Leopold-Franzens-University
Innsbruck

Institute of Computer Science
**Quality Engineering**

# Eclipse Advanced OCL Editor

**Bachelor Thesis**

Supervisor: Ph.D. Eng. Joanna Chimiak-Opoka

**Arslan Ekrem**

SS 2008

Innsbruck, December 16, 2008

# Abstract

Nowadays the usage of Unified Modeling Language (UML) is becoming more popular in industrial projects. For quality assurance and the assessment of large scale models, constraints and queries are used. The Object Constraint Language (OCL) is suitable for accomplishing this task. However, practitioners find it difficult to define OCL expressions. Thus, there is a need for simplification mechanisms to promote the use of OCL. One simplification mechanism is the concept of OCL libraries for model queries, which simplify the definition of OCL expressions by hiding definition details from the end user. Moreover, these libraries lead to a high reuse factor, they are configurable and testable and they are documented. The libraries can be checked for syntax correctness and can be tested against a desired semantics.

In this thesis an Advanced OCL Editor is presented supporting features related to different aspects of the OCL library definition, such as *syntactical correctness* (syntax checking, syntax completion, usage of templates), *semantical correctness* (testing), *readability* (syntax and error highlighting, code formatting), *reusability* (comments, documentation).

# Kurzfassung

Die Unified Modeling Language (UML) wird heutzutage vermehrt in der Industrie eingesetzt. Für die Qualitätssicherung und Beurteilung von umfangreichen Modellen werden Constraints (Zusicherungen) und Queries (Anfragen) verwendet. Die Object Constraint Language (OCL) ist ein geeignetes Werkzeug dafür. Jedoch wird das Definieren von OCL-Ausdrücken in der Industrie als zu schwierig angesehen. Deshalb sind Möglichkeiten gefragt, um diese Aufgabe zu vereinfachen, damit OCL vermehrt eingesetzt wird. Eine Möglichkeit ist der Einsatz von OCL-Programmbibliotheken, die die Details eines OCL-Ausdrucks und somit die Komplexität vom Endanwender verbergen. Darüber hinaus ermöglichen die Programmbibliotheken die Wiederverwendung von OCL-Ausdrücken. Sie sind konfigurierbar, können getestet werden und sind dokumentiert. Die Programmbibliotheken können auf ihre Syntax überprüft sowie auf die gewünschte Semantik getestet werden.

In dieser Bakkalaureatsarbeit soll der Advanced OCL Editor vorgestellt werden, der das Erstellen von OCL-Programmbibliotheken in vielerlei Hinsicht unterstützt, wie zum Beispiel *Syntax* (Syntaxüberprüfung, Syntaxvervollständigung, Verwendung von Vorlagen), *Semantik* (Testen), *Lesbarkeit* (Syntax- und Fehlerhervorhebung, Code-Formatierung), *Wiederverwendbarkeit* (Kommentare, Dokumentation).

# Contents


*1. Introduction ........ 1*

**1.1. Background ........ 1**

**1.2. Motivation ........ 1**

*2. Requirements Specification ........ 3*

**2.1. Overview ........ 3**

**2.2. Functional Requirements ........ 4**

**2.3. Technical Requirements ........ 5**

**2.4. Class Diagram ........ 6**

**2.5. Description of the Class Diagram ........ 6**

2.5.1 Visibility ........ 7

**2.6. Use Cases ........ 8**

*3. Architecture and Context ........ 19*

**3.1. Design Paradigms ........ 19**

3.1.1 Three-Tier Architecture ........ 20

3.1.2 Model-View-Controller ........ 21

3.1.3 Design Patterns ........ 22

3.1.4 Creational Pattern – Singleton ........ 22

3.1.5 Behavioral Pattern – Observer Pattern ........ 23

3.1.6 Behavioral Pattern – Visitor Pattern ........ 24

**3.2. Design Overview and Context ........ 25**

3.2.1 Component Diagram ........ 25

3.2.2 Library Interpretation Process ........ 26

3.2.3 Advanced OCL Editor in the Context of SQUAM ........ 27

**3.3. Design Details ........ 28**

3.3.1 Class Diagram ........ 29

3.3.2 Presentation Layer ........ 30
3.3.3 Application Layer ........ 36
***4. Implementation ........ 40***
**4.1. Java ........ 41**
**4.2. Eclipse ........ 42**
4.2.1 Eclipse IDE ........ 42
4.2.2 Eclipse Rich Client Platform ........ 42
4.2.3 Eclipse OCL ........ 44
**4.3. ANTLR Parser Generator and ANTLRWorks ........ 45**
**4.4. User Manual ........ 45**
***5. Project Plan ........ 46***
***6. List of Figures ........ 49***
***7. List of Tables ........ 50***
***8. List of Literature ........ 51***

Chapter 1

# Introduction

## 1.1. Background

The Unified Modeling Language (UML, [1]) is a standardized general-purpose modeling language for the object-oriented analysis and design of systems [2]. Nowadays UML is also increasingly used in industrial projects in large scale models. Along with the growing size of UML models the need for model constraints, model querying, and their automation has come up. The support of model constraints is required to assure the correctness of models in application domains. The support of model querying is required for an automated or semi-automated model inspection.

The Object Constraint Language (OCL, [3]) is a formal language used to specify constraints for model elements in UML models. OCL's ability to navigate the model and form collections of objects has contributed to its usage as a query language. Based on these properties, OCL can be used for querying large scale models [4].

## 1.2. Motivation

Although OCL is an expressive formal language and applicable for the tasks described in 1.1, it is not widely used as practitioners find it difficult to define OCL expressions. Thus, there is a need for simplification mechanisms to promote the use of OCL. A possible simplification mechanism is the concept of OCL libraries for model queries described in [4]. This concept simplifies the definition of OCL

expressions by hiding definition details from the end user. Moreover, the libraries lead to a high reuse factor. They are configurable, testable, and documented. Furthermore, the support of tools can simplify the definition of OCL expressions. Although there are tools that support OCL related activities, they do not focus on writing OCL expressions. The combination of the concept of OCL libraries and a user friendly editor supporting OCL expressions as well as OCL libraries has resulted in the idea of an Advanced OCL Editor.

The goal of the Advanced OCL Editor is to simplify the writing of OCL expressions. It should support the end user in writing, testing, and documenting OCL libraries. While the conceptual aspect of OCL libraries is beyond the scope of this thesis and can be found in [4] (see Appendix B), this thesis describes the requirements (Chapter 2), the architecture (Chapter 3), and the implementation (Chapter 4) of the Advanced OCL Editor. In addition, a brief overview of the project plan is given (Chapter 5).

Chapter 2

# Requirements Specification

## 2.1. Overview

The goal of the Advanced OCL Editor is to make writing OCL code easier and therefore an error-free and less time-consuming task. The textual notation of OCL makes it similar to programming languages. The tasks and problems related to programming in OCL are the same as those related to other programming languages. Therefore, mechanisms typically used in programming languages to deal with problems are also appropriate for the Advanced OCL Editor for which the following requirements have been defined:

- **F1: Syntactical correctness** is required to make writing an OCL code syntactically error-free. In general, the syntactical correctness is the basic property required for any source code as well as the prerequisite for an interpretation of the OCL code.
- **F2: Semantical correctness** is required for a semantically error-free writing of an OCL code. Semantical correctness can be assured by formal verification or empirical investigation.

- **F3: Readability** refers to the ease with which a reader can comprehend the OCL code. Some factors enhancing its readability are coding rules, different indentations styles, decomposition, and comments.
- **F4: Reusability** can help to write an OCL code more efficiently. Reusable modules reduce the implementation time and increase the likelihood that the prior testing and use have eliminated bugs. Thus, reusability improves the semantical correctness.
- **F5: Understandability** is important for reusing and exchanging an OCL code. Besides readability, also documentation makes a code easier to understand.

## 2.2. Functional Requirements

A set of required features for an Advanced OCL Editor has been determined. These features are related to the requirements described above:

- **Syntax checking** increases the *syntactical correctness* (F1) and the user satisfaction. This feature is necessary for any language editor. In the context of the Advanced OCL Editor, syntax checking ensures syntactical correctness against the OCL grammar and the underlying meta-model.
- **Syntax completion** is a feature which increases the user satisfaction, the efficiency, and the *syntactical correctness* (F1) of OCL expressions. The Advanced OCL Editor should support standard OCL and OCL libraries.
- **Templates** increase the user satisfaction, the efficiency, and the *syntactical correctness* (F1) of OCL expressions. Templates can be defined for concepts related to OCL libraries.
- **Testability** increases the *semantical correctness* (F2) of OCL expressions. A sufficient number of unit tests ensure the correctness of an incremental development of OCL expressions and libraries respectively. These tests are similar to unit tests provided for many programming languages.
- **Syntax and error highlighting** contributes to the efficiency, a better *readability* (F3), and user satisfaction and it enables a faster error correction.
- **Code formatter** enables a better *readability* (F3) and the exchange of OCL expressions and libraries respectively.
- **Documentation and comments** is important for sharing OCL expressions and libraries respectively as well as for an easier *understanding* (F5), *readability* (F3), and *reusability* (F4). It covers writing documentation comments and the documentation generation.

## 2.3. Technical Requirements

The Advanced OCL Editor should meet the following technical requirements:

- The Advanced OCL Editor should be available in two versions:
  - Plug-in for Eclipse [5]
  - Rich Client Platform stand-alone application [6]
- Library storage in file system/Subversion [7]
- Static meta-model binding for a library and test models
- Library import: reuse of existing OCL libraries; import mechanism like in Java [8]
- Managing files as a project: a library is stored in a file; a project consists of a number of files
- Tree representation of a library:
  - `library name`
    - `meta-model`
    - `import declarations`
      - `required library 1`
      - …
    - `definitions`
      - `context::definition name 1`
      - …
    - `queries`
      - `context::query name 2`
      - …
    - `tests`
      - `model name 1`
        - `test name 1`
        - …
      - `model name 2`
        - `test name 2`
        - …
      - …
- List of problems: a list of all problems, warnings, and errors (e.g. syntax errors)

## 2.4. Class Diagram

The conceptual class diagram (Figure 1) shows the main building blocks of the system.

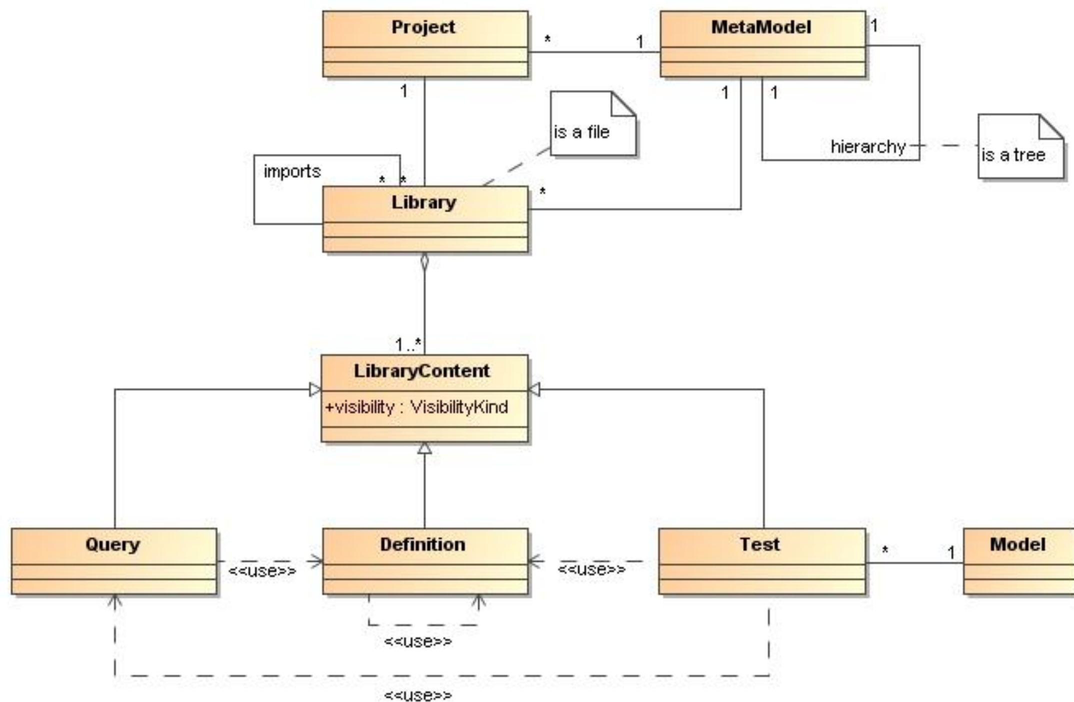


Figure 1 Conceptual Class Diagram

## 2.5. Description of the Class Diagram

In this section, the elements are described which are needed to meet the requirements discussed in the previous sections. The relations between the elements are illustrated in the conceptual class diagram (Figure 1).

- **Project**: A *library* is stored in a file. A number of libraries (files) is managed as project. A project has an associated *meta-model*. A project simplifies the maintenance of libraries.
- **MetaModel**: A meta-model is assigned to *projects* and *libraries* and specifies the scope of the application of the libraries which can be used for models as instances of the meta-model.
- **Library**: A library is specific to a given *meta-model*. It can import other libraries and consists of some non-empty content. The import mechanism enables the usage of expressions defined in other libraries. The visibility of

imported expressions depends on their visibility of the *library content* in which the expressions are defined. Libraries are stored in files.

- **LibraryContent**: There are three different types of library content: *definitions*, *tests*, and *queries*. The visibility of the library content is specified by VisibilityKind defined in the UML 2.0 Superstructure [1]. The visibilities are described in the next section.
- **Query**: The queries are a collection of OCL methods dedicated to the usage for model querying in a modeling or an analysis tool.
- **Definition**: The definitions are a collection of OCL method and attribute definitions to be used in other OCL expressions in the same or in another library. Moreover, they should be tested and accessible in the testing libraries.
- **Test**: The tests are test case definitions for OCL expressions, in particular definitions. For the evaluation of tests, a concrete user model is required.
- **Model**: For tests, a model is required. The model has to be an instance of the meta-model. Moreover, the meta-model specified in the testing library has to be the same as in the corresponding tested library.

### 2.5.1 Visibility

There are four visibility kinds defined in the UML 2.0 Superstructure [1]: private, protected, package, and public. The interpretations of the public and private elements are the same as in the UML specification. The interpretation of protected elements is modified, while the package visibility is not used. A private expression is only visible inside the library which owns it. A public or protected expression is visible in other libraries which import the one that owns it. Moreover, a public expression is visible in tools that use the library. Table 1 illustrates the possible combinations of visibilities for an external use, in exporting libraries, and within the same library.

Concerning the visibility by default, queries are public, definitions are protected, and tests are private since queries should be used in modeling tools, definitions should be tested and accessible in the testing libraries or used in another libraries, and tests are not intended to be reused elsewhere.

| scope / visibility kind | external | exported | same |
|---|---|---|---|
| public | true | true | true |
| protected | false | true | true |
| private | false | false | true |

Table 1 Possible Combinations of Visibilities

## 2.6. Use Cases

In this section, different use cases are discussed. Figure 2 shows an overview of the use cases followed by detailed textual descriptions thereof. Trivial use cases and use cases described by the included use case are not discussed separately. These use cases (Project modification, Project storage, OCL library modification, Edit OCL queries, Edit OCL definitions, Edit OCL tests, Edit OCL documentation) are in orange. The use cases “OCL documentation export” and “Edit OCL documentation” have been implemented by Hannes Mösl and are not part of this thesis. Nevertheless, these use cases are illustrated and described to provide a complete overview.

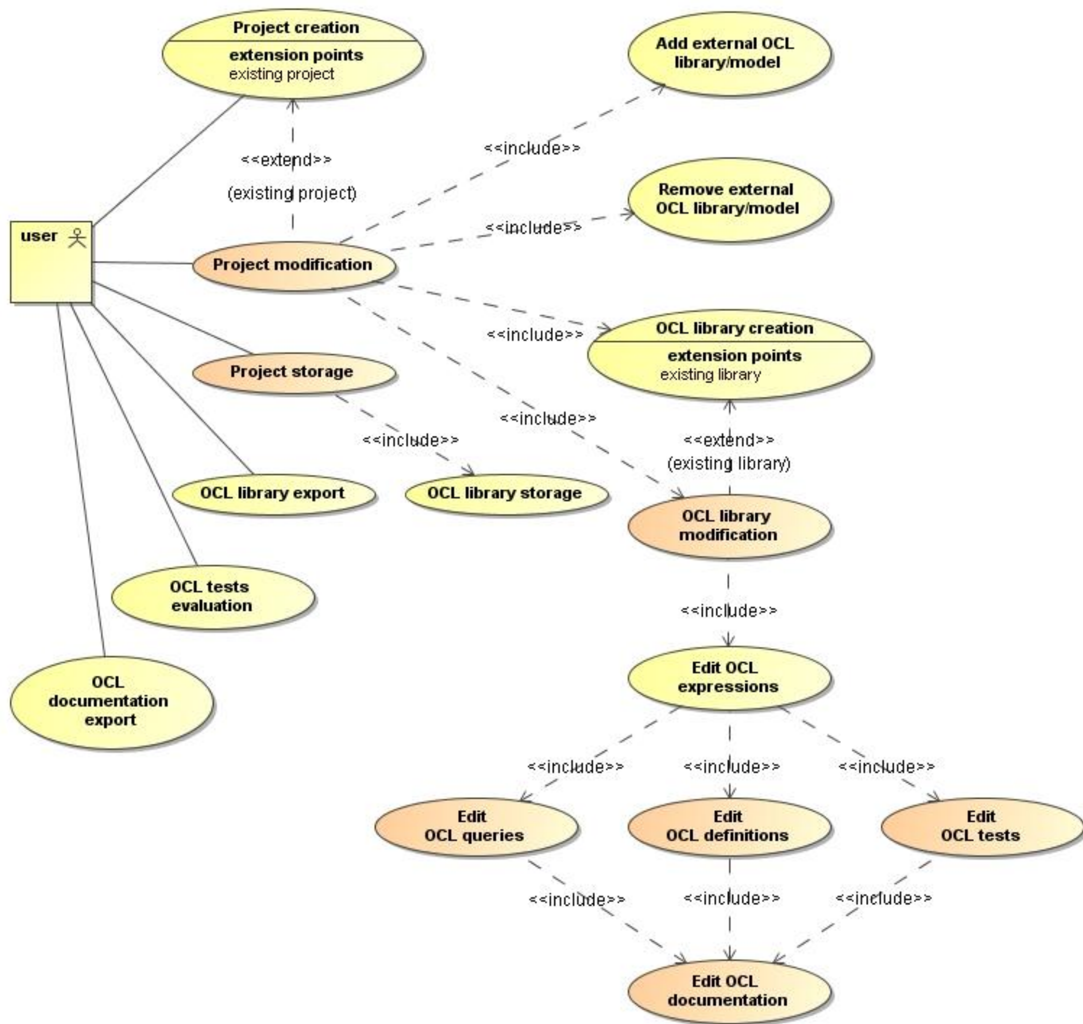


Figure 2 Use Case Diagram

| **Project creation** | |
|---|---|
| Summary: | The user wants to create a new OCL library project. The user names the project, specifies a package name, and selects a meta-model. |
| Actors: | User |
| Preconditions: | |
| Basic flow: | 1. The user initiates a new OCL library project.<br>2. The user names the project and selects the storage location.<br>3. The user specifies a package name.<br>4. The user defines the meta-model and the modeling level.<br>5. The system creates the project, folders for OCL libraries and user models, and an initial user model. |
| Postconditions: | The names of the OCL library project and package are unique. |
| Alternative flows: | 2a. *The project already exists.*<br>The user receives an error message.<br>3a. *The package name already exists.*<br>The user receives an error message.<br>5a. *The user cancels the project creation.*<br>All creation data are discarded. The file system is not modified for the project. |

| **Add external OCL library/model** | |
|---|---|
| Summary: | The user wants to (re)use an external OCL library/model. |
| Actors: | User |
| Preconditions: | The user has to have a configured OCL library project and an external OCL library/user model. |
| Basic flow: | 1. The user selects the OCL library project.<br>2. The user opens the properties of the OCL library project.<br>3. The user selects the external OCL library/model. For models, the user has to additionally select a unique name that will be used to identify the model.<br>4. The system associates the external OCL library/model with the selected project. |
| Postconditions: | The names of the OCL library and model are unique. |
| Alternative flows: | 3a. *The name of the OCL library/model has already been added.*<br>The user receives an error message.<br>4a. *The user cancels the operation.*<br>The external resource is not associated with the project. |

| **Remove external OCL library/model** | |
|---|---|
| Summary: | The user wants to remove an association with an external OCL library/model from the project. |
| Actors: | User |
| Preconditions: | The user has to have a configured OCL library project and an associated external OCL library/user model. |
| Basic flow: | 1. The user selects the OCL library project.<br>2. The user opens the properties of the OCL library project.<br>3. The user removes the associated external OCL library/model from the project.<br>4. The system removes the association with the external OCL library/model from the selected project. |
| Postconditions: | |
| Alternative flows: | 4a. *The user cancels the operation.*<br>The association with the external resource is not removed from the project. |

| **OCL library creation** | |
|---|---|
| Summary: | The user wants to create a new OCL library. |
| Actors: | User |
| Preconditions: | The user has to have a configured OCL library project. |
| Basic flow: | 1. The user initiates a new OCL library.<br>2. The user names the library and selects the storage location.<br>3. The system creates a new OCL library file. |
| Postconditions: | The names of OCL libraries are unique. |
| Alternative flows: | 2a. *The OCL library already exists.*<br>The user receives an error message.<br>3a. *The user cancels the OCL library creation.*<br>All creation data are discarded. The file system is not modified. |

| Edit OCL expressions | |
| --- | --- |
| Summary: | The user wants to create or modify the library content (OCL definitions, OCL queries, or OCL tests). |
| Actors: | User |
| Preconditions: | The user has to have a configured OCL library project and an OCL library. |
| Basic flow: | 1. The user opens an OCL library.<br>2. The user inserts changes or deletes a certain library content.<br>3. The user saves the changes. (see Use-Case Description: OCL library storage ) |
| Postconditions: | |
| Alternative flows: | 3a. *The user closes the OCL library without saving it*. All changes are discarded. The corresponding file is not modified. |

| **OCL library storage** | |
|---|---|
| Summary: | The user wants to save library modifications. |
| Actors: | User |
| Preconditions: | The user has to have a configured OCL library project and modified OCL libraries. |
| Basic flow: | 1. The user saves the modified OCL library.<br>2. The system checks the syntactical correctness of the OCL library.<br>3. The system shows all errors and problems that have occurred in the list of problems. Moreover, the system marks the problems in the library editing window. All representations of the library (e.g. the tree representation) are also marked.<br>4. The system refreshes the tree representation of the library.<br>5. The system applies the changes to the corresponding file. |
| Postconditions: | Saved modifications are stored in the file system. |
| Alternative flows: | 3a. *The OCL library is syntactically correct.*<br>All previous warnings and problems concerning the corresponding OCL library are removed from the list of problems. Possibly existing error markers are removed.<br>4a. *The tree representation is not available.*<br>An empty tree marked with a question mark is displayed. |

| **OCL library export** | |
|---|---|
| Summary: | The user wants to export OCL libraries to reuse or exchange them. |
| Actors: | User |
| Preconditions: | The user has to have a configured OCL library project. |
| Basic flow: | 1. The user starts the OCL library export.<br>2. The user selects the OCL libraries he wants to export and specifies the export destination.<br>3. The system creates a JAR file including the selected libraries and an index of the required libraries which are not in the JAR file. |
| Postconditions: | The names of the libraries in the exported JAR file are unique. |
| Alternative flows: | 2a. *The user selects two libraries with identical names.* The user receives an error message.<br>3a. *The user cancels the OCL library export*. All creation data are discarded. The file system is not modified. |

| **OCL tests evaluation** | |
|---|---|
| Summary: | The user wants to evaluate OCL tests to ensure the syntactical correctness of OCL libraries. The user can evaluate tests on different levels: project, library, a test model, or a single test. |
| Actors: | User |
| Preconditions: | The user has to have a configured OCL library project and at least one library with OCL tests. |
| Basic flow: | 1. The user selects the project for which he wants to run all tests.<br>2. The user starts the test evaluation.<br>3. The system illustrates which tests have passed and which ones have failed. |
| Postconditions: | |
| Alternative flows: | 1a. *The user selects the desired test evaluation level (library, test model, or a single test) from the tree representation.*<br>All included tests are evaluated. |

| **OCL documentation export** | |
|---|---|
| Summary: | The user wants to generate an HTML documentation of the OCL libraries. |
| Actors: | User |
| Preconditions: | The user has to have a configured OCL library project and at least one library with an OCL documentation. |
| Basic flow: | 1. The user starts the OCL documentation export.<br>2. The user selects the OCL libraries of which he wants to generate an HTML documentation and specifies the export location.<br>3. The user defines the visibilities for which he wants to generate an HTML documentation. The members of the selected visibilities will be exported.<br>4. The system generates an HTML documentation of the selected OCL libraries and an index. |
| Postconditions: | |
| Alternative flows: | 4a. *The user cancels the OCL documentation export.* All creation data are discarded. The file system is not modified. |

Chapter 3

# Architecture and Context

The software architecture of a software system refers to the structure(s) of the system which comprise(s) software elements, the externally visible properties of these elements, and the relationships and constraints between them [9]. Hence, the software architecture structures the software system into elements with specified responsibilities. Furthermore, the quality of the entire system can be determined on the basis of the quality of these elements. Moreover, the software architecture defines interfaces for future extensions [2].

## 3.1. Design Paradigms

In this section, the general architectural and design pattern used for the Advanced OCL Editor are introduced. At first, two architectural patterns are presented: the three-tier architecture (Section 3.1.1) and the model-view-controller architecture (Section 3.1.2). Then, a brief overview of the design pattern is given (Section 3.1.3) and the Singleton pattern (Section 3.1.4), the Observer pattern (Section 3.1.5), and the Visitor pattern (Section 3.1.6) are described in detail.

### 3.1.1 Three-Tier Architecture

The three-tier architecture separates a software system into three interacting layers of which each has its own specific responsibilities (Figure 3):

- The **presentation layer** is responsible for the graphical representation of an application.
- The **application layer** implements the business logic.
- The **data layer** represents data and provides an interface to a database, if the system uses one.

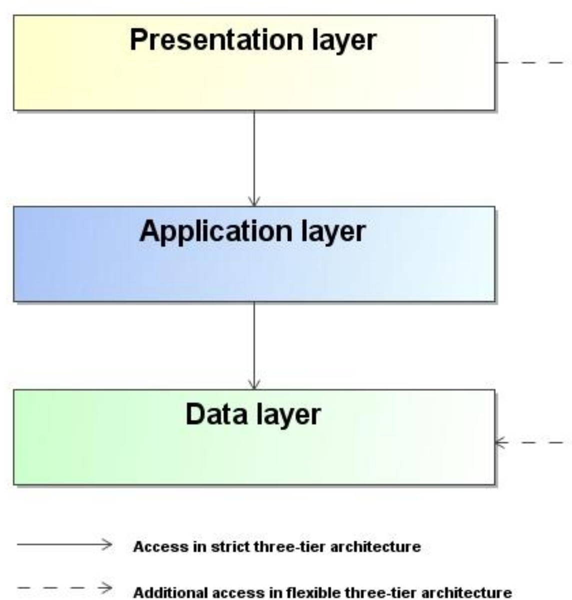


Figure 3 Three-Tier Architecture

A three-tier architecture meets the fundamental requirements of a layered architecture; this means that elements may access only elements of their own layer or a lower one [10]. Here two specifications are possible:

- **Strict three-tier architecture**: The presentation layer may only access the application layer. The advantage of this architecture is that the presentation layer only depends on the application layer. Hence, the presentation layer is completely independent from the data layer.
- **Flexible three-tier architecture**: The presentation layer may not only access the application layer but also the data layer. This architecture is more flexible and has a better performance, but the maintainability, changeability, and portability of such an architecture are lower [11].

### 3.1.2 Model-View-Controller

A fundamental idea of a layered architecture is that the representation layer (user interface) is not directly accessed by another layer. Today the isolation of the user interface and the business logic is a basic principle of software design. Many of these ideas have their origin in the Model-View-Controller (MVC) architecture [11]. MVC considers three roles:

- The **model** is an object that represents certain information on domain.
- The **view** represents the display of the model in the user interface.
- The **controller** takes the user input, invokes changes on the model, and causes the view to update appropriately [12].

Figure 4 illustrates the relationship between these three roles. The solid lines represent a direct association, while the dashed lines indicate an indirect association (e.g. observer pattern). The model has no knowledge of the view and may not directly access the associated view and controller objects. In case of an update, it has to notify all dependent objects about the changes. This consistent isolation enables an independent development of the user interface and the business logic. Moreover, it improves the exchangeability of the user interface [11].

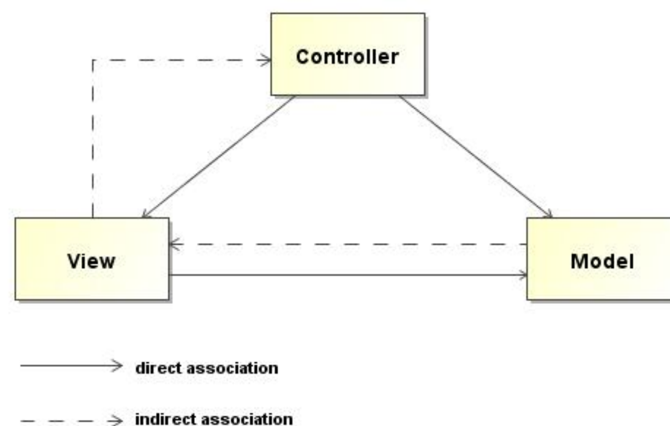


Figure 4 Model-View-Controller

### 3.1.3 Design Patterns

A design pattern is an approved and generic solution to a commonly occurring problem in software design. Design patterns are grouped into the following categories:

- **Creational patterns** create objects without a direct object instantiation.
- **Structural patterns** describe how classes and objects can be combined into larger structures.
- **Behavioral patterns** deal with the communication between the objects of a system [13].

In the following sections, the creational and behavioral patterns used in the Advanced OCL Editor are described.

### 3.1.4 Creational Pattern – Singleton

The singleton pattern restricts the instantiation of a class to one object and enables a public access to this object. This pattern is useful when exactly one easily accessible object is needed [13]. The implementation of a singleton pattern is presented in the following code snippet:

```
public class Singleton {
	private static Singleton instance = null;
	private Singleton() {}
	public static Singleton getInstance() {
		if(instance == null)
			instance = new Singleton()
		return instance;
	}
}
```

The singleton pattern is implemented by creating a class with a method (`getInstance()`) that creates a new instance of the class if it does not yet exist. If an instance (`instance`) already exists, it returns a reference to that object. To make sure that the object cannot be instantiated in any other way, the constructor (`Singleton()`) is made private.

The singleton pattern is realized in the following classes (described in Section 3.3):

- `Activator`
- `MetaModelManager`
- `OCLLibraryManager`
- `OCLModelManager`

### 3.1.5 Behavioral Pattern – Observer Pattern

The observer pattern consists of a `subject` object, which contains data that may change over the time, and a number of `observer` objects. If the data changes, the `subject` object automatically notifies the `observer` objects, usually by calling one of their methods [14]. There are two communication methods for this process:

- **Push model**: The `subject` sends detailed information on the change to the `observer` whether it wants it or not.
- **Pull model**: The `subject` only notifies the `observer` if a change in its state appears, while it is the responsibility of the `observer` to get the required data from the `subject` [15].

The choice of the communication method and the resulting efficiency depend on the application area. The push model can be inefficient if a large amount of data needs to be sent, whereas the pull model can be ineffective since the communication is done in two steps and problems might appear in multi-threading environments [16]. The common communication method in the Advanced OCL Editor is the push model.

The observer pattern is very often associated with the MVC paradigm. In MVC, the observer pattern is used to create a loose coupling between the model and the view [14]. Moreover, the observer pattern is used for event management. Hence, this pattern is used in various view elements which are based on the MVC architecture. Some classes (described in Section 3.3) using the observer pattern are:

- `OCLContentOutlinePage`
- `OCLDocumentModel`
- `OCLEditor`

### 3.1.6 Behavioral Pattern – Visitor Pattern

The visitor pattern separates an algorithm from an object structure upon which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying these structures. In essence, the visitor (`Visitor`) allows adding new virtual functions to a family of classes (`Element`) without modifying the classes themselves; instead, a visitor class (`ConcreteVisitor`) is created which implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as an input and implements the goal by means of a double dispatch [17]. The relationship between the elements is illustrated in Figure 5. The Advanced OCL Editor implements the following visitor classes (described in Section 3.3):

- `AbstractIncrementalProjectBuilder`
- `OCLLibraryIncrementalProjectBuilder`.

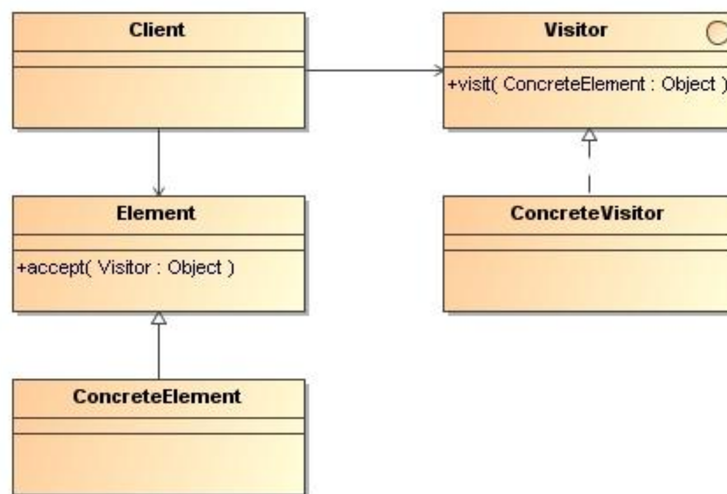


Figure 5 Visitor Pattern

## 3.2. Design Overview and Context

In the following sub-chapters, the architecture of the Advanced OCL Editor (Section 3.2.1) is presented which is based on the introduced architectures (Section 3.1). Furthermore, the library interpretation process is described (Section 3.2.2) and Figure 8 illustrates how the Advanced OCL Editor is integrated into the SQUAM Framework [18] (Section 3.2.3).

### 3.2.1 Component Diagram

A software component is a system element with an explicit interface which can be connected to other components without any modifications [2]. A component diagram shows how a software system is split up into different components and shows the dependencies among them [19].

The component diagram illustrated in Figure 6 presents the main components of the Advanced OCL Editor in the context of the three-tier architecture. The orange components (`gui`, `parser`, `library parser`) are implemented in the context of this thesis. The top-most level of the application is the graphical user interface (`gui`). In the application layer, the relevant parts for OCL library interpretation can be found: `parser`, `library parser`, and two external components, i.e. `OCL interpreter` and `compiler compiler`. The `parser` is responsible for OCL-expression-parsing. Therefore, the `parser` uses the `library parser` that separates the pure OCL from the OCL library syntax. The `library parser` itself uses an external `compiler compiler`. Finally, the pure OCL parts of the OCL library are interpreted by an `OCL interpreter`. In the last layer, the `file system` is considered. The library interpretation process is described in the next section.

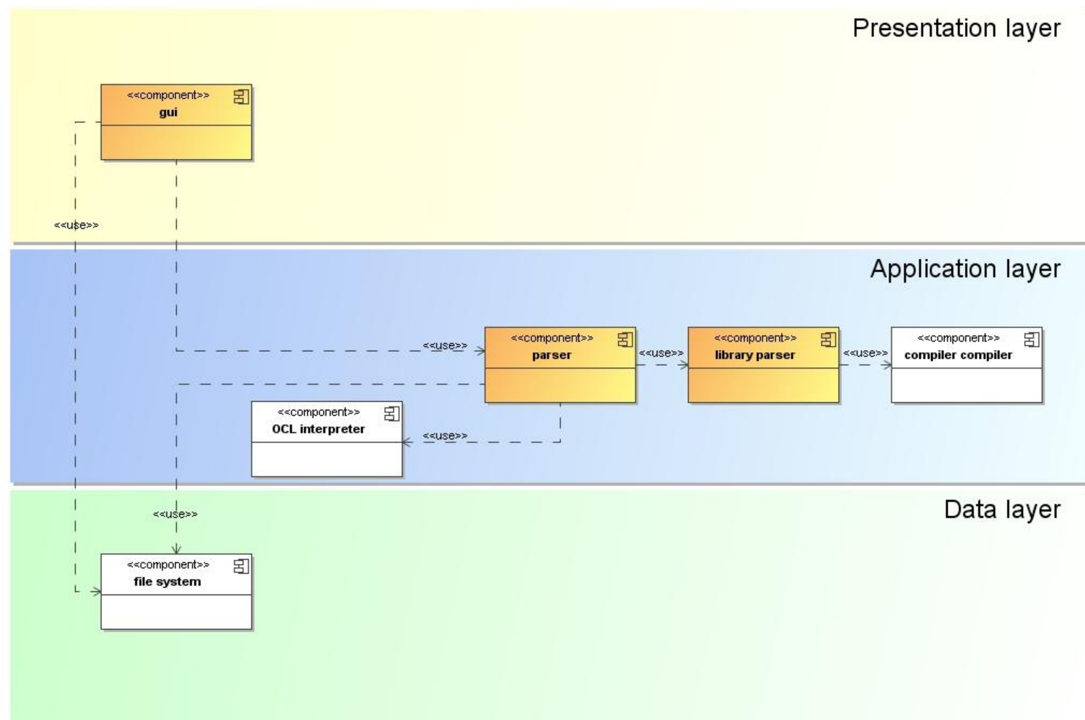


Figure 6 Advanced OCL Editor Architecture

### 3.2.2 Library Interpretation Process

OCL libraries include parts which are standard OCL expressions. Therefore, the process of the library interpretation is split into two parts (Figure 7). The first one is related to the interpretation of standard OCL and is in the scope of the `OCL interpreter` functionality. The second one is related to the OCL library concept and is included in the `OCL Editor` functionality. The extraction of a standard OCL expression takes place in several components, i.e. `parser`, `library parser` and `compiler compiler`. Hence, the second part is split into the responsibilities of these components.

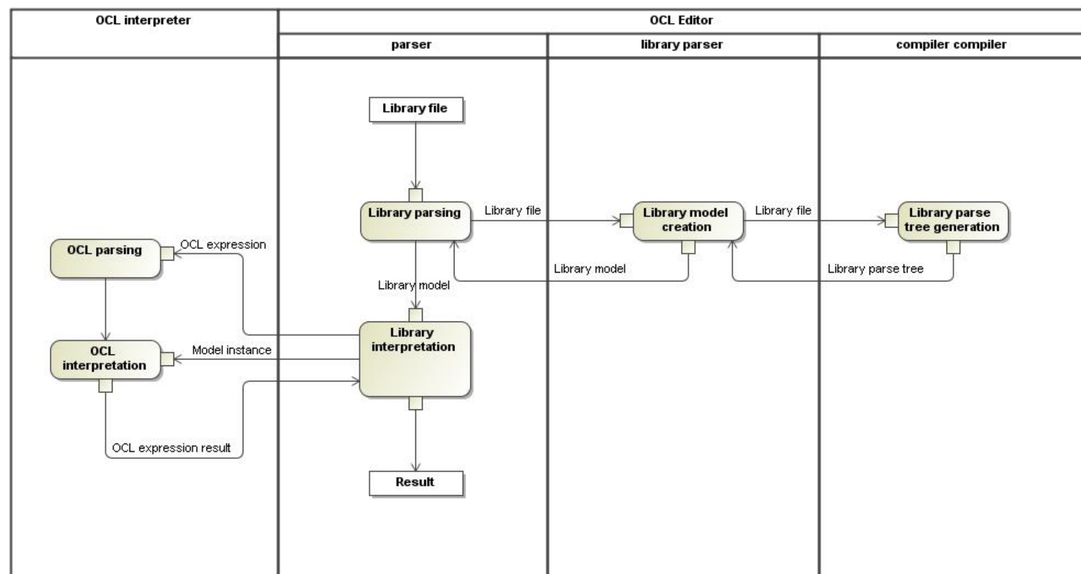


Figure 7 The Library Interpretation Process

The interpretation process starts with the library file used in the `OCL Editor`. This file includes a library to be interpreted. The input library is parsed in the `compiler compiler` by which an appropriate parse tree (tokens) is obtained. Afterwards the parse tree is transformed into a Library model object in the `library parser`. Parts of the Library model related to the standard OCL are sent to the `OCL interpreter` together with model instances. In the `OCL interpreter` OCL-expression -parsing and -interpretation take place and the results are sent back to the `OCL Editor`. In the `OCL Editor`, the results from the `OCL interpreter` are used to obtain the complete interpretation of the input library.

### 3.2.3 Advanced OCL Editor in the Context of SQUAM

As mentioned before the Advanced OCL Editor is a part of the SQUAM architecture [18]. Relevant parts of the architecture are demonstrated in Figure 8. At the top level, `OCLEditor`, `SQUAMFramework`, and two external components, i.e. `EclipseOCL` and `UMLTool` are considered. `OCLEvaluator` and `GeneralLibrary` are parts of the `SQUAMFramework` related to the library concepts, while other parts of `SQUAM Framework` architecture are omitted.

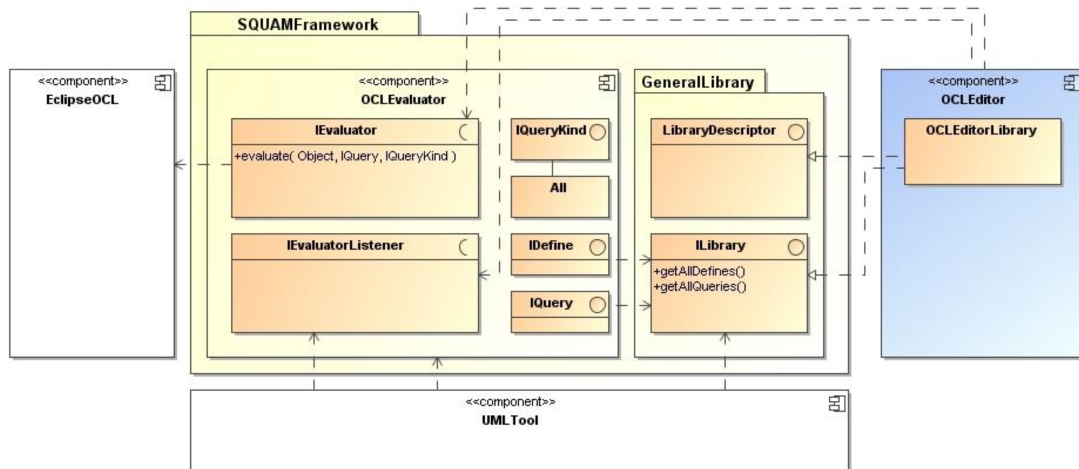


Figure 8 Architecture in the Context of SQUAM

`OCLEvaluator` is based on `EclipseOCL`, which is not used directly but via an interface. The `IEvaluator` interface is designed to enable the exchange of the underlying OCL interpreter and to include additional functionality. Another interface is `IEvaluatorListener` to which observers of a particular kind (`IQueryKind`) of OCL expression evaluations can register. In Figure 8, two observers, `UMLTool` and `OCLEditor`, are registered to `IEvaluatorListener`. Additionally, two interfaces, `IDefine` and `IQuery`, are designed to hide the internal representation of defines and queries. `GeneralLibrary` provides an extension point which enables the definition of different types of libraries. A library is specified by implementing the `ILibrary` interface which provides methods to get all defined queries and defines. A library includes a `LibraryDescriptor` with information on the library. `OCLEditorLibrary` is an implementation of `GeneralLibrary` used in the `OCLEditor` and it implements both interfaces. Observers of OCL expressions, like `UMLTool`, use the `ILibrary` interface, which conceals the concrete implementation.

## 3.3. Design Details

In this section, the implemented classes are discussed. The package diagram (Figure 9) gives an overview of the entire system (Section 3.3.1). Then, the classes will be described according to the layers of the three-tier architecture, i.e. presentation layer (Section 3.3.2) and application layer (Section 3.3.3). The data layer is neglected since the Eclipse File System is used for data persistence.

### 3.3.1 Class Diagram

Figure 9 gives an overview of the entire system. The packages `template`, `ocldoc`, and `ocldocexport` have been implemented by Hannes Mösl and are not part of this thesis. Therefore, these parts of the application are not described in detail. Due to the complexity of the system, not a complete class diagram is displayed, but specific parts thereof will be delineated in the following sections.

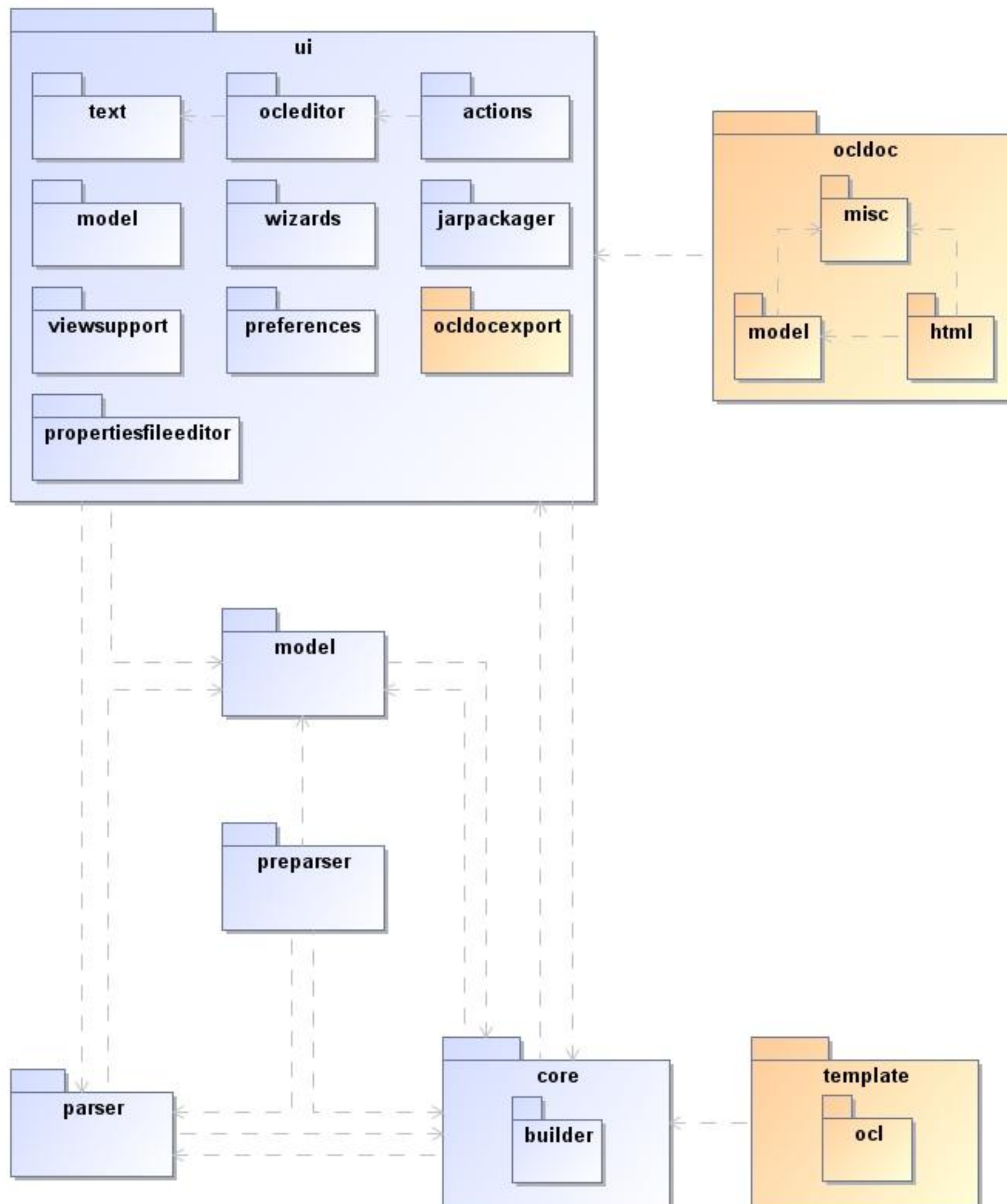


Figure 9 Package Diagram

### 3.3.2 Presentation Layer

The presentation layer is responsible for the graphical representation of an application. Therefore, the `ui` package consists of various wizards, dialogs, views, and of course the editor. The presentation layer includes the following classes:

| Class | Description | I |
|---|---|---|
| Package: info.squam.ocl.editor.ui | | |
| OclPerspectiveFactory | Generates the initial page layout. | E |
| ResourceManager | Manages images for the Advanced OCL Editor. | E |
| Package: info.squam.ocl.editor.ui.text | | |
| IOCLPartitionTypes | Definition of the OCL partitioning and its partitions. | H |
| OCLAnnotationHover | Determines all markers for the given line and collects and concatenates their messages. | E |
| OclColorProvider | Manages SWT color objects and contains the color keys used for the syntax highlighting OCL code and OCLDoc compliant comments. | E |
| OCLConfiguration | Configuration of a source viewer which displays the OCL code. | P |
| OCLDocument | Document for OCL expressions. | P |
| OCLDocumentProvider | Creates OCL documents. | E |
| OclDocumentSetupParticipant | The document setup participant for the OCL. | E |
| OclMultiLineRule | A rule for detecting OCL patterns which begin with a given sequence and may end with a given sequence thereby spanning multiple lines. | H |
| OCLPartitionScanner | Divides the OCL code into single-line comments, multi-line comments, OCLDoc, and OCL library content. | P |

| Class | Description | I |
|---|---|---|
| Package: info.squam.ocl.editor.ui.text | | |
| OCLWhitespaceDetector | An OCL aware white space detector. | E |
| OclWordDetector | An OCL aware word detector. | E |
| Package: info.squam.ocl.editor.ui.text.ocl | | |
| CompletionProposalComparator | Compares two CompletionProposals according to their display strings. | H |
| IOCLSyntax | Definition of the OCL syntax. | P |
| OCLAutoCompletion | An OCL code completion support for OCL libraries, OCL definitions, OCL queries, and OCL tests. | H |
| OclCodeScanner | An OCL code scanner which recognizes keywords, types, strings, iterators, functions, and comments. | E |
| OCLCompletionProcessor | OCL code completion processor. | P |
| OclTemplateCompletionProcessor | OCL template completion processor. | H |
| OCLTemplateInnerPartitionScanner | Scanner used for OCL templates which divides the OCL code into the OCL library header, OCL definitions, OCL queries, and OCL tests. | H |
| OCLTemplateOuterPartitionScanner | Scanner used for OCL templates which divides the OCL code into single-line comments, multi-line comments, OCLDoc, and OCL library content. | H |
| Package: info.squam.ocl.editor.ui.ocleditor | | |
| OCLContentOutlinePage | Represents the library structure as a tree. | E |
| OCLDocumentModel | Model for `OCLEditor`, enables automatic parsing after a specific delay. | P |
| OCLEditor | OCL library specific text editor. | P |
| OCLOutlineInput | Model for `OCLContentOutlinePage`. | H |

| Class | Description | I |
| --- | --- | --- |
| Package: info.squam.ocl.editor.ui.actions | | |
| EvaluateAction | Action to evaluate OCL expressions. | E |
| RunTestAction | Action to run OCL tests. | E |
| Package: info.squam.ocl.editor.ui.model | | |
| ModelImportSelectionContentProvider | Content provider for user models. | E |
| ModelingLevelProvider | Content provider for modeling levels. | E |
| TargetMetamodel | Represents meta-models. | E |
| TargetModelProvider | Content provider for meta-models. | E |
| UserJarsContentProvider | Content provider for imported Java Archive (JAR) files including OCL libraries. | E |
| UserModelsContentProvider | Content provider for imported user models. | E |
| Package: info.squam.ocl.editor.ui.wizards | | |
| NewOclLibraryProjectWizard | Wizard to create a new OCL library project. | E |
| NewOclLibraryProjectWizardPage | Wizard page to create a new OCL library project. | E |
| NewOclLibraryWizard | Wizard to create a new OCL library. | E |
| NewOclLibraryWizardMainpage | Wizard page to create a new OCL library. | E |
| OCLLibraryImportWizard | Wizard to import OCL library projects. | H |
| Package: info.squam.ocl.editor.ui.jarpackager | | |
| ExportJarWizard | Wizard for exporting OCL libraries from the workspace to a JAR file. | E |
| OCLManifestProvider | Creates manifest files for exported JAR files including OCL libraries. | E |
| WizardJarExportPage | Wizard page for the Export JAR Wizard. | E |

| Class | Description | I |
|---|---|---|
| Package: info.squam.ocl.editor.ui.viewsupport | | |
| DecorationImages | Manages images for label decoration. | E |
| LibraryLabelDecorator | Decorates OCL libraries in the Package Explorer with problem markers. | E |
| LibraryLabelProvider | OCL library label provider for the Package Explorer. | E |
| OclLibraryDecorator | OCL library project label provider for the Package Explorer. | E |
| Package: info.squam.ocl.editor.ui.viewsupport | | |
| OclProjectDecorator | Decorates OCL library projects in the Package Explorer with problem markers. | E |
| OverlayImageIcon | This class is used for overlaying image icons. | E |
| UserLibrariesLabelDecorator | Decorates libraries in the properties dialog with problem markers. | E |
| UserLibrariesLabelProvider | Library label provider for the properties dialog. | E |
| UserModelsLabelDecorator | Decorates user models in the properties dialog with problem markers. | E |
| UserModelsLabelProvider | User model label provider for the properties dialog. | E |
| Package: info.squam.ocl.editor.ui.preferences | | |
| OCLMetamodelPreferencePage | Specifies the meta-model. | H |
| OCLPreferencePage | Defines the target meta-model and the modeling level. | E |
| OclTemplatePreferencesPage | Configures template preferences. | H |
| PreferenceInitializer | Initializes the preferences with default values. | P |

| Class | Description | I |
|---|---|---|
| Package: info.squam.ocl.editor.ui.propertiesfileeditor | | |
| ImportSelectionDialog | Selects external user models or OCL libraries. | E |
| OCLPropertyPage | Defines the target meta-model and the modeling level. | E |
| UserLibrariesPropertyPage | Imports external JAR files including OCL libraries. | E |
| UserModelsPropertyPage | Imports external user models. | E |

Table 2 Classes Representation Layer

**Table legend**

- "I" = Implemented by
- "E" = Ekrem Arslan
- "H" = Hannes Mösl
- "P" = Partially by Ekrem Arslan and Hannes Mösl

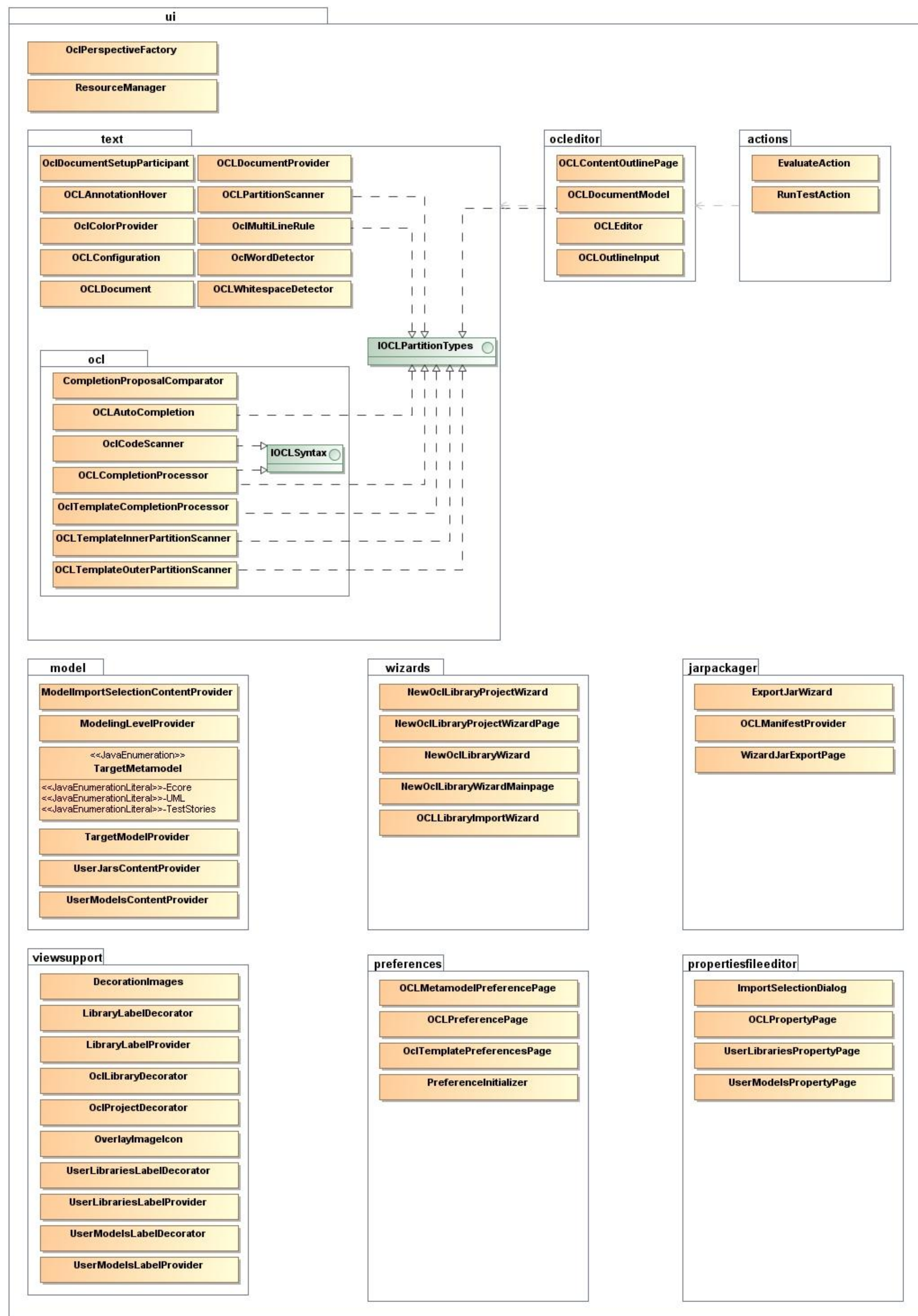


Figure 10 Class Diagram Representation Layer

### 3.3.3 Application Layer

The application layer models the functional core of the application and includes the following classes:

| Class | Description | I |
|---|---|---|
| Package: info.squam.ocl.editor.model | | |
| DefineImpl | Represents an OCL definition. | E |
| IOCLError | Common protocol for OCL elements which can have errors. | E |
| IOCLVisibility | Represents the visibility kind of OCL library contents. | E |
| LibraryModel | Represents an OCL library. | P |
| ModelingLevel | Enumeration of modeling levels on which OCL expressions can be defined. | P |
| OclToken | Represents a token parsed by the OclParser. | E |
| QueryImpl | Represents an OCL query. | E |
| QueryMessage | Represents an OCL query message. | E |
| Severity | Represents the severity kind of OCL tests. | E |
| StringToken | Represents a token parsed by the `OclPreParser` that contains a string. | E |
| Test | Represents an OCL test. | E |
| Visibility | Enumeration of visibility kinds on which OCL library contents can be defined. | E |
| Package: info.squam.ocl.editor.parser | | |
| ImportedOCLFile | Represents an imported OCL library. | E |
| LibraryModelParser | Analyzes the structure and the pure OCL parts of an OCL library. | P |
| OCLDefineParseException | Exception thrown by the `LibraryModelParser` when encountering parsing errors in definitions. | E |
| OCLError | Represents an exception thrown by the `LibraryModelParser`. | E |

| Class | Description | I |
|---|---|---|
| Package: info.squam.ocl.editor.parser | | |
| OCLInternalException | Exception thrown by the `LibraryModelParser` when encountering errors other than parsing errors. | H |
| OCLParseError | Exception thrown by the `LibraryModelParser` when encountering parsing errors in the OCL library content other than definitions. | E |
| Package: info.squam.ocl.editor.preparser | | |
| ExtendedOclParser | An interface to `OclParser` that forwards exceptions. | E |
| OclLexer | Generated lexer that converts a sequence of characters into a sequence of tokens. | G |
| OclParser | Generated parser that analyzes a sequence of tokens to determine their grammatical structure with respect to the specified grammar with the parser building a parse tree. This parser is responsible for OCL libraries. | G |
| OCLPreParser | Creates a `LibraryModel` from a parse tree. | E |
| OCLRecognitionError | Exception thrown by the `OCLPreParser` when encountering recognition errors. | E |
| Package: info.squam.ocl.editor.core | | |
| AbstractProjectNature | Handles project natures. | E |
| Activator | Central access point for the Advanced OCL Editor. | P |
| DuplicateLibraryException | Exception thrown by the `OCLLibraryManager` when detecting a library twice. | E |
| DuplicateModelException | Exception thrown by the `OCLModelManager` when detecting a user model twice. | E |

| Class | Description | I |
|---|---|---|
| Package: info.squam.ocl.editor.core | | |
| Jar | Represents an imported JAR file including OCL libraries. | E |
| Library | Represents an OCL library. | E |
| MetaModelManager | Manages meta-models. | H |
| MissingLibraryException | Exception thrown by the `Activator` when a required library is missing. | E |
| OclEditorLibrary | Implements the `ILibrary` interface of the SQUAM framework. | E |
| OclEditorPluginConstants | A set of constants. | P |
| OCLLibraryManager | Manages OCL libraries. | P |
| OclLibraryNature | Configures a project as an OCL library project. | E |
| OclLibraryProjectCreator | Creates OCL library projects. | E |
| OCLModelManager | Manages user models. | P |
| UserModel | Represents a user model. | E |
| Package: info.squam.ocl.editor.core.builder | | |
| AbstractIncrementalProjectBuilder | Handles builders. | E |
| ModelError | Represents an error which occurs when reading a user model. | E |
| OCLFileReader | Reads OCL libraries from a file. | P |
| OclLibraryIncrementalProjectBuilder | Processes resources in an OCL library project and produces the built output. | P |
| OCLModelReader | Reads user models from a file. | P |

Table 3 Classes Application Layer

**Table legend**

- “I” = Implemented by
- “E” = Ekrem Arslan
- “H” = Hannes Mösl
- “P” = Partially by Ekrem Arslan and Hannes Mösl
- “G” = Generated

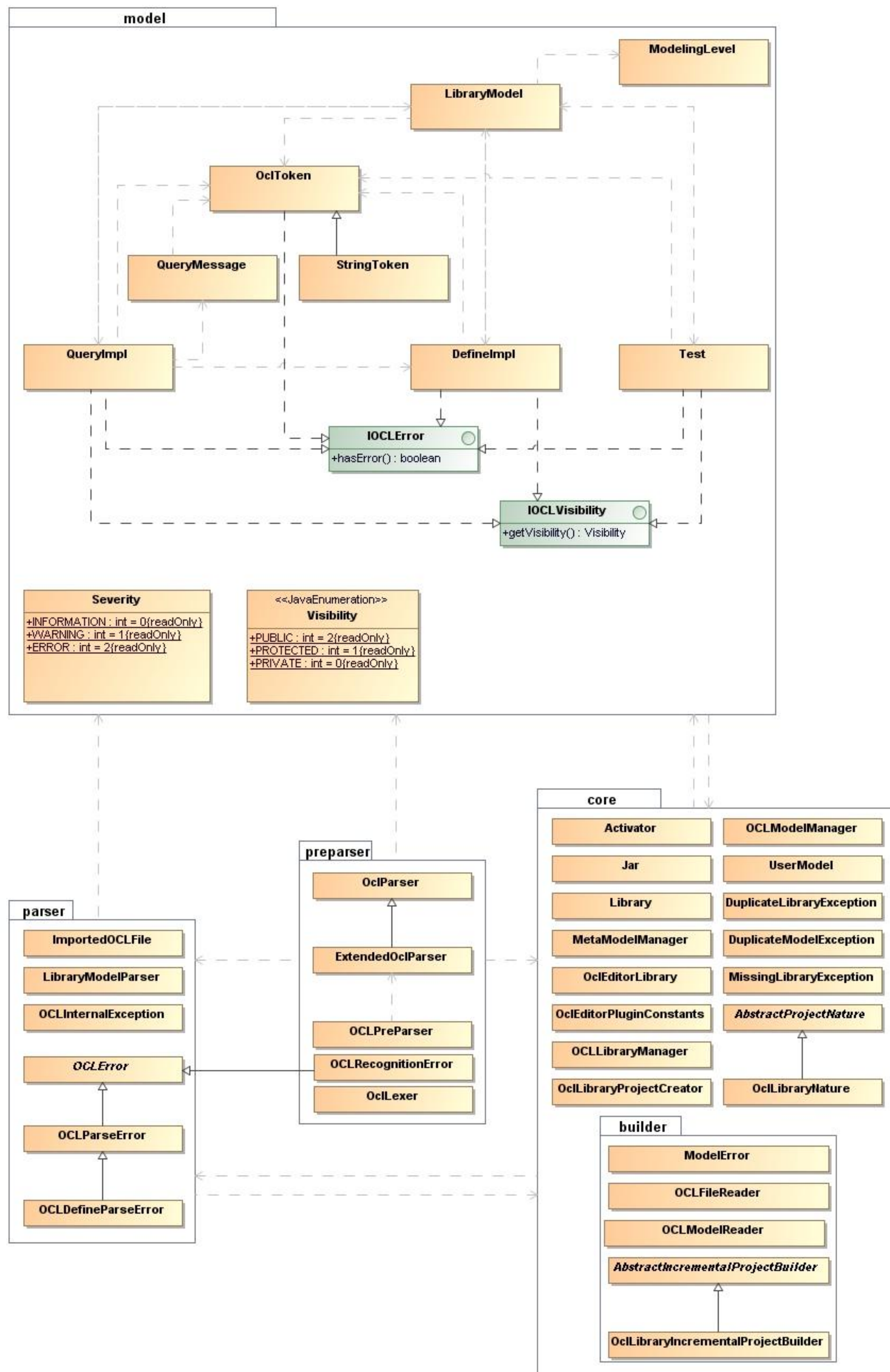


Figure 11 Class Diagram Application Layer

Chapter 4

# Implementation

This chapter discusses the tools and methods used to implement the Advanced OCL Editor. Figure 12 gives an overview of the technologies involved. Afterwards these technologies are discussed in more detail. The first part briefly introduces the programming language Java and the second part presents Eclipse first as an integrated development environment (IDE) and then as a Rich Client Platform (RCP). Finally, the Eclipse OCL interpreter is presented. The third section of this chapter introduces the ANTLR parser generator.

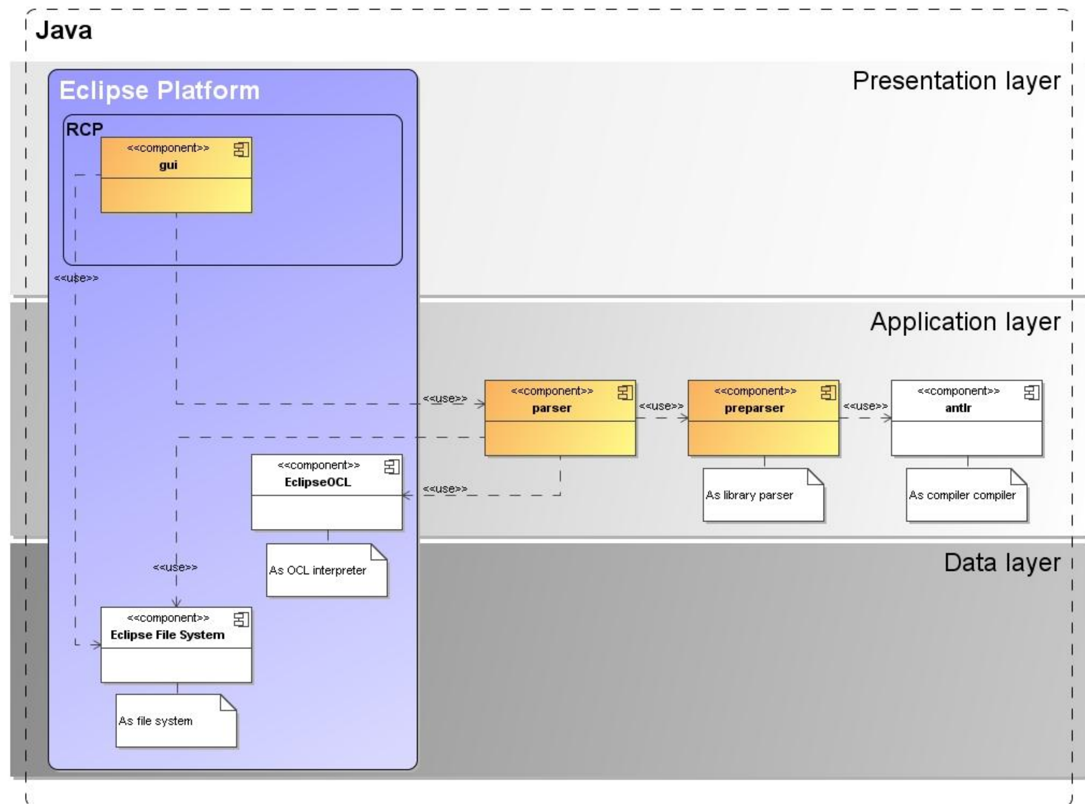


Figure 12 Concrete Architecture and the Technologies Involved

## 4.1. Java

The programming language Java [8] plays an essential role for the Advanced OCL Editor, since the platform Eclipse [5] and the Eclipse plug-in architecture are developed in the programming language Java. Moreover, the Advanced OCL Editor is implemented in Java (JDK 6.0). Java has originally been developed by Sun Microsystems and was released in 1995 as a core component of the Sun Microsystems Java platform. The language derives much of its syntax from the programming languages C and C++, but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to a byte code which can run on any Java Virtual Machine (JVM) regardless of the computer architecture [20].

## 4.2. Eclipse

Eclipse is a Java-based extensible open source platform. By itself, it is a framework and a set of services for building applications with plug-in components. By default, Eclipse comes with a standard set of plug-ins, including the Java Development Tools (JDT). While Eclipse is best-known for its use as Java IDE (Section 4.2.1), it is also increasingly used for developing rich client applications based on the Eclipse Rich Client Platform (RCP) (Section 4.2.2) [21].

### 4.2.1 Eclipse IDE

Syntax highlighting, syntax checking, code completion, and code generation are only a few essential features of Eclipse Java IDE which support developers in their daily work. Furthermore, Eclipse provides different search and navigation possibilities such as “call hierarchy” or “search references”. The integrated Java debugger helps developers to eliminate bugs. The Eclipse JDT offers an IDE with a built-in incremental Java compiler and a full model of the Java source files which allows for an advanced refactoring and code analysis [22]. Moreover, Eclipse Java IDE offers support for the Concurrent Versions System (CVS) which is essential for a distributed development. Based on the plug-in architecture of Eclipse, several modeling and software development tool providers offer integration possibilities for Eclipse. Hence, developers can build an IDE including their preferred set of development tools.

### 4.2.2 Eclipse Rich Client Platform

While Eclipse is a powerful development environment, it can also be used to develop stand-alone applications which re-use the Eclipse framework and its functionality. The most important architectural characteristic of Eclipse is its plug-in architecture (Figure 13). The Eclipse IDE itself consists of a number of plug-ins which depend on each other. A plug-in, like the Advanced OCL Editor plug-in, is the smallest deployable and installable software component of Eclipse. For Eclipse, the entire RCP application is a plug-in.

Each plug-in can define extension points which in turn define possibilities for functionality contributions by other plug-ins. The GeneralLibrary element of the SQUAM framework (Figure 8) defines such an extension point, `info.squam.library.general.library`, where other plug-ins can provide

their implementation of an OCL library. Hence, plug-ins can use extensions, e.g. provide a functionality to these extension points.

The basis of this architecture is the runtime environment of Eclipse which is based on the OSGI Alliance [23]. Eclipse uses the OSGI reference implementation Equinox [24] to run upon. The plug-in concept of Eclipse is the same as the bundle concept of OSGI. Eclipse RCP provides and uses the same framework as the Eclipse Workbench [25].

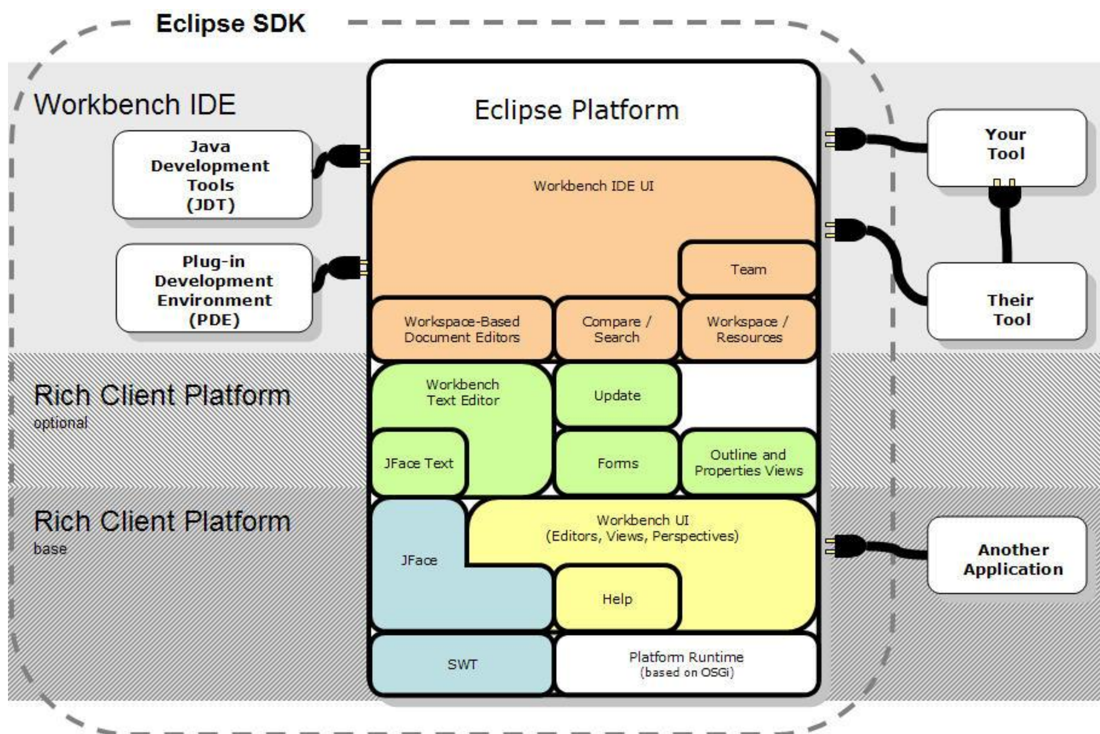


Figure 13 Overview of the Eclipse Architecture [26]

The Rich Client Platform mainly consists of components for the user interface. The Standard Widget Toolkit (SWT) and the JFace provide the basic functionalities for the GUI. The Workbench is built upon these GUI libraries. The components of the Workbench are available for every rich client application and form the core functionality of the Rich Client Platform. Additionally, the Rich Client Platform provides components for the Eclipse Help System, Update Manager, and components for text processing [27]. Developers can re-use many of these components and build sophisticated user interfaces very fast and easily.

### 4.2.3 Eclipse OCL

The Eclipse OCL Parser/Interpreter provides an implementation of the Object Constraint Language 2.0 specification for EMF-based [28] meta-models and models. The Eclipse OCL supports Ecore models as well as UML models. In addition to OCL constraint and query parsing, evaluation, and model validation, Eclipse OCL also provides an infrastructure for content assist in textual editors.

The OCL Parser provides two APIs for parsing constraint and query expressions. The `OCLHelper` interface, which is predominantly used in the Advanced OCL Editor for pure OCL parsing, has primarily been designed for parsing constraints and query expressions embedded in models, such as Ecore or UML models. Moreover, the `OCLHelper` API provides support for content assist by parsing partial OCL expressions and supplying completion suggestions. The `OCL` class, which is the main entry point to the parsing API, implements the parsing of OCL documents, for example from text files. In both cases, the concept of `Environment` is crucial.

The `OCL` class defines instances of autonomous OCL parsing and evaluation environments. It has a single root `Environment` created by an `EnvironmentFactory` implementation for a particular EMF-based meta-model. The OCL environment conceptually consists of the model which is to be constrained along with all of the constraints and additional operations and attributes defined. The Advanced OCL Editor creates OCL instances using the shared Ecore environment factory instance [29]:

```
//create an OCL instance for Ecore
OCL<?, EClassifier, ?, ?, ?, ?, ?, ?, ?, Constraint, EClass,
EObject> ocl;
ocl = OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);
```

The wildcards in the code snippet represent type parameters that are especially useful in the OCL API. Such type and method signatures make it difficult to understand the code. Therefore, a simplified interface (e.g. façade pattern) to the API would hide its internal complexity and could improve its reusability.

## 4.3. ANTLR Parser Generator and ANTLRWorks

ANTLR, ANother Tool for Language Recognition, is a language tool which provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions in a variety of target languages (e.g. Java). In common terminology, ANTLR is a compiler generator or a compiler compiler.

ANTLR reads a language description file called grammar and generates a number of source code files and other auxiliary files. Most uses of ANTLR generates at least one (and quite often both) of these tools:

- **Lexer** reads an input character or byte stream, divides it into tokens using specified patterns, and generates a token stream as an output. It can also flag some tokens such as whitespace and comments as hidden. The lexer for the Advanced OCL Editor is named `OclLexer`.
- **Parser** reads a token stream (normally generated by a lexer), matches phrases in the target language according to the specified rules (patterns), and typically performs a semantic action for each phrase matched (e.g. it generates an Abstract Syntax Tree). The generated parser for the Advanced OCL Editor is the `OclParser` component of the pre-parser [30].

ANTLRWorks is a grammar development environment for ANTLR v3 grammars. It combines an editor with an interpreter for rapid prototyping and a debugger for isolating grammar errors [31]. The grammar file for the Advanced OCL Editor, which has been developed with ANTLRWorks, can be found in Appendix A.

## 4.4. User Manual

Download, installation instructions, and user manual can be found at http://squam.info/ocleditor.

Chapter 5

# Project Plan

The first meeting regarding the Advanced OCL Editor took place on August 16, 2007. At that time I had heard about OCL in a lecture, but to me it was only a concept to define constraints on UML models. Since I did not have any real knowledge on OCL, I had to find out what OCL is, in which contexts it can be used, and what its syntax looks like. Furthermore, there were technical requirements to meet which necessitated an additional familiarization, i.e. Eclipse EMF, Eclipse OCL, and the SQUAM framework. An aggravating factor was that my technical contact left the team in the middle of September 2007. Moreover, there was a new concept of OCL libraries which had to be specified. Hence, the familiarization and definition of the requirements and concepts took quite a long time as can be seen in the project plan (Figure 14). The project plan illustrates a target/actual comparison. The bright bars in the Gantt chart represent the planned values and the dark bars the as-is state. The grey months (February 2008, April 2008, and August 2008) are periods of absence in which no development took place.

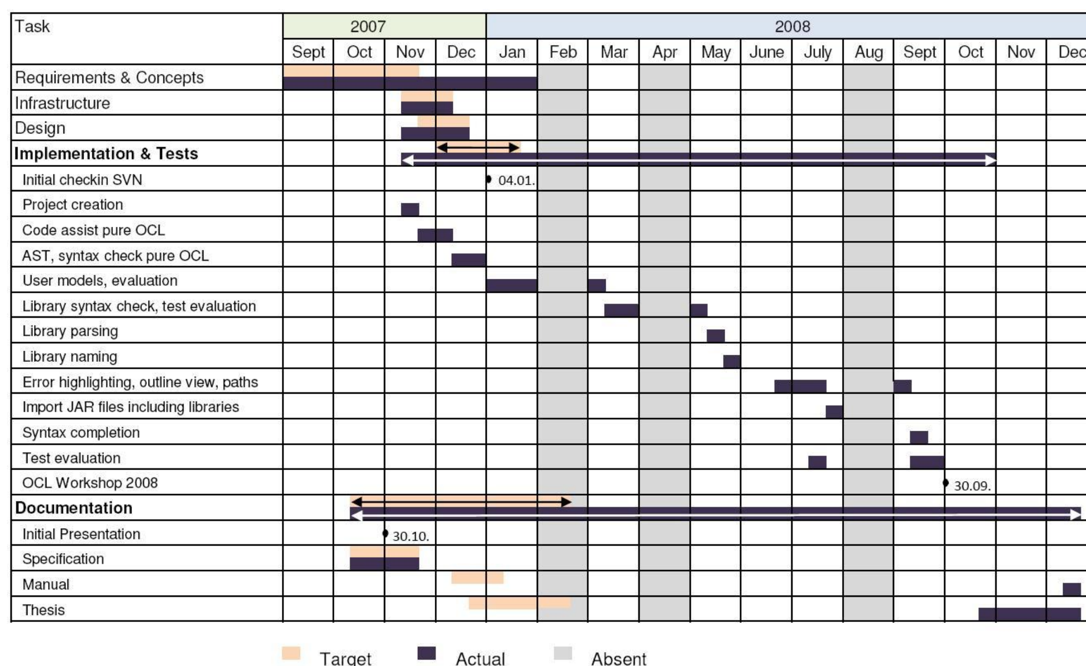


Figure 14 Target/Actual Comparison

This quite optimistic project plan was developed in the middle of October 2007. Considering that I am a part time student and that therefore the time I could spend on this thesis was limited, it was quite difficult to stick to this plan. Especially the time for the implementation time was too short which we recognized after a couple of weeks. Afterwards, we changed to an iterative development. The iteration steps are listed in Figure 14 under “Implementation & Tests”. During the entire project we held regular meetings which were very helpful and essential as we talked about our progress and the next development steps and since they forced me to make progress. The OCL Workshop 2008 on September 30, 2008 was an additional motivation to put more effort into the Advanced OCL Editor with which we had a new final deadline as the editor was to be presented at this workshop. Despite all efforts I would not have been able to implement all features until the OCL Workshop if it had not been for Hannes Mösl, a new developer who had joined the team in July 2008. As mentioned in the previous chapters Hannes Mösl has implemented several elements of the Advanced OCL Editor, e.g. the OCL documentation, which are not discussed in this thesis.

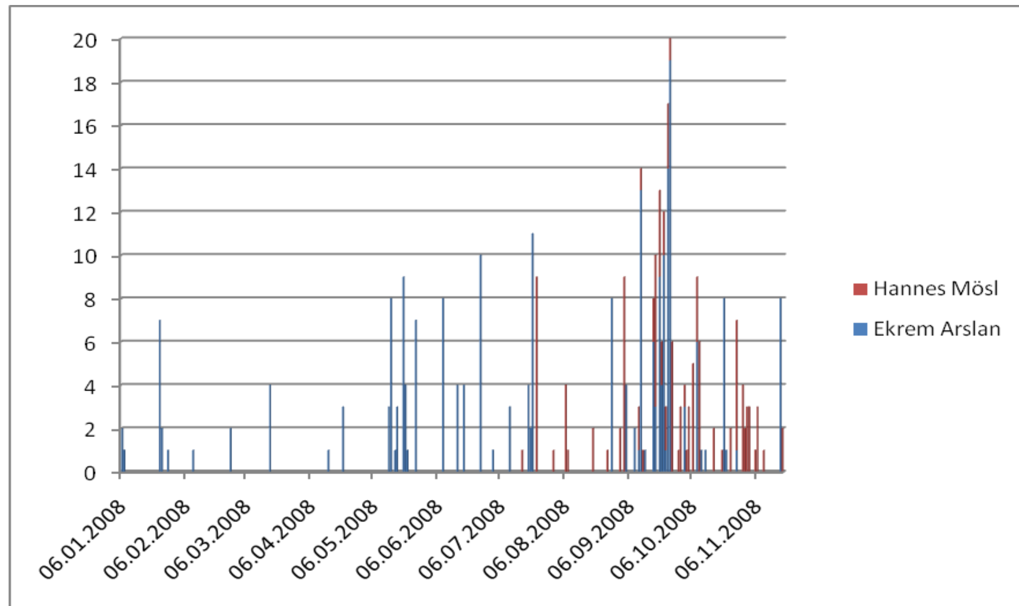


Figure 15 Changes SVN

Figure 15 illustrates the changes in the SVN. What is remarkable is the number of changes in September 2008 which could have two reasons. First of all, the deadline was coming nearer (the OCL Workshop) and, secondly, two developers worked on the same project. Before July 2008 I used the SVN more or less to backup the source code. After that time it was an essential tool to merge and exchange the source code. Team development without a version control system is unimaginable.

Finally, I want comment on the experiences I have made while working on this project. What I have learned during this project is the importance of time management. In the planning stage of the project I developed a very optimistic time estimate for the completion of the project, but I know now that a realistic planning is more valuable. For me the most challenging part of the project was to get familiar with the different concepts and technologies of OCL. The implementation was of course time consuming but not so challenging any more. During this project I gained insight into academic research which is invaluable for me. Moreover, I had the possibility to work in a team with very precious persons and I am eternally grateful to all of them.

# List of Figures

Conceptual Class Diagram ........ 6
Use Case Diagram ........ 9
Three-Tier Architecture ........ 20
Model-View-Controller ........ 21
Visitor Pattern ........ 24
Advanced OCL Editor Architecture ........ 26
The Library Interpretation Process ........ 27
Architecture in the Context of SQUAM ........ 28
Package Diagram ........ 29
Class Diagram Representation Layer ........ 35
Class Diagram Application Layer ........ 39
Concrete Architecture and the Technologies Involved ........ 41
Overview of the Eclipse Architecture [26] ........ 43
Target/Actual Comparison ........ 47
Changes SVN ........ 48

# List of Tables

Possible Combinations of Visibilities ........................................................................ 8
Classes Representation Layer ................................................................................... 34
Classes Application Layer ......................................................................................... 38

# List of Literature


[1] OMG: Unified Modeling Language: Superstructure, version 2.0. Final Adopted Specification, ptc/03-08-02, Object Management Group (2003)

[2] Breu, R.: Softwareentwicklung 4, SS 2004, University of Innsbruck, http://qe-informatik.uibk.ac.at

[3] OMG: Object Constraint Language. OMG Available Specification. Version 2.0 (2006)

[4] Chimiak-Opoka, J., Arslan, E., Peer, F. J.: OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions of the Object Constraint Language, submitted, attached in appendix B (October 2008)

[5] Eclipse – Open Development Platform, 2008. Documents and Download: http://www.eclipse.org

[6] RCP – Rich Client Platform, 2008. Documents and Download: http://www.eclipse.org/home/categories/rcp.php

[7] Subversion Open Source Version Control System, 2008. Documents and Download: http://subversion.tigris.org

[8] Java – Programming Language, 2008. Documents and Download: http://java.sun.com

[9] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, Addison-Wesley, Second Edition, 2003.

[10] Züllighoven, H.: Object-Oriented Cunstruction Handbook, dpunkt.verlag, 2004

[11] Balzert, H.: Lehrbuch der Software-Technik. Software Entwicklung, Spektrum Akademischer Verlag, Second Edition, 2000

[12] Fowler, M.: Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002

[13] Balzert, H.: Lehrbuch der Objektmodellierung: Analyse und Entwurf, Spektrum Akademischer Verlag, 1999

[14] De Nicola, R.: Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, Springer, 2007

[15] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reuseable Object-Oriented Software, Addison-Wesley, 2000

[16] http://www.oodesign.com/observer-pattern.html (retrieved on December 4, 2008, 09:24 CET)

[17] Article Visitor pattern. In: Wikipedia, The Free Encyclopedia. Revision: November 20, 2008, 15:41 CET. URL: http://en.wikipedia.org/wiki/Visitor_pattern (retrieved on November 27, 2008, 23:01 CET)

[18] Chimiak-Opoka, J., Lenz, C.: Use of OCL in a model assessment framework: An experience report. Electronic Communications of the EASST 5 (2006)

[19] Article Component diagram. In: Wikipedia, The Free Encyclopedia. Revision: September 6, 2008, 18:56 CET. URL: http://en.wikipedia.org/wiki/Component_diagram (retrieved on November 27, 2008, 22:57 CET)

[20] Article Java (programming language). In: Wikipedia, The Free Encyclopedia. Revision: November 26, 2008, 17:30 CET. URL: http://en.wikipedia.org/wiki/Java_(programming_language) (retrieved on November 28, 2008, 11:31 CET)

[21] Article Get started with the Eclipse Platform. In: Eclipse resources. Revision: July 17, 2008. URL: http://www.ibm.com/developerworks/opensource/library/os-eclipse-platform/?S_TACT=105AGX59&S_CMP=GR&ca=dgr-eclipse-1 (retrieved on November 30, 2008, 21:00 CET)

[22] Article Eclipse (software). In: Wikipedia, The Free Encyclopedia. Revision: November 13, 2008, 06:50 CET. URL: http://en.wikipedia.org/wiki/Eclipse_(software) (retrieved on November 30, 2008, 20:57 CET)

[23] OSGI: OSGI Alliance Specifications, Release 4 Version 4.1 http://www.osgi.org/Specifications/HomePage, May 2007

[24] Equinox, 2008. Documents and Download: http://www.eclipse.org/equinox/

[25] Vogel, L.: Article Eclipse Rich Client Platform (RCP) with Eclipse Ganymede (3.4) – Tutorial, Version 0.9, November 3, 2008. URL: http://www.vogella.de/articles/RichClientPlatform/article.html (retrieved on November 30, 2008, 22:34 CET)

[26] http://www.jdg2e.com/ch08.architecture/doc/index.html (retrieved on December 1, 2008, 14:32 CET)

[27] Daum, B.: Rich-Client-Entwicklung mit Eclipse 3.2: Anwendungen entwickeln mit der Rich Client Platform, dpunkt.verlag, 2., aktualisierte Auflage, 2007

[28] Eclipse Modeling Framework Project (EMF), 2008. Documents and Download: http://www.eclipse.org/modeling/emf/

[29] Eclipse documentation: OCL Developer Guide, 2008

[30] Article Five minute introduction to ANTLR 3. Revision: October 23, 2008. URL: http://www.antlr.org/wiki/display/ANTLR3/Five+minute+introduction+to+ANTLR+3 (retrieved on December 4, 2008, 19:36 CET)

[31] ANTLRWorks, 2008: Documents and Download: http://www.antlr.org/works/index.html

# Appendix A

The first instruction for ANTRL builds a flat abstract syntax tree (AST). The first section of the grammar file contains a list of tokens which together with the rewriting rules in the parser rules section structures the AST. Moreover, these tokens are used in the `OCLPreParser` to navigate the tree and to identify nodes. Rewriting rules are described by the arrow operator. An ^ operator indicates a new sub-tree. By rewriting rules, nodes can also be added to or removed from the syntax tree (e.g. `severityLevel` where the keyword “severity” is removed from the syntax tree). The last section of the grammar file contains lexer rules. A fragment in front of a lexer rule tells ANTLR that the rule is only used as a part of another lexer rule.

```
grammar Ocl;

options {
    output=AST;
}

tokens {
    LIB_NAME;
    LIB_HEADER;
    METAMODEL_DEF;
    IMPORT_DEF;
    REQUIRE_DEF;
    LIB_CONTENT;
    VISIBILITY;
    DEFINES_SET;
    DEFINES_CONTEXT;
    DEFINES_BLOCK;
    DEFINE_UNIT;
    DEFINE_NAME;
    QUERIES_SET;
    QUERIES_CONTEXT;
    QUERIES_BLOCK;
    QUERY_UNIT;
    QUERY_TRIGGER;
    QUERY_MESSAGE;
    QUERY_NAME;
    QUERY_CONTENT;
    SEVERITY_LEVEL;
    TESTS_BLOCK;
    TEST_SET;
    TEST_MODEL;
    TESTS_SET;
    TEST_UNIT;
    TEST_NAME;
    TEST_CONTENT;
    EXPECTED_RESULT_SET;
    EXPECTED_RESULT;
}

/*------------------------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------------------------*/

library :       'library' String libHeader? libContent+ 'endlibrary' EOF ->
                ^( LIB_NAME String) ^( LIB_HEADER libHeader?)
                ^( LIB_CONTENT libContent+);
```

```
libHeader
	:	metamodeldef? importdef* ->
		^( METAMODEL_DEF metamodeldef?) ^(IMPORT_DEF importdef*);
importdef
	:	'require' libName ->
		^( REQUIRE_DEF libName);
metamodeldef
	:	'metamodel' modelName ->
		modelName;
libContent
	:	visibilityKind (defines -> ^(DEFINES_BLOCK
		^(VISIBILITY visibilityKind) defines) |
		queries -> ^(QUERIES_BLOCK ^(VISIBILITY visibilityKind) queries) |
		tests -> ^(TESTS_BLOCK ^(VISIBILITY visibilityKind) tests));
defines :	'definitions' definesBlock+  'enddefinitions' ->
		 definesBlock+;

definesBlock
	:	extContextDeclaration definition+ ->
		^(DEFINES_CONTEXT extContextDeclaration) ^(DEFINES_SET definition+);
definition
	:	'def' String? ':' evaluation -> ^(DEFINE_UNIT 'def'
		^(DEFINE_NAME String?) ':' evaluation);
extContextDeclaration
	:	'context' String -> String | 'nonecontext';

queries :	'queries' queriesBlock+ 'endqueries'->
		queriesBlock+;
queriesBlock
	:	extContextDeclaration query+ ->
		^(QUERIES_CONTEXT extContextDeclaration) ^(QUERIES_SET query+);
query	:	queryHeader trigger? message? ->
		^(QUERY_UNIT queryHeader ^(QUERY_TRIGGER trigger?)
		^(QUERY_MESSAGE message));
trigger :	'trigger' evaluation ->
		evaluation;
queryHeader
	:	'query' String? ':' severityLevel? evaluation ->
		^(QUERY_NAME String?) ^(SEVERITY_LEVEL severityLevel?)
		^(QUERY_CONTENT evaluation);
severityLevel
	:	'severity' String ->
		String;
evaluation
	:	(String|Characters)+ .*;
message :	'message' composedString 'endmessage' ->
		composedString;
quotedString
	:	'"' (EscapeSequence | ~('\\'|'"'))* '"';

composedString
	:	(quotedString| .* ) ('+' composedString)*;
tests	:	'tests' testSettings testUnit+ 'endtests'->
		^(TEST_MODEL testSettings) ^(TESTS_SET testUnit+);

testSettings
	:	('model' modelName)? -> modelName;
testUnit:	'test' String? ':' evaluation expectedResult+ ->
		^(TEST_UNIT ^(TEST_NAME String?) ^(TEST_CONTENT evaluation)
		^(EXPECTED_RESULT_SET expectedResult+));
expectedResult
	:	'expected' evaluation ->
		^(EXPECTED_RESULT evaluation);
visibilityKind
	:	'private' | 'protected' | 'public';
libName :	qualifiedName;
modelName
	:	qualifiedName;
qualifiedName
	:	String ('.' String)*;
```

```
/*------------------------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------------------------*/

String
        :       (Letter|Digit)+;

/*
   0..9 |
   Different Alphabets
*/
Digit
    :  '\u0030'..'\u0039' |
       '\u0660'..'\u0669' |
       '\u06f0'..'\u06f9' |
       '\u0966'..'\u096f' |
       '\u09e6'..'\u09ef' |
       '\u0a66'..'\u0a6f' |
       '\u0ae6'..'\u0aef' |
       '\u0b66'..'\u0b6f' |
       '\u0be7'..'\u0bef' |
       '\u0c66'..'\u0c6f' |
       '\u0ce6'..'\u0cef' |
       '\u0d66'..'\u0d6f' |
       '\u0e50'..'\u0e59' |
       '\u0ed0'..'\u0ed9' |
       '\u1040'..'\u1049' ;

/* $ |
   A..Z |
   _ |
   a..z |
   À..Ö |
   Ø..ö |
   ø..ÿ |
   Different Alphabets
*/
fragment
Letter
    :  '\u0024' |
       '\u0041'..'\u005a' |
       '\u005f' |
       '\u0061'..'\u007a' |
       '\u00c0'..'\u00d6' |
       '\u00d8'..'\u00f6' |
       '\u00f8'..'\u00ff' |
       '\u0100'..'\u1fff' |
       '\u3040'..'\u318f' |
       '\u3300'..'\u337f' |
       '\u3400'..'\u3d2d' |
       '\u4e00'..'\u9fff' |
       '\uf900'..'\ufaff' ;

Characters
    : '='|'>'|'<'|'('|')'|'{'|'}'|','|'-'|
      ';'|'%'|'|'|'!'|'@'|'`'|'´'|'~'|'#'|
      '&'|'^'|'\\'|'['|']'|'?'|'/'|'*'|'\'';

EscapeSequence
    :  '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\');

WS  : (' '|'\r'|'\t'|'\u000C'|'\n')+ {$channel = HIDDEN;} ;

COMMENT
    :   '/*' .* '*/' {$channel=HIDDEN;} ;

LINE_COMMENT
    : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;} ;

OCL_LINE_COMMENT
        : '--' ~('\n'|'\r')* '\r'? '\n'      {$channel=HIDDEN;};
```

# Appendix B

Chimiak-Opoka, J., Arslan, E., Peer, F. J.: OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions of the Object Constraint Language (October 2008).

Submitted to FASE'09 Fundamental Approaches to Software Engineering on October 2, 2008.