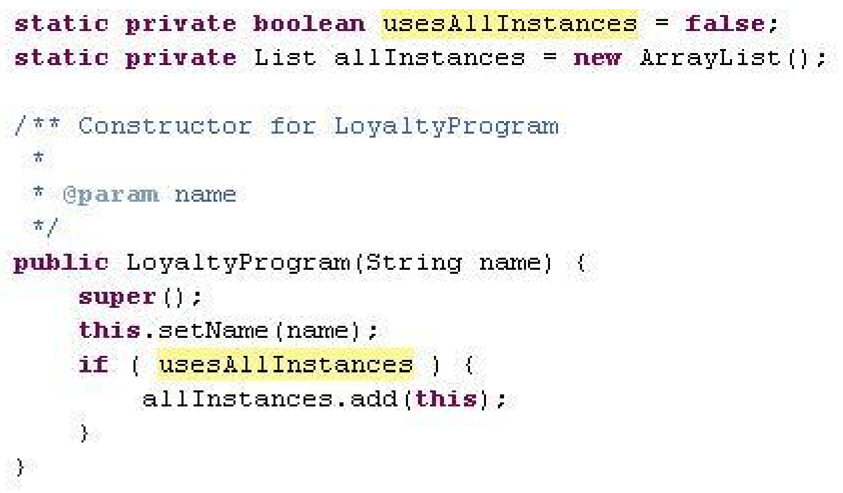
From the Octopus documentation, a drawback to use a static field to hold all instances of a certain class is that the garbage collection will not do its work properly when an instance is present in `allInstances` collection but is not used anymore elsewhere. Having a reference to an object that yet does not prevent the object from being garbage collected is possible in Java with `java.lang.ref.WeakReference`. Part of the API is given below:

```
WeakReference(Object referent)
```
Creates a new weak reference that refers to the given object.

```
public Object get()
```
Returns this reference object's referent. If this reference object has been cleared, either by the program or by the garbage collector, then this method returns null.

Suppose that the garbage collector determines at a certain point in time that an object is weakly reachable. At that time it will atomically clear all weak references to that object and all weak references to any other weakly-reachable objects from which that object is reachable through a chain of strong and soft references. At the same time it will declare all of the formerly weakly-reachable objects to be finalizable. This API allows a programmer to maintain special references to objects that allow the program to interact with the garbage collector in limited ways. Detailed information about weak reference can be found in SDK documentation.

Additionally, an optimization could be made to Octopus generated Java code. Instead of generating the code like:



```
static private boolean usesAllInstances = false;
static private List allInstances = new ArrayList();

/** Constructor for LoyaltyProgram
 *
 * @param name
 */
public LoyaltyProgram(String name) {
    super();
    this.setName(name);
    if ( usesAllInstances ) {
        allInstances.add(this);
    }
}
```

**Figure 5.14 `usesAllInstances` in Octopus Generated Java Code**

one could generate `usesAllInstances` field to be `final` and set it to `true` for some classes. That way, the compiler can optimize away unreachable code in the then-part of

```
if ( usesAllInstances )
```

Another argument for declaring that field `final` is that it does not make sense to make it go from `false` to `true` at runtime, as there is no way to know what instances might have been created beforehand and still be reachable on the heap. Setting it from `true` to `false` instead would be reasonable, resulting in freeing the `WeakReferences` that collect `allInstances`.

Let us consider the following invariant:

```
context Department
inv numberEmployees:
self.employee->size() <= Employee.allInstances()->size()/2
```

In the example, we will assume that unlinking an `employee` from the `department` makes it a candidate for garbage collection. In terms of management of the static variable `Employee.allInstances`, assuming the pattern based on `WeakReference` is used, one has the items of that collection unchanged, only a wrapped value in `WeakReference` to that `employee` (which goes to null).

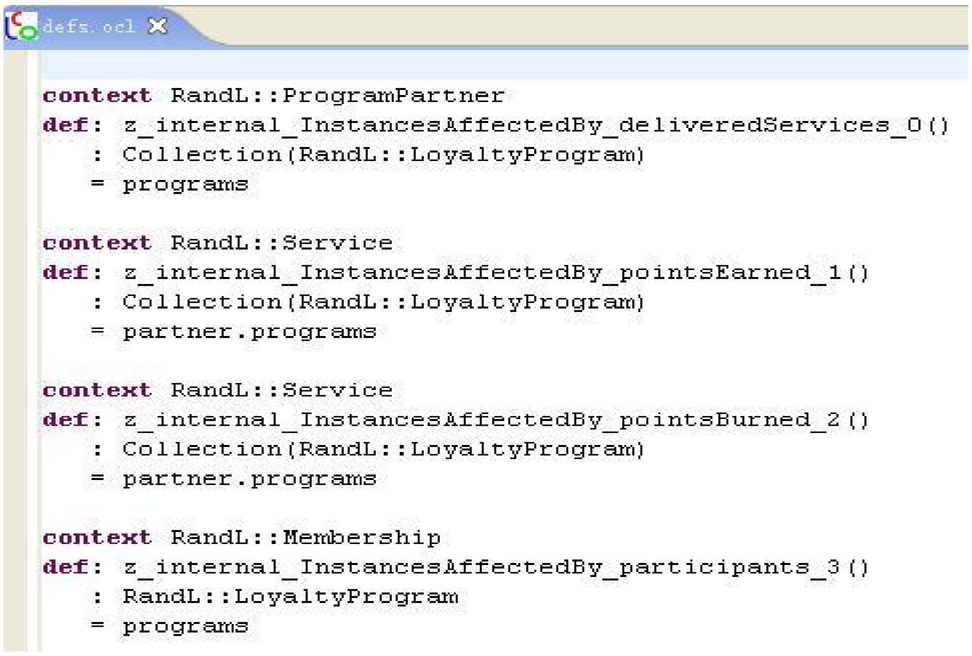
## 5.2.7 Other Issues

No updates during the evaluation of invariants should be assumed. Once the instances returned by the back navigation have been determined, the next step consists in invoking the invariant checking method, which is supposed to be side-effect-free. Otherwise, the pointcuts to detect state updates will be activated. This should be considered as an error or an exception.

It does not appear that during the evaluation of invariants, intermediate results would result in the instantiation of business domain objects. There is no object instantiation construct in OCL. One needs resorting to invoking a user-defined operation to achieve that. All OCL self-provided operations (e.g. `size()`, `asSequence()`, `oclIsUndefined()`) do not influence the business domain objects. An OCL expression containing a user-defined operation cannot be considered to be side-effect-free.

After the back navigation path is computed, one can take the advantage of the Java code generation functionality provided by Octopus. Here a single OCL file is generated, which contains only OCL def expressions of the form `z_internal_InstancesAffectedBy_BlaBla`. Each definition expression corresponds to the result of a back navigation path. The result can either be a collection of instances or a single instance, depending on the multiplicity of the association. Later this .ocl file will be translated to Java by Octopus. The body of the AspectJ advice will invoke those generated methods to gather the references to the affected instances and to wrap them in the WeakReferences before queuing them for evaluation at transaction commit time.

Figure 5.15 displays the generated defs.ocl of invariant `noAccounts`.



```
context RandL::ProgramPartner
def: z_internal_InstancesAffectedBy_deliveredServices_0()
    : Collection(RandL::LoyaltyProgram)
    = programs

context RandL::Service
def: z_internal_InstancesAffectedBy_pointsEarned_1()
    : Collection(RandL::LoyaltyProgram)
    = partner.programs

context RandL::Service
def: z_internal_InstancesAffectedBy_pointsBurned_2()
    : Collection(RandL::LoyaltyProgram)
    = partner.programs

context RandL::Membership
def: z_internal_InstancesAffectedBy_participants_3()
    : RandL::LoyaltyProgram
    = programs
```

**Figure 5.15 Generated defs.ocl of Invariant `noAccounts`**

### 5.2.8 Consistency Verification Infrastructure

In order to perform the consistency verification on models, an infrastructure is needed. Remember in our case, there are two types of back navigation, or better to say, one is computable, the other does not exist (in this case, `allInstances` of the context class should be checked, so we denote the back navigation path as "`allInstances`"). Two different mechanisms are necessary to handle them. For the case that back navigation path is computable, `WeakReference` is applied to hold the affected instance. What we also need is a `String` to save the name of the invariant checking method generated by Octopus. These two build a pair and we name it as `WeakRefInstance_Inv_Pair`. Again it corresponds to a single affected instance. To collect all affected instances, `WeakRefInstance_Inv_Set` is used. So at transaction commit time, one can iterate over this set and use the Java reflection to invoke the invariant checking method on each affected instance to check the consistency, which is done by invoking the method `EvaluateWeakRefInstance_Inv_Set`. If the invariant is broken, an error message is printed in the console. At the end all instances which did not break the consistency should be deleted from the `WeakRefInstance_Inv_Set` and only those broken ones will be kept for rechecking at the next transaction commit time.

For the second case, one has to handle the invariant checking of all instances. Here instead of affected instance, affected class type should be saved. At the transaction commit time, one can get `allInstances` from that class type (by using Java reflection) and then iterate over each of them to invoke the invariant checking method. The rest is the same as the first case. We name them `Class_Inv_Pair` and `Class_Inv_Set`.

At last we need a class `InvCheckInfrastructure` to hold these two different sets. These two sets are defined as static fields so that one can access them by calling the class every time. At transaction commit time, the consistency can be checked just by invoking `InvCheckInfrastructure.EvaluateConsistency()`, which is also a static method. The complete Java code of this infrastructure is listed in appendix B.

### 5.2.9 AspectJ Code Templates

Also in this approach one needs to design the general AspectJ code templates for enforcing invariants for later checking. There are three different cases, depending on whether the back navigation path exists and the number of affected instances if the path exists. The templates are shown in figure 5.16.

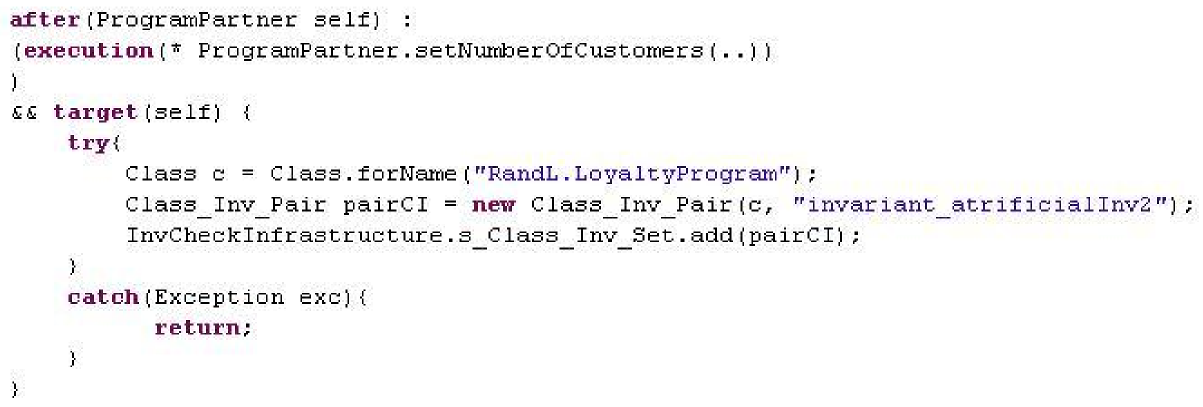
```
//for the case that back navigation exists and a collection of instances are affected
after (SomeClass self) :
    (
        pointcut generated according to the pointcut pattern
     )
     && target(self) {
        Collection coll = null;
        try{
                coll = self.z_internal_InstancesAffectedBy_BlaBla();
        }
        catch(Exception exc){
                return;    //error message could be printed in the console here
        }
        if(coll.equals(null)) return;
        Iterator it = coll.iterator();
        while ( it.hasNext() ) {
        Object o = it.next();
        WeakRefInstance_Inv_Pair      pairWI      =      new      WeakRefInstance_Inv_Pair(o,
"InvariantCheckingMethodName");
        InvCheckInfrastructure.s_WeakRefInstance_Inv_Set.add(pairWI);
      }
    }
```

```
//for the case that back navigation exists and a single instance is affected
after (SomeClass self) :
    (
        pointcut generated according to the pointcut pattern
     )
     && target(self) {
       Object o = null;
       try{
                o = self.z_internal_InstancesAffectedBy_BlaBla();
       }
       catch(Exception exc){
                return;    //error message could be printed in the console here
       }
       if(o.equals(null)) return;
       WeakRefInstance_Inv_Pair      pairWI      =      new      WeakRefInstance_Inv_Pair(o,
"InvariantCheckingMethodName");
       InvCheckInfrastructure.s_WeakRefInstance_Inv_Set.add(pairWI);
     }
```

```
//for the case that back navigation does not exist
after(SomeClass self) :
     (
        pointcut generated according to the pointcut pattern
     )
     && target(self) {
        try{
          Class c = Class.forName("AffectedClassName");
          Class_Inv_Pair pairCI = new Class_Inv_Pair(c, " InvariantCheckingMethodName");
          InvCheckInfrastructure.s_Class_Inv_Set.add(pairCI);
        }
        catch(Exception exc){
                return;    //error message could be printed in the console here
        }
     }
```

**Figure 5.16 AspectJ Code Templates**

If the defined back navigation method returns a collection of affected instances, one has first to iterate over the collection to get each instance, and then add the instance and the name of its invariant checking method to the invariant checking infrastructure. If no back navigation path exists, the affected class type in stead of instance is recorded. An example of the third case is given below in figure 5.17:



```
after(ProgramPartner self) :
(execution(* ProgramPartner.setNumberOfCustomers(..))
)
&& target(self) {
    try{
        Class c = Class.forName("RandL.LoyaltyProgram");
        Class_Inv_Pair pairCI = new Class_Inv_Pair(c, "invariant_atrificialInv2");
        InvCheckInfrastructure.s_Class_Inv_Set.add(pairCI);
    }
    catch(Exception exc){
          return;
    }
}
```

**Figure 5.17 Sample AspectJ Code**

## 5.2.10 Code Generation Process

The code generation process is divided into pre-generation phase and formal code generation phase. In the pre-generation phase, both the .uml and .ocl files are parsed and models are built into memory. During this phase, OCL invariant expressions are visited and back navigation paths for each data element are computed. Each back navigation path results in a new OCL definition expression in the newly generated `defs.ocl` file. Now, the `defs.ocl` file and the previous defined .uml and .ocl files build new models.

In the formal code generation phase, again, new models (equipped with operations defined in `defs.ocl` file) are built in memory. Invariants of each class are visited. For each potentially affected element in that OCL invariant expression, AspectJ file with properly defined pointcut and advice is created. The AspectJ file is created under the same package where the corresponding Java file is located. The Java files for invariant checking infrastructure are also generated at the end, which are all located under the `src/Infrastructure` directory.

## 5.2.11 Implementation

Each `OJClass` has a field `f_hasAspectJ` of type `boolean`. Default value of `f_hasAspectJ` is `false`. This field is set to `true` if the corresponding `OJClass` has invariant defined and so the `OJClass` will be translated to AspectJ.

A new package `aspectjgenerators` is created under `com.klasse.octopus.codegen.umlToJava`. New classes related to AspectJ file generation are added here. To study these new classes, we will first go back to class TransformationController, where the AspectJ generation starts.

In the class TransformationController, new method generateAspectJ is added, where an AspectJController instance is created and its transform method is invoked. In this transform method, the umlmodel is checked first. If things goes fine, instance of class AspectJInvariantsGenerator is created, which extends DefaultPackageVisitor and can be used to visit the umlmodel. In this case only Class is of interest to us. In method class_Before, each OJClass is checked to see if it contains invariants. If so, AspectJInvariantCreator is created and its addInvariant method is responsible for appending AspectJ info to OJPackage.

Before continuing, an extension to OclContextImpl should be explained. A field f_correspondingJavaMethodName is added to OclContextImpl. Its function is to save the corresponding Java method name of the OclContext. In com.klasse.octopus.codegen.umlToJava.expgenerators.creators InvariantCreator, method addInvariant is responsible for creating the invariant Java method. Extension is made here

```
((OclContextImpl)cont).setCorrespondingJavaMethodName(INV_NAME);
```

to save the INV_NAME to the OclContextImpl, so that next time when the visitors travels this OclContext, it knows what the name of the Java method is. So the sequence is important that generateAspectJ should be invoked after the invocation of generateExpressions.

Back to AspectJInvariantCreator, in method addInvariant, AspectJInvariantVisitor is created to process the invariant. It implements IAstVisitor and takes two parameters. One is the OJPackage, which represents the whole Java model. The other is a String, which records the CorrespondingJavaMethodName.

Take the example again:

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

```
self +
     |- partners.deliveredServices +
     |                              |- pointsEarned
     |                              |- pointsBurned
     |- Membership.account
```

Here we are interested in IAttributeCallExp, IAssociationEndCallExp and IAssociationClassCallExp. We need to find the non side-effect-free methods, parent class and back navigation for each of them. The starting point is the method addAspectJtoParentClass. Here setHasAspectJ is invoked to set the f_hasAspectJ to true to indicate that the OJClass contains invariant and should be translated to AspectJ code later.

For example, when the visitor encounters deliveredServices, the parent class is ProgramPartner. A pointcut and advice should be generated for ProgramPartner. The parent class can be computed by:

```
IClassifier c = exp.getSource().getNodeType();
if ( c instanceof ClassifierImpl && !(c instanceof IEnumerationType)) {
    ClassifierImpl in = (ClassifierImpl) c;
    OJPathName path = GenerationHelpers.pathname(in.getPathName());
    OJClass myClass = f_OJPackage.findClass(path);
    …
```

With the help of `StructuralFeatureMap`, one can get the names of all non side-effect-free methods. For the case of `IAssociationEndCallExp` and `IAssociationClassCallExp`, one should first check the multiplicity to see whether it's Lto1 or LtoN and then choose the correct non side-effect-free method pattern, which is done in method `isMultiplicityTypeLtoN`.

## 5.3 Summary

In this chapter, we implemented our second prototype based on AOP and back navigation algorithm. A simple and straight forward approach was first discussed to illustrate the benefit of applying AOP technology. Afterwards, what also became clear was that the pointcut and advice were defined too generally, which may result in poor performance in a large modeling project. Due to that reason, a more complicated approach was conceived to conquer the problem. Back navigation algorithm is used to compute the potentially affected instances and also in this way the invocation of invariant checking methods can be limited to the minimum. The improvement of performance is achieved.

# Chapter 6

# Conclusions

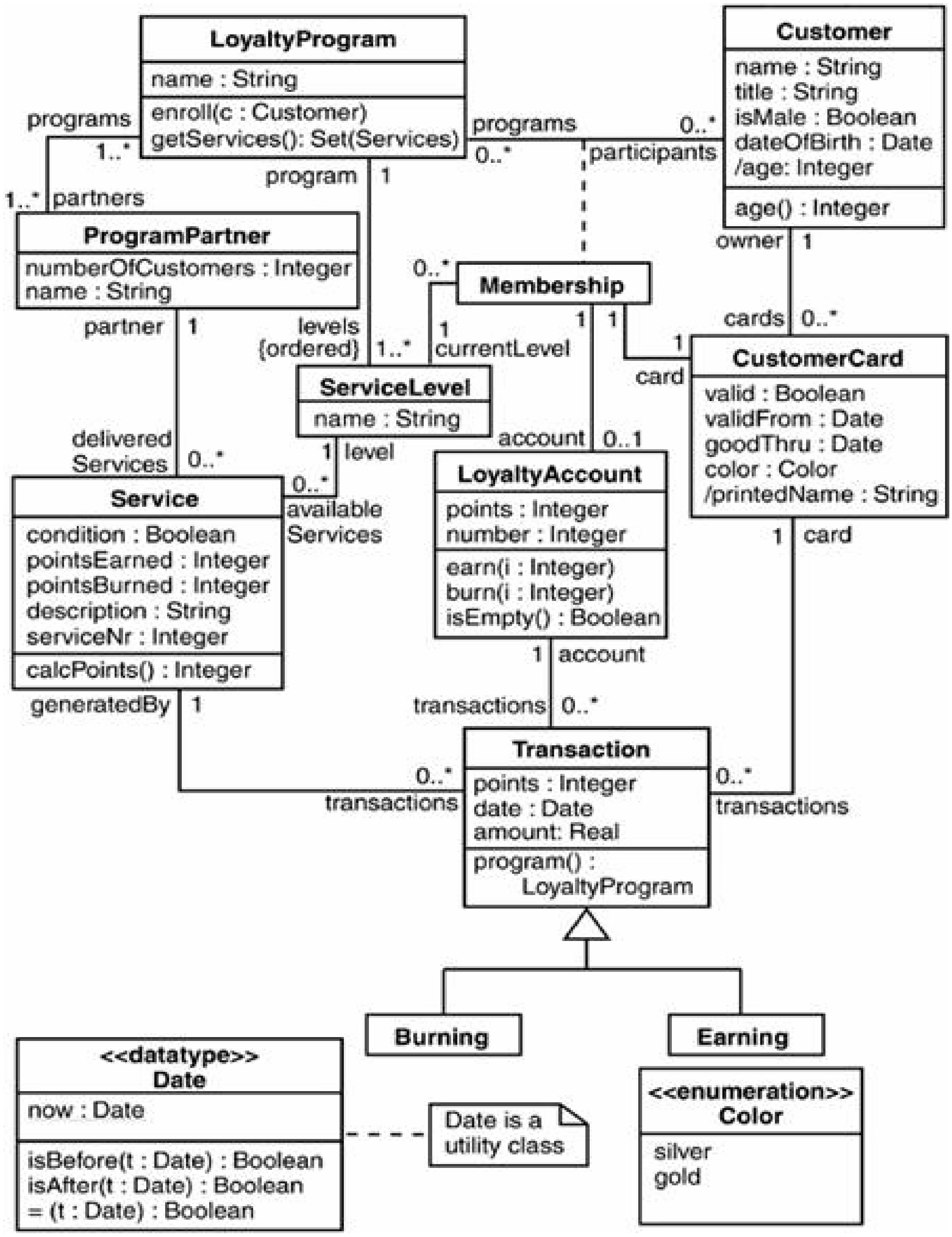
The goal of this master thesis is to design and implement prototypes which solve the consistency verification problems in model engineering.

We started from the introduction of consistency problems in model engineering. After that, the project “Using ATL for Checking Models” was studied and served as a good example for us. This approach introduces the extension of constraints with additional information and can be used for any metamodel and in several contexts for consistency verification.

Before we started to implement our first prototype, the technology “Business Rules” was introduced and discussed in detail, which is one of the key components used in our first prototype. One of the main advantages of this technology enables the separation of consistency verification logic from the application code; the other is the flexibility of defining rule action part to handle the inconsistency. Also the performance of the business rule engine is impressive. Transformation need be carried out from OCL to ILOG rule language here. Due to the different structure and semantic of these languages, one may wonder whether this mapping is applicable. Indeed, sometimes a direct mapping is not possible. But thanks to the Octopus generated Java code and the direct Java method invocation ability in ILOG rule, no information is missing after the transformation.

The second prototype is based on AOP and back navigation algorithm. A general AOP approach with poor performance was first presented. Especially for large modeling project, it is by no means acceptable. In the second version, improvement was achieved by combining the back navigation algorithm with AOP. By doing so, the pointcut and advice can all be strictly defined.

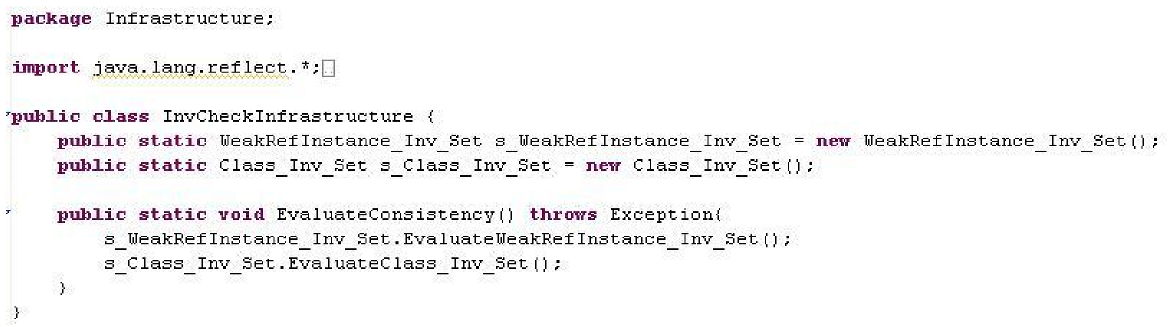
# Appendix A



**UML Diagram of Royal and Loyal**

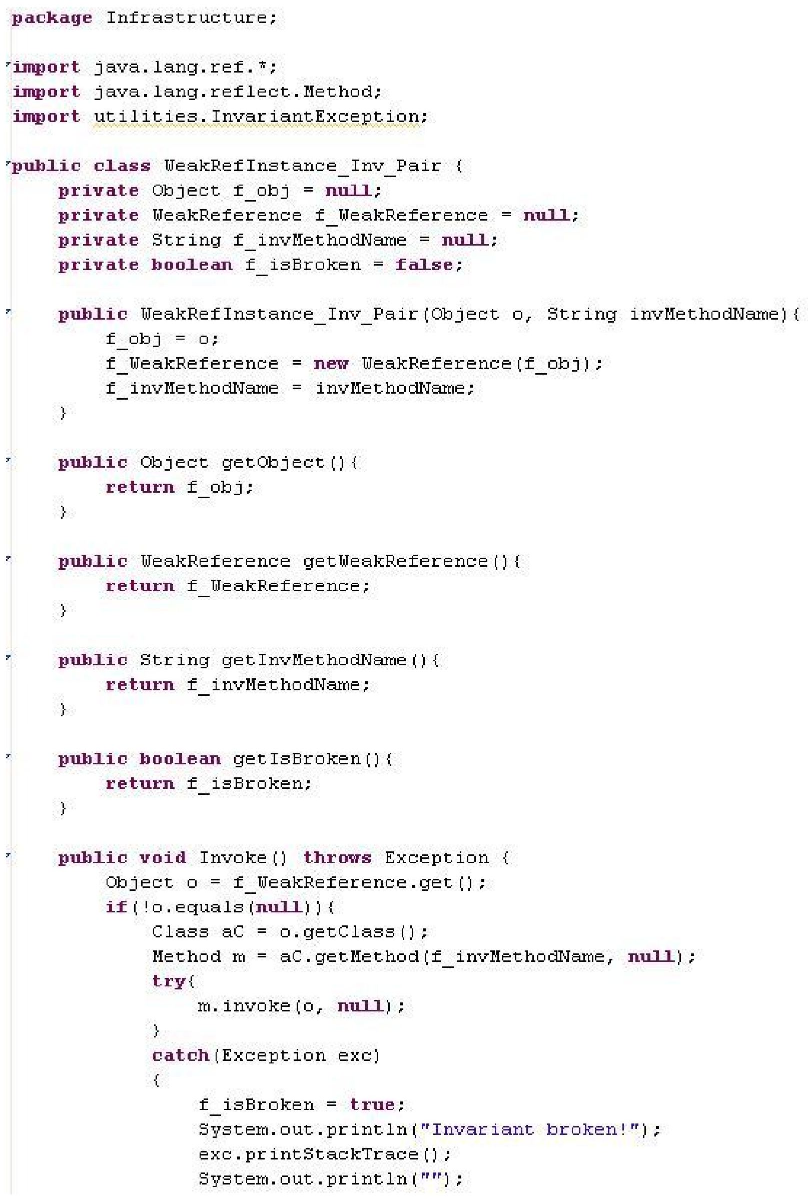
# Appendix B

Infrastructure.InvCheckInfrastructure

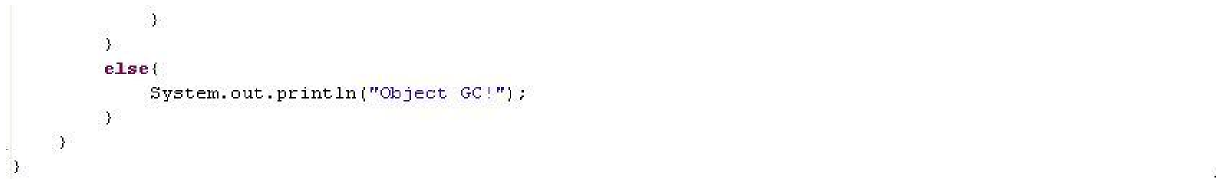


```
package Infrastructure;

import java.lang.reflect.*;

public class InvCheckInfrastructure {
    public static WeakRefInstance_Inv_Set s_WeakRefInstance_Inv_Set = new WeakRefInstance_Inv_Set();
    public static Class_Inv_Set s_Class_Inv_Set = new Class_Inv_Set();

    public static void EvaluateConsistency() throws Exception{
        s_WeakRefInstance_Inv_Set.EvaluateWeakRefInstance_Inv_Set();
        s_Class_Inv_Set.EvaluateClass_Inv_Set();
    }
}
```

Infrastructure.WeakRefInstance_Inv_Pair

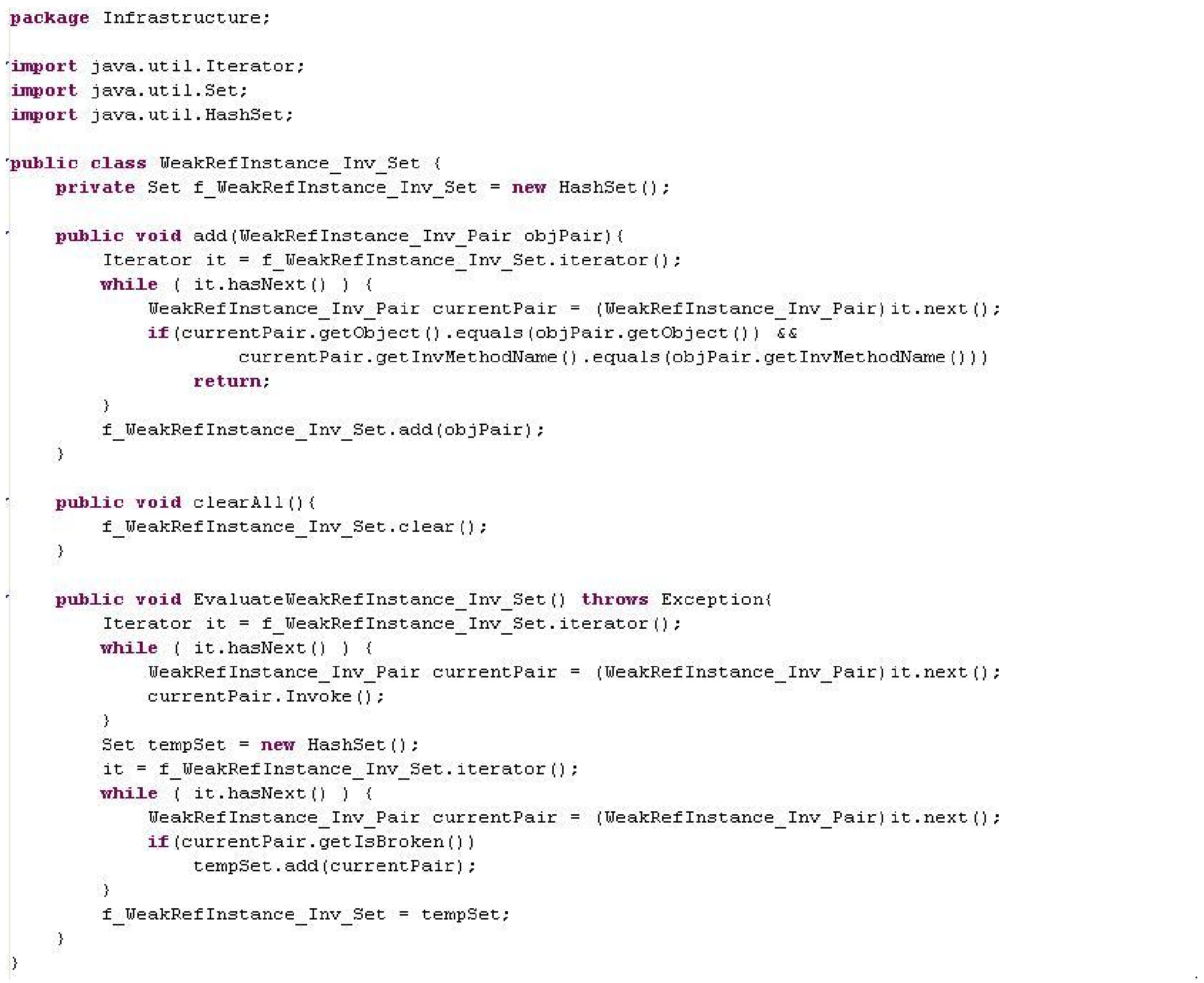


```
package Infrastructure;

import java.lang.ref.*;
import java.lang.reflect.Method;
import utilities.InvariantException;

public class WeakRefInstance_Inv_Pair {
    private Object f_obj = null;
    private WeakReference f_WeakReference = null;
    private String f_invMethodName = null;
    private boolean f_isBroken = false;

    public WeakRefInstance_Inv_Pair(Object o, String invMethodName){
        f_obj = o;
        f_WeakReference = new WeakReference(f_obj);
        f_invMethodName = invMethodName;
    }

    public Object getObject(){
        return f_obj;
    }

    public WeakReference getWeakReference(){
        return f_WeakReference;
    }

    public String getInvMethodName(){
        return f_invMethodName;
    }

    public boolean getIsBroken(){
        return f_isBroken;
    }

    public void Invoke() throws Exception {
        Object o = f_WeakReference.get();
        if(!o.equals(null)){
            Class aC = o.getClass();
            Method m = aC.getMethod(f_invMethodName, null);
            try{
                m.invoke(o, null);
            }
            catch(Exception exc)
            {
                f_isBroken = true;
                System.out.println("Invariant broken!");
                exc.printStackTrace();
                System.out.println("");
```



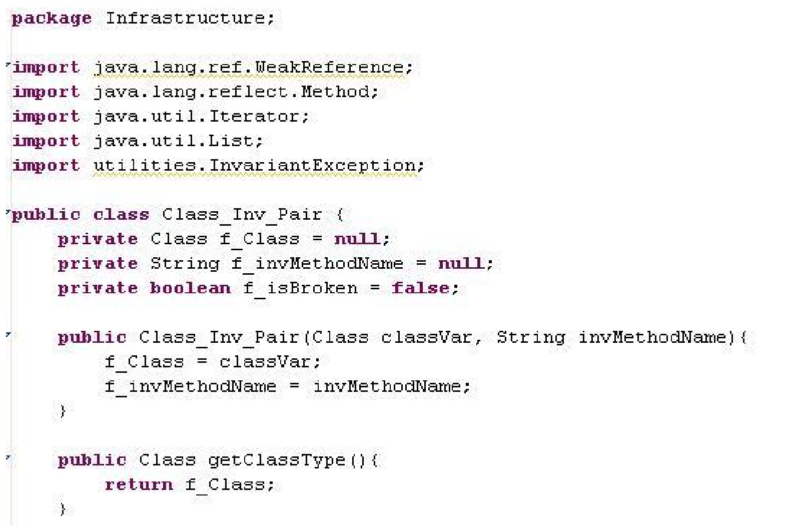
```
                }
            }
            else{
                System.out.println("Object GC!");
            }
        }
}
```

Infrastructure.WeakRefInstance_Inv_Set

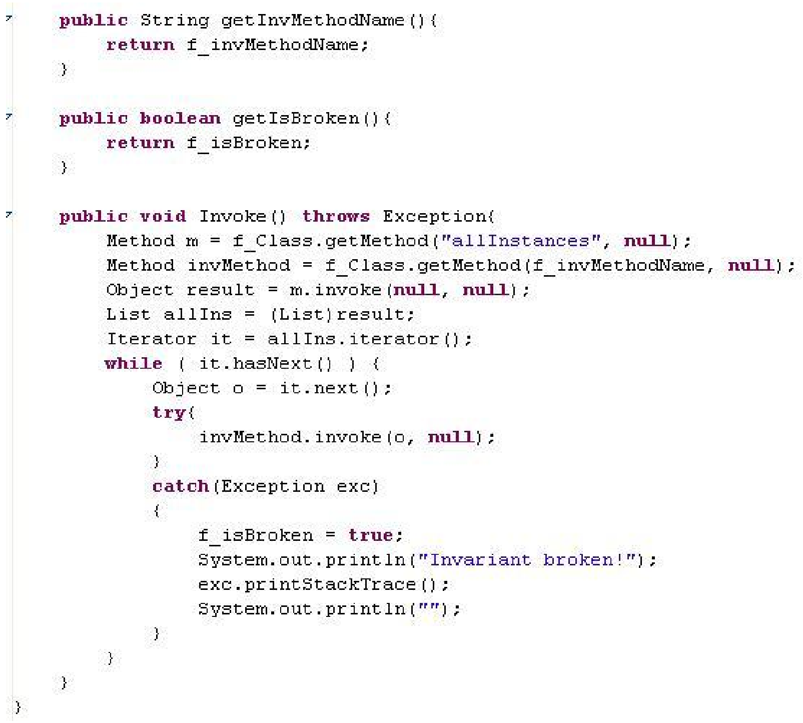


```
package Infrastructure;

import java.util.Iterator;
import java.util.Set;
import java.util.HashSet;

public class WeakRefInstance_Inv_Set {
    private Set f_WeakRefInstance_Inv_Set = new HashSet();

    public void add(WeakRefInstance_Inv_Pair objPair){
        Iterator it = f_WeakRefInstance_Inv_Set.iterator();
        while ( it.hasNext() ) {
            WeakRefInstance_Inv_Pair currentPair = (WeakRefInstance_Inv_Pair)it.next();
            if(currentPair.getObject().equals(objPair.getObject()) &&
                    currentPair.getInvMethodName().equals(objPair.getInvMethodName()))
                return;
        }
        f_WeakRefInstance_Inv_Set.add(objPair);
    }

    public void clearAll(){
        f_WeakRefInstance_Inv_Set.clear();
    }

    public void EvaluateWeakRefInstance_Inv_Set() throws Exception{
        Iterator it = f_WeakRefInstance_Inv_Set.iterator();
        while ( it.hasNext() ) {
            WeakRefInstance_Inv_Pair currentPair = (WeakRefInstance_Inv_Pair)it.next();
            currentPair.Invoke();
        }
        Set tempSet = new HashSet();
        it = f_WeakRefInstance_Inv_Set.iterator();
        while ( it.hasNext() ) {
            WeakRefInstance_Inv_Pair currentPair = (WeakRefInstance_Inv_Pair)it.next();
            if(currentPair.getIsBroken())
                tempSet.add(currentPair);
        }
        f_WeakRefInstance_Inv_Set = tempSet;
    }
}
```

Infrastructure.Class_Inv_Pair

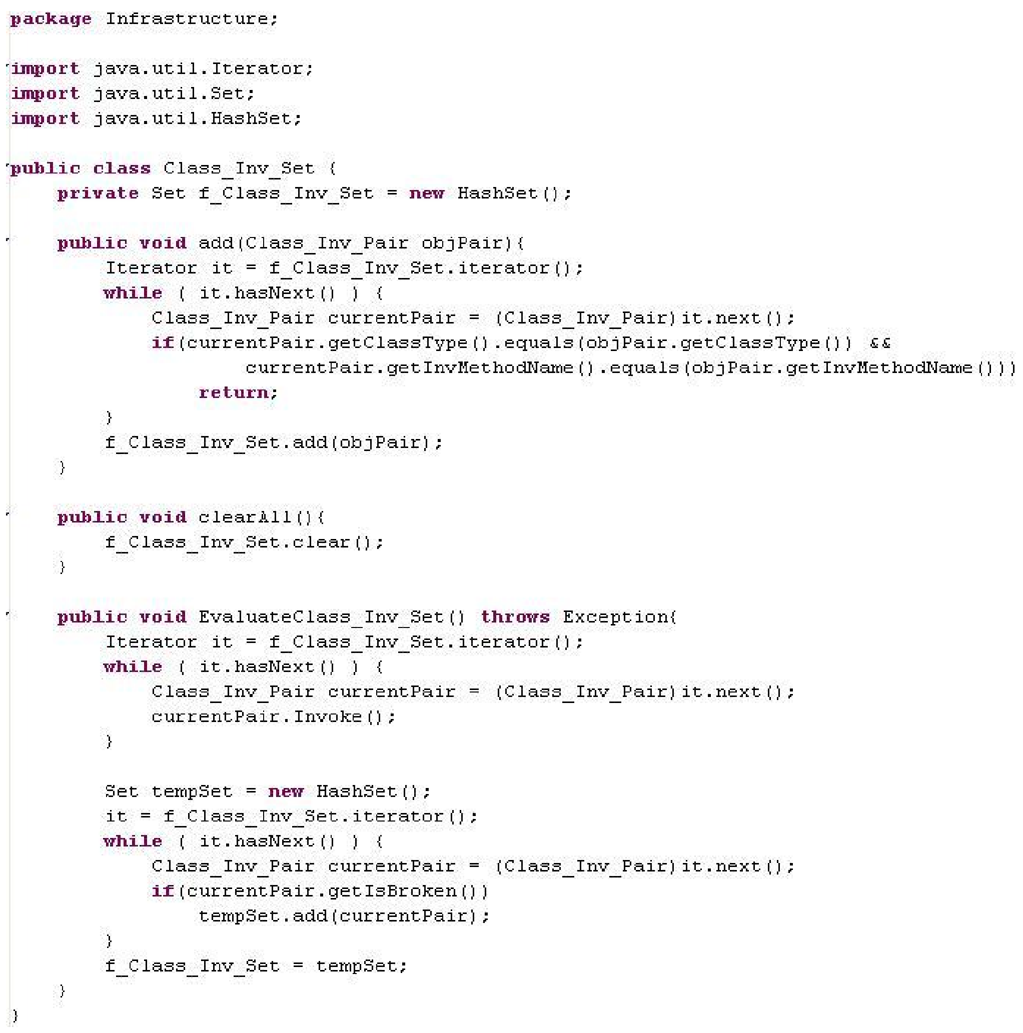


```
package Infrastructure;

import java.lang.ref.WeakReference;
import java.lang.reflect.Method;
import java.util.Iterator;
import java.util.List;
import utilities.InvariantException;

public class Class_Inv_Pair {
    private Class f_Class = null;
    private String f_invMethodName = null;
    private boolean f_isBroken = false;

    public Class_Inv_Pair(Class classVar, String invMethodName){
        f_Class = classVar;
        f_invMethodName = invMethodName;
    }

    public Class getClassType(){
        return f_Class;
    }
```



```java
    public String getInvMethodName(){
        return f_invMethodName;
    }

    public boolean getIsBroken(){
        return f_isBroken;
    }

    public void Invoke() throws Exception{
        Method m = f_Class.getMethod("allInstances", null);
        Method invMethod = f_Class.getMethod(f_invMethodName, null);
        Object result = m.invoke(null, null);
        List allIns = (List)result;
        Iterator it = allIns.iterator();
        while ( it.hasNext() ) {
            Object o = it.next();
            try{
                invMethod.invoke(o, null);
            }
            catch(Exception exc)
            {
                f_isBroken = true;
                System.out.println("Invariant broken!");
                exc.printStackTrace();
                System.out.println("");
            }
        }
    }
}
```

Infrastructure.Class_Inv_Set



```java
package Infrastructure;

import java.util.Iterator;
import java.util.Set;
import java.util.HashSet;

public class Class_Inv_Set {
    private Set f_Class_Inv_Set = new HashSet();

    public void add(Class_Inv_Pair objPair){
        Iterator it = f_Class_Inv_Set.iterator();
        while ( it.hasNext() ) {
            Class_Inv_Pair currentPair = (Class_Inv_Pair)it.next();
            if(currentPair.getClassType().equals(objPair.getClassType()) &&
                    currentPair.getInvMethodName().equals(objPair.getInvMethodName()))
                return;
        }
        f_Class_Inv_Set.add(objPair);
    }

    public void clearAll(){
        f_Class_Inv_Set.clear();
    }

    public void EvaluateClass_Inv_Set() throws Exception{
        Iterator it = f_Class_Inv_Set.iterator();
        while ( it.hasNext() ) {
            Class_Inv_Pair currentPair = (Class_Inv_Pair)it.next();
            currentPair.Invoke();
        }

        Set tempSet = new HashSet();
        it = f_Class_Inv_Set.iterator();
        while ( it.hasNext() ) {
            Class_Inv_Pair currentPair = (Class_Inv_Pair)it.next();
            if(currentPair.getIsBroken())
                tempSet.add(currentPair);
        }
        f_Class_Inv_Set = tempSet;
    }
}
```

# References

[1] Octopus Official Website
URL: http://www.klasse.nl/octopus/index.html

[2] Octopus Documentation
URL: http://www.klasse.nl/octopus/octopus-developer-pack.zip

[3] ATL Official Website
URL: http://www.eclipse.org/gmt/atl/

[4] ILOG Business Rules Official Website
URL: http://www.ilog.com/products/jrules/

[5] AspectJ Development Tools (AJDT) Official Website
URL: http://www.eclipse.org/ajdt/

[6] Object Management Group, Model Driven Architecture Official Website
URL: http://www.omg.org/mda/

[Xue05] Meng Xue, OCL Engine, Technical University Hamburg Harburg, 2005, http://www.sts.tu-harburg.de/pw-and-m-theses/papers.html/

[Jouault05] Jean Bezivin, Frederic Jouault, Using ATL for Checking Models, University of Nantes, 2005, http://tfs.cs.tu-berlin.de/gramot/FinalVersions/PDF/BezivinJouault.pdf/

[Warmer03]Bast, W., A. Kleppe, and J. Warmer, MDA Explained: The Model Driven Architecture: Practice and Promise, Addison Wesley, 2003.

[Akehurst01] D. H. Akehurst and B. Bordbar, On Querying UML Data Models with OCL, <<UML>> - The Unified Modeling Language, Modeling Languages, Concepts and Tools, 4th International Conference, Toronto, Canada, 2001.

[Booch99] Grady Booch, James Rumbaugh, and Ivar Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[Eriksson00] Hans-Erik Eriksson and Magnus Penker, Business Modeling with UML, Business Patterns at Work, John Wiley & Sons, 2000.

[Kleppe03] Anneke Kleppe, Jos Warmer, and Wim Bast, MDA Explained; The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003.

[Kleppe03] Jos Warmer, Anneke Kleppe, Object Constraint Language, Getting Your Models Ready for MDA, Second Edition, Addison Wesley, 2003.

[OCL03] Response to the UML 2.0 OCL RfP, revision 1.6, OMG document ad 2003-01-06.

[Richters01] Mark Richters, A Precise Approach to Validating UML Models and OCL Constraints, Logos Verlag Berlin, 2001.

[Rumbaugh99] James Rumbaugh, Grady Booch, and Ivar Jacobson, Unified Modeling Language Reference Manual, Addison-Wesley, 1999.

[Arthorne04] John Arthorne, Chris Laffra, Official Eclipse 3.0 Faqs, Addison-Wesley, 2004.

[Gamma03] Erich Gamma,Kent Beck, Contributing to Eclipse: Principles, Patterns, and Plug-Ins, Addison-Wesley, 2003.

[Richters01] Mark Richters and Martin Gogolla, OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, Advances in Object Modelling with the OCL, pages 43–69. Springer, Berlin, 2001.

[Kiczales01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten,J. Palm, and W. G. Griswold, An overview of AspectJ, Springer-Verlag, June 2001.

[Holzner04] Steve Holzner, Eclipse Cookbook, O'Reilly, 2004.

[Gallardo03] David Gallardo, Ed Burnette, Robert McGovern, Eclipse in ActionA Gudie For Java Developers, Manning, 2003.

[Colyer04] Adrian Colyer, Andy Clement, George Harley, Matthew Webster, Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Addison Wesley, 2004.

[Briand04] L. C. Briand, W. Dzidek, Y. Labiche, Using Aspect-Oriented Programming to Instrument OCL Contracts in Java, Technical Report SCE-04-03, Carleton University, 2004 http://www.sce.carleton.ca/Squall/pubs/tech_report/TR_SCE-04-03.pdf