Arbeitsbereich Softwaresysteme

Roundtrip Engineering for Classes: Mapping between UML Diagram and Java Structures based on Poseidon for UML™ and the Eclipse™ Platform

Submitted by:
Sunay YALDIZ

Submitted in partial fullfilment of the requirements for the degree
Master of Science in Information and Media Technologies

Supervised by:

Prof. Dr. Joachim Schmidt (STS)
Prof. Dr. Volker Turau (Telematik)
Msc. Miguel Garcia (STS)
Jesco von Voss (Gentleware AG)
Software Systems Department
TECHNICAL UNIVERSITY HAMBURG-HARBURG
GERMANY

Declaration

I declare that:

This work has been prepared by me,

All literal or content based quotations are clearly pointed out,

And no other sources or aids than the declared ones have been used.

Hamburg, *1. November* 2004

Sunay Yaldiz

Acknowledgements

I would like to thank Professor Joachim W. Schmidt of STS for supervising this thesis. My thanks also go to the Professor Volker Turau of Telematik department for being co-coordinator.

Research assistant Miguel Garcia of STS was very much helping during the thesis with his guidance on the topic as well as survey of literature.

Jesco von Voss of Gentleware AG helped me throughout this work. His suggestions and view on the problems were always great help to me.

Finally I want to thank Dr. Marko Boger for giving the chance to do this thesis work at comfortable and friendly environment of Gentleware AG.

For Mom and Dad,

CONTENTS

INDEX OF FIGURES

INDEX OF TABLES

LIST OF ABBREVIATIONS (MOSTLY MENTIONED ONES)

AOP	Aspect Oriented Programming
AOSD	Aspect Oriented Software Development
API	Application Programming Interface
AST	Abstract Syntax Tree
DI	Diagram Interchange
DTD	Document Type Definition
EJB	Enterprise Java Beans
EMF	Eclipse Modeling Framework
IDE	Integrated Development Environment
JBoogie	Java Based Object Oriented Interaction and Editing
JDT	Java Development Tools
MDA	Model Driven Architecture
MDR	Metadata Repository
MOF	Meta Object Facility
NOA	Number of Associations
NOC	Number of Classes
OMG	Object Management Group
SOC	Separation of Concerns
SWT	Standard Widget Toolkit
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

2 Introduction

2.1 Motivation and Scope

Most software projects today are designed using the UML notation. UML tools allow the developers/modelers to model the software system before starting the implementation. The time spent on analysis and modeling pays off given that it is possible to obtain source code from the derived models and continue to refine the system manually in the implementation phase. Most design tools offer code generation for popular programming languages. However, recent development methods (extreme programming, prototyping, feature-based programming) require frequent switches between design and implementation phases, and therefore code generation alone is not a suitable approach. A UML tool must also reflect the changes of the source code in the model.

In many projects, UML diagrams of the design phase become outdated as the code is developed further. To obtain the highest efficiency in any project lifecycle, tools integrating source code/classes and UML models are necessary. These tools must avoid information loss between UML models and source code. The question of what and when to synchronize is a main point. The synchronization must happen continuously, so that the time span where differences between model and code exist is as small as possible, and without information loss. Some tools offer both UML editor and source code editor, and synchronize model and code continuously. To allow a best-of-breed selection of tools, however, a loose coupling between an integrated development environment (IDE) and UML editor is desirable. A general interface between UML modeling tools and IDEs would be of high value to all software developers, who would then be able to choose a tool of their liking in both areas (design and coding).

In this master thesis, an analysis of transformation between UML models and Java source code will be done. To understand the problem well and present solutions suggested in previous studies, a review of literature will be done. The review will focus on mapping the UML constructs to Java concepts as loss-free as possible. The mappings are to be used both in code generation from UML models and in reverse engineering from Java source files. An overview of some tools supporting roundtrip engineering will be given.

The analysis and mapping methods for UML models to Java code will be then realized by an implementation of the chosen subset of the synchronization operations. The aimed synchronization takes place between UML models represented in Poseidon for UML tool and Java source code in the Eclipse Platform.

2.2 Structure of the Thesis

This report desribes the methodology and the tasks followed during the thesis work. The structure of the report is as follows:

In this chapter, an introduction to the thesis work is given. The motivation behind this work and the structure of the report are described.

Chapter 2, 3 and 6 delve into the details of Code Generation, Reverse Engineering and Roundtrip Engineering between UML models and Java source, concentrating on the subset relevant to this thesis. An extensive review of the literature as well as a review of approaches in select UML tools approaches will be given, especially in chapter 6, *Code Generation*, where the main problems during mapping the UML constructs to Java code are described with help of the reviewed papers. Chapter 2 and 3 contain the reviews for Reverse Engineering and

Roundtrip Engineering. Reverse Engineering can not be thought independently of the mappings analyzed in Chapter 6. Roundtrip Engineering is the process of combining Code Generation and Reverse Engineering to achieve continuous synchronization of model and source code.

Chapter 4 describes basic background concepts and tools. As the main concentration on this work is UML models, an introduction to UML will be given. Poseidon for UML is a UML modeling tool. In this chapter, Poseidon tool with its basic GUI and architecture is described.

Chapter 5 contains the Eclipse Platform descriptions. First an overview of its GUI and main components are given. Eclipse is a platform created to be extended by plugins, enabling their seamless integration. The plugin architecture of Eclipse is also described. The last part of the chapter is the description of the Java Development Tools, a Java IDE, and the main concepts of the Java Model and abstract syntax trees (AST) which are used in the prototype. This chapter also mentions the problem of Swing and Standard Widget Toolkit (SWT) libraries' incompatibility. Swing is the standard GUI library for the Java programming language. Unfortunately, Eclipse GUI, the Eclipse Workbench, is implemented with SWT library which made the integration of plugins with Swing GUI impossible. This problem is solved[1] and since Eclipse 3.0 release[2], plugins based on Swing can be integrated into the Eclipse platform.

Chapters 7 focus on the integration of Poseidon and Eclipse. It describes the architecture of the system design which is implemented, giving a summary of the architecture and components. Then, the chapter goes on by describing the implemented functionality conforming to the design. The main problems encountered during this thesis work are also described in this chapter.

In the last chapter the conclusions are presented. A summary of the thesis work topic and motivation are given. Finally, at the end of this chapter, the future work areas are summarized. A list of possible extensions to the work is briefly sketched.

[1] It is solved but integrating introduces many problems basically because of the event queue handling and threads in these two libraries.

[2] Eclipse 3.0 is released on June 25th.

3 Roundtrip Engineering

When developing a program, there is often a model represented in UML and source code. The purpose of the model is to visualize structural relationships, interesting interactions, or any feature about the system. The source code contains the implementation in a specific programming language and contains all the details required to execute the program. Neither design nor source code is the master store of information. They are equally important components with different focuses. Although they are different, they still share a lot of information. The information they share is ideally kept at any given time in synch. If they are not in synch, a process to synchronize them is desired. But it is both errorprone and boring to have to update the same information in two places by hand. **Roundtrip engineering** comes into play directly at this point. Roundtrip engineering aims to keep the model and the source code always in synch. Real time roundtripping changes the model and the source code accordingly whenever a change on one side occurs.

As UML tools have evolved, the level of integration between model and code has increased dramatically. In the early days of UML tools, modeling and coding were completely separate activities. Ideally a developer was first designing the model and then starting to implement the model in a programming language. Forward engineering generates code automatically from the model which simplifies the implementation process. On the other hand, for the existing source codes, reverse engineering was introduced, which is able to create the design model for the implementation. *A modeling tool that can seamlessly blend forward and reverse engineering is said to support roundtrip engineering*. This feature enables independent changes to the code and model to be synchronized.

As defined by Booch [1] [3], roundtrip is: *A style of design that emphasizes the incremental and iterative development of a system, through the refinement of different yet consistent logical and physical views of the system as a whole; The process of object-oriented design is guided by the concepts of roundtrip gestalt design; Roundtrip gestalt design is a recognition of that fact that the big picture of a design affects its details, and that the details often affect the big picture.*

Roundtrip engineering is different from forward and reverse engineering. Forward and reverse engineering are mainly one-way activities that take input and generate the required output. Roundtrip engineering extends these features and makes use of both of them to keep model and source code synchronized.

In software design a basic rule that is well known is *no design remains unchanged*. For small systems as well as for the larger ones the rule holds. Even the best software design contains issues that are not reflected. These issues are discovered as implementation proceeds. This suggests the design and implementation do not stay in synch. To keep the design and implementation code is very difficult, but also very important. The roundtrip engineering functionality enables the UML tools to synchronize the model with the changes in the source code.

The idea of Roundtrip Engineering is closely related to reverse engineering. Reverse engineering can be defined as the process of reconstructing the design of a product from the product itself. Assume that there is a reverse engineering procedure that is always able to give the design of a given product. Now assume that there is a procedure that will always generate the product from a given design. *If a design is reverse engineered from a product, used to generate a product and the generated product is identical to the original product then this is a roundtrip engineering system*. [2]

Bruegge and Dutoit [3] define round trip engineering as: *A model maintenance activity that combines forward and reverse engineering. Changes to the implementation model are*

[3] On page 517

propagated to the analysis and design models through ReverseEngineering. Changes to the analysis and design models are propagated to the implementation model through ForwardEngineering.

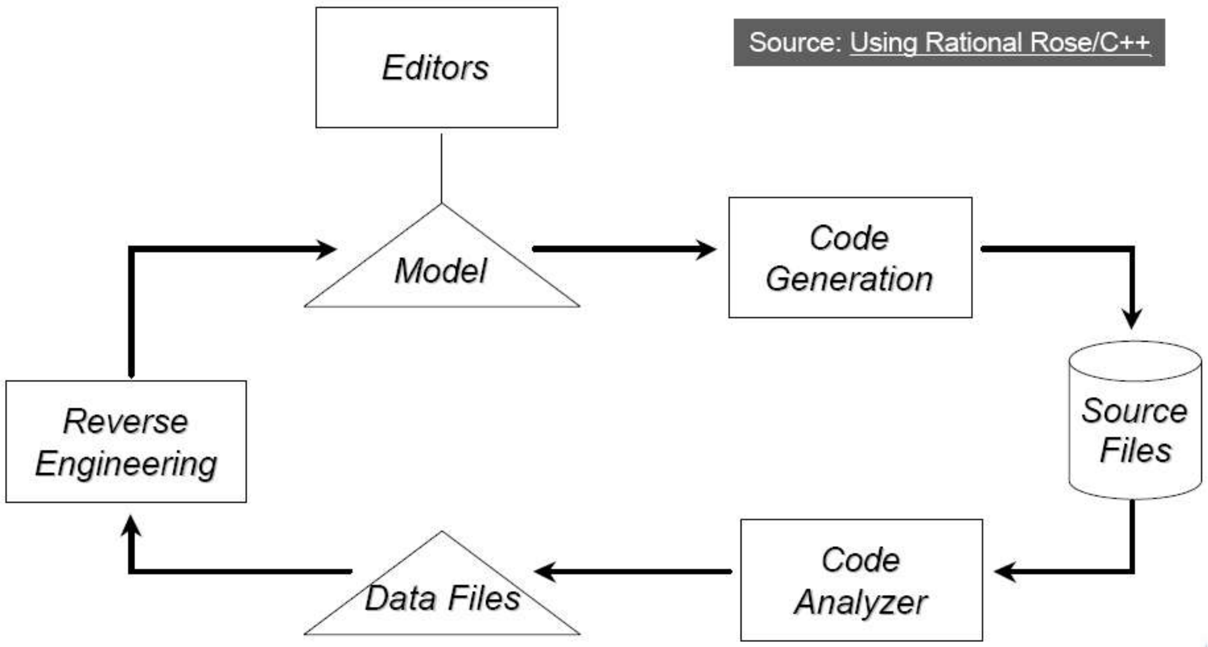


Figure 2-3.1 Roundtrip engineering process

Figure 2-3.1 shows the complete roundtripping process. As can be seen on the figure, roundtripping is a circle of modeling and implementation where changes on one of the representations trigger changes on the other one.

3.1 State of the Art

In this section, I will try to provide the state of the art for roundtrip engineering. Roundtripping approaches for some of the chosen commercial case tools as well as research prototypes are given.

As in reverse engineering and code generation, most of the commercial tools as well as research prototypes use only class diagrams for roundtripping. Roundtripping in many tools is directly dependent on reverse engineering and code generation. It can not be thought as a separate independent process. UML class diagrams and sequence diagrams are the mostly used diagram types, especially the class diagrams. This section starts with a short analysis of roundtripping with class diagrams, then an overview for other class diagrams' roundtripping is given. At the end of the section, some chosen tools and prototypes are explained based on their roundtripping approaches.

3.1.1 Class Diagrams

Current roundtrip engineering between UML and Java is based on static reverse engineering and the class diagrams of the UML. Many of the UML vendors offer roundtrip engineering support roundtrip engineering between UML class diagrams and Java source code. State of-the-art CASE tools like Rational Rose, TogetherJ, and Rhapsody, provide editors for various kinds of UML diagrams. However, since most UML behavior diagrams describe only scenarios, code generation and roundtrip engineering support is restricted to class diagrams and (in case of Rhapsody and Rational Rose RT) state-charts.[4]

Some of the research prototypes as Fujaba support class diagram roundtripping as well as story diagrams, state diagram and object diagram roundtripping. At roundtripping process, information from all these diagrams are used, at reverse engineering as well as at code generation.

TogetherJ, Rational Rose and Poseidon for UML's roundtripping is, as stated above, based on static structure of the source code and class diagrams. The behavior is difficult to capture.

3.1.2 Other UML Diagrams

State diagrams, activity diagrams and collaboration diagrams and sequence diagrams are used in roundtripping but in a very limited way. Currently only some research prototypes like Fujaba have limited roundtripping support for activity diagrams and collaboration diagrams as well as state diagrams. These diagrams represent the dynamic behavior of the programs.

As the context of this thesis work does not include roundtripping with UML diagrams other than class diagrams, no further information will be given on this topic.

3.1.3 FUJABA

Fujaba allows using UML class and behavior diagrams as a very high-level visual programming language called Story-Diagrams. The paper [4] gives descriptions on roundtrip engineering support for the visual programming language that exist in the Fujaba environment. The concepts for code generation are also described in [5], [6].

Fujaba can reverse engineer source code which follows some conventions. Currently it can not reverse engineer arbitrary code. It generates code based on specifications, and the code to reengineer should obey the specifications. Fujaba introduces a diagram type Story-Diagrams. Story diagrams' goal is to roundtrip static structure of the programs as well as the dynamic behavior.

As the generation of the Java code out of specifications stated in [5] and [6], the reversing is also divided in two parts. In the first part, the static information, namely the class diagrams will be constructed, and in the second part, the story diagrams are constructed.

Class diagrams can be recognized from Java code if the code is generated from Fujaba itself, or a developer uses the naming conventions and implementation concepts of Fujaba.

Fujaba uses Story-Diagrams for the specification of dynamic aspects. Story-Diagrams are a combination of UML activity diagrams and UML collaboration diagrams. Activity diagrams are used to specify the control flow and each activity can contain pure Java source code as well as a graph rewrite rule. The control flow can be reconstructed directly out of the syntax graph. Each activity contains exactly one Java statement and branches and loops are displayed as transitions with guards. The roundtrip engineering also works if a developer makes manual changes in the source code as long as she/he uses the naming conventions and implementation concepts of Fujaba. The recognition of state-charts has not been mentioned here, because it works like the described process, as well.

3.1.4 Poseidon for UML™

Poseidon for UML supports roundtrip engineering. The support is currently restricted to class diagrams as many commercial vendors. Reverse engineering and code generation components exist but for the time being the roundtrip engineering works as follows:

The Poseidon user selects the files to reverse engineer. There is an option stating whether the *roundtrip engineering* component should be on for reverse-engineered files. If the user enables roundtrip engineering, then the component gets active. The user selects the time interval the files should be checked in the resource system for the changes. The interval is generally stated as seconds. During the checks, if changes on the file are found by the roundtrip component, the reverse engineering action runs again for the changed files. In this way, the UML model stays synchronized with the source code. As it is clear, the support is currently only 1-way. As the UML model changes, the source code is not updated. Only when the user executes code generation and selects the directory for the reverse-engineered files, the code gets overwritten.

This approach introduces some problems. The generated code overwrites all the source code. The developer/user can lose information which is not represented in the UML model.

3.1.5 Omondo EclipseUML Plugin

Omondo's EclipseUML [7] plugin is a widely used UML plugin for the Eclipse environment, exhibiting a complete integration with Eclipse. There is no standalone version of this UML tool.

The roundtrip engineering approach of Omondo is as follows: For a Java project, the user starts with reverse engineering. Complete reverse engineering happens only once. During the reverse engineering process, the UML model data is saved as Javadoc comments with XDoclets. All of the UML model information is available then in the Java source files. After the reverse engineering finishes, if the user changes the UML model that is open, the changes are triggered to the source code. For example, for a new class created as a UML model element, the corresponding source code is created. It is similar for the other UML model elements.

The changes that occur on the source code editor are not reflected on the UML model automatically. The user should select the UML model element, whose corresponding source code has changed, and update this element manually.

3.1.6 TogetherCC

TogetherCC (Together Control Center) [4] tool is often used by developers who want to keep the model and the source code in synch. The user can edit the UML model as well as the source code. The changes are propogated to the other. In case of creation of a new UML class, interface, operation or attribute the corresponding element in Java source code is created.

Dependency

Together distinguishes dependency relationships between UML model elements in the source code with added comments. For example, for a dependency between classes A and B, the following source code is added:

```
public class BaseClass {
  /** @link dependency */
    /*# Class1 lnkClass1; */
}
```

[4] version 6.2 is analyzed.

In the UML model, the class `BaseClass` is dependent on the class `Class1`.

Association

For association relationships, Together again adds Javadoc comments in the source code. The type property of an association has values *association*, *aggregation*, and *composition*. For the default[5] association created between classes `Class1` and `Class2` , the following source code is created:

```
public class Class1 {
  /**
       * @labelDirection forward
       * @supplierCardinality 0..*
       */
      private Class2 lnkClass2;
  }
```

As it is seen in the code snippet, the cardinalities are saved with the tags `@supplierCardinality` and `@clientCardinality`.

Aggregation

The default aggregation relationship is represented in the source code as follows:

```
  /**
       * @link aggregation
       */
  private Class1 lnkClass1;
```

Composition

A composition relationship between two interfaces:

```
 public interface Interface2 {
     void operation1();

     /**
      * @link aggregationByValue
      */
     /*# Interface1 lnkInterface1; */
 }
```

LinkByPattern (for association, aggregation and composition)

LinkByPattern icon adds association relationship to a UML model. The corresponding source code pattern can be created conforming to the selected pattern. The following figures, Figure 2-3.2 and Figure 2-3.3, show a portion of the patterns available for the aggregation relationship. Figure 2-3.3 shows a complete screen shot for *choose pattern* dialog. The dialog includes a list of patterns, the description for the selected pattern, the code preview and the parameters of the pattern. The selections basically consist of collection classes in different Java versions. The selections also include collections from JGL (The Generic Collection Library for Java) [8].

[5] See **LinkByPattern** Section, especially Figure 2-3.3.

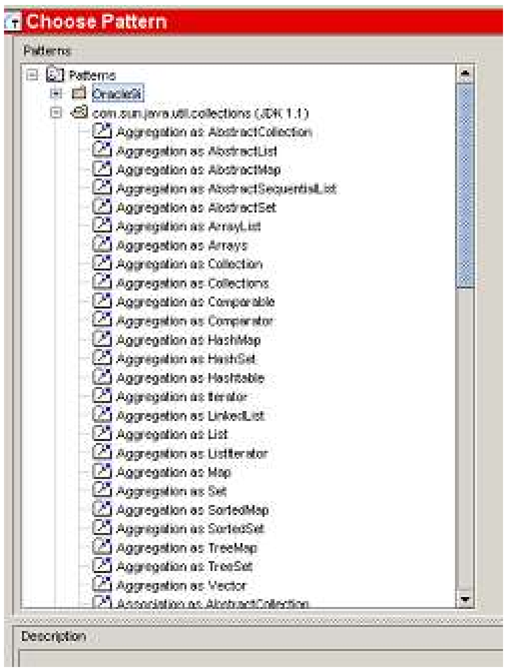
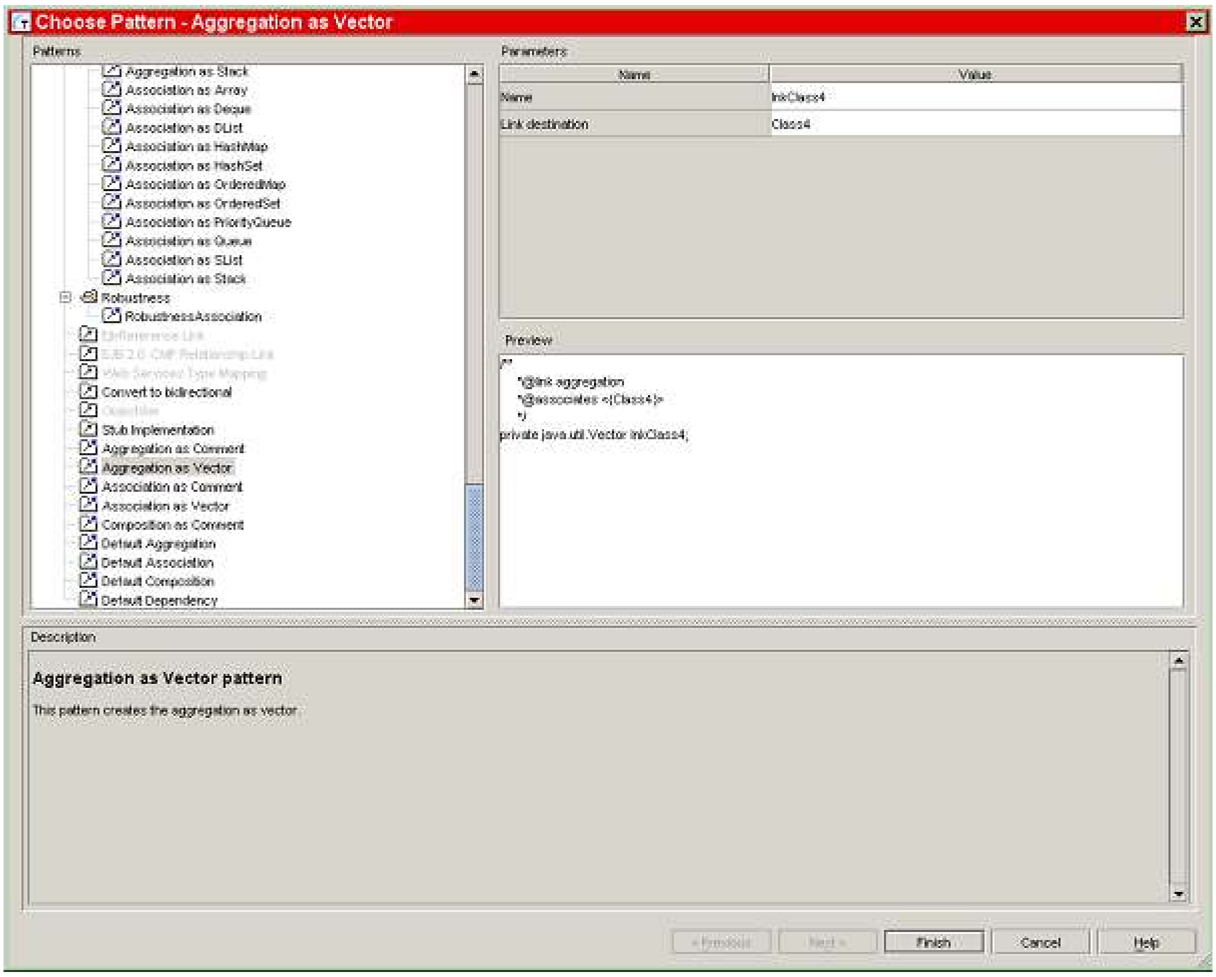

Choose Pattern
Patterns
Patterns
Oracle9i
com.sun.java.util.collections (JDK 1.1)
Aggregation as AbstractCollection
Aggregation as AbstractList
Aggregation as AbstractMap
Aggregation as AbstractSequentialList
Aggregation as AbstractSet
Aggregation as ArrayList
Aggregation as Arrays
Aggregation as Collection
Aggregation as Collections
Aggregation as Comparable
Aggregation as Comparator
Aggregation as HashMap
Aggregation as HashSet
Aggregation as Hashtable
Aggregation as Iterator
Aggregation as LinkedList
Aggregation as List
Aggregation as ListIterator
Aggregation as Map
Aggregation as Set
Aggregation as SortedMap
Aggregation as SortedSet
Aggregation as TreeMap
Aggregation as TreeSet
Aggregation as Vector
Association as AbstractCollection
Description

Figure 2-3.2 LinkByPattern Dialog



Name	Value
Name	lnkClass4
Link destination	Class4

```
/**
    *@link aggregation
    *@associates <{Class4}>
    */
private java.util.Vector lnkClass4;
```

Aggregation as Vector pattern

This pattern creates the aggregation as vector.

Figure 2-3.3 LinkByPattern Dialog (2)

In summary, for different kinds of associations (including aggregation and composition), Together provides a list of patterns. The code created for the association depends on the selected pattern. Some parts of the code templates for the patterns are also editable.

If the user writes code conforming to one of the patterns' code, the corresponding UML relationship is created in the UML model. But without the describing Javadoc tags and comments for a collection, the added UML model element would be a simple attribute.

Together solves the problem of mapping UML association, aggregation, composition and dependency relationships by adding comments to the source code that describe the relationships. The allowed multiplicities are 1, 0, * and 1..*. The code created is not always directly representing the meaning the UML relationship has, but even with some loss in represented information, this solution is one that works.

However, the added comments crowd the source code. So many lines of comments are added to represent an association relationship. Besides, the interaction with the source code editor, editing the source code directly is really slow. Sometimes only a selection in the source code editor took a long time. The Together environment, I think, is more suitable for UML model management, whose changes are propogated to the source code. As an IDE, it is not as rich as Eclipse or IntelliJ Idea.

3.2 Summary and Comments

Roundtrip engineering has many aids for the users. The rule that "*No design remains unchanged*" does not introduce problems if roundtripping is a part of the development process. Whenever a model change occurs, the implementation code is updated respectively and whenever the code change occurs, the design model is updated by reverse engineering.

Although roundtrip engineering is very useful, for big projects it introduces some problems even if no information loss happens in the reverse engineering. The UML models kept in synch contain so many details that they do not satisfy their own goals. The models do not help very much in understanding the system. The models are only a visual representation of the low-level source code. When roundtrip engineering is adopted, the models are in the same abstraction level as the source code. Having too detailed synchronized diagrams are almost as useless as the ones that do not exist; it is not easy to capture information from such diagrams.

UML is a very widely used modeling language. It is a de facto standard currently in the software industry as a modeling language. But as Martin Fowler defines, **UML as sketch** is often the way users use UML. This way of using UML does not necessitate roundtrip engineering. UML diagrams are only used to understand the system and show aspects to other team members in the software projects. It is basically a whiteboard usage of UML.

4

5 Reverse Engineering

This chapter describes reverse engineering in general and focuses on reverse engineering of Java source code to extract UML models. The UML model to be obtained is only limited to class diagrams considering the time limitations in this work as well as the fact that most research done is on class diagram reverse engineering.

5.1 Definition of Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify its current components and their dependencies to extract and create system abstractions and design information while the subject system is not altered; however, additional knowledge about the system is produced. The goal of the reverse engineering is to analyze the software systems in order to make the software more understandable for maintenance, evolution and re-engineering purposes.

The term reverse engineering finds its origins in hardware technology and denotes the process of obtaining the specification of complex hardware systems. Now the meaning of this notion has shifted to software. As far as known there is not (yet) a standard definition of what reverse engineering is but in [9] it is defined as: *The process of analyzing a subject system with two goals in mind:*

- *identify the system's components and their interrelationships; and,*
- *create representations of the system in another form or at a higher level of abstraction.*

Reverse engineering is a process of examination only: the software system under consideration is not modified.

Reverse engineering restricts itself to investigating a system. Adaptation of a system is beyond reverse engineering but within the scope of system renovation.

Reverse engineering supports program comprehension. Program comprehension helps in the process of maintenance, documentation, reuse and forward engineering of the target system. Program comprehension is supported by producing design models from the software. The static modeling of the target software by reverse engineering is called *static reverse engineering*. The dynamic behavior of the system is modeled by *dynamic reverse engineering*.

To understand existing systems, both static and dynamic information is necessary. The static information resides in the components, source code, and physical entities of the software systems whereas dynamic information is based on runtime behavior of the program objects which generally necessitates execution of the program for reverse engineering.

5.2 Related Work

This chapter will provide a survey of the literature. The focus of the survey is reverse engineering of Java systems to get *UML class diagrams*. As the review is done, the possible problems will be presented. The problems are especially important in context of roundtrip engineering as any information lost in reverse engineering introduces additional problems in roundtripping.

There has been a lot of research on reverse engineering of Java systems. Mainly the research is focused on obtaining the static structure of the programs for using the UML static models as documentation for the projects. Reverse engineering is a part of the roundtrip engineering process. Roundtrip engineering aims to synchronize source code and design models at any time. Especially reverse engineering will be reviewed in this context, in order to help in roundtrip engineering.

5.2.1 Static and dynamic Reverse Engineering

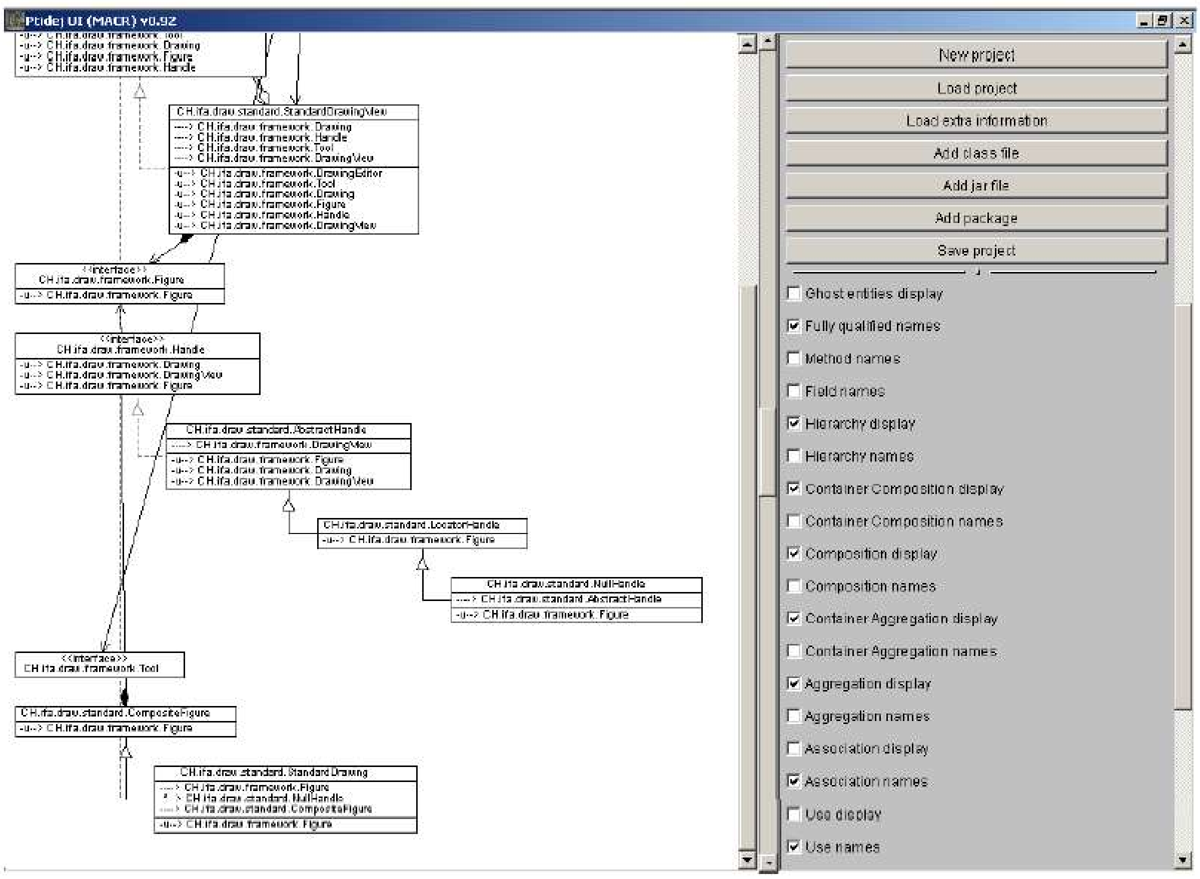
Survey of the literature up to now shows that there are many open issues in reverse engineering of Java systems to extract UML models. Static reverse engineering shows the main and basic structure of the systems. Dynamic reverse engineering is also a popular topic although there is not as much research done as static reverse engineering.

The work done in paper [10] consists of performing an exhaustive study of UML class diagrams constituents with respect to their recovery from C++, Java, and Smalltalk source code and implementation of a tool suite, Ptidej, to reverse engineer Java source code abstractly and precisely. For the analyzed problem in [10], they suggest a solution. The problem is so that UML class diagrams produced during design are often forgotten during implementation, under time pressure usually. Thus, they frequently present major discrepancies with implementation and are of little help to maintainers who must support released programs. Maintainers need means to recover UML class diagrams from programs' implementation. These means it must be *automated* considering the large size of programs and they must produce *abstract* yet *precise* class diagrams to help maintainers in their tasks. The criteria for good reverse engineering are proposed as abstractness and preciseness. The authors survey existing reverse-engineering tools and other tools with reverse-engineering capabilities, such as Chava [11], ArgoUML, IDEA, Rational Rose, TogetherJ, Womble [12]. They show that these tools produce neither abstract nor precise class diagrams with respect to source code.

Another paper by the same author Gu´eh´eneuc is [13]. In this paper, they propose a reverse engineering tool suite, PTIDEJ, to build precise class diagrams from Java programs, with respect to their implementation and behavior. They describe static and dynamic models of Java programs and algorithms to analyze these models and to build class diagrams. In particular, they describe their algorithms to infer use, association, aggregation, and composition relationships, because these relationships do not have precise definitions. Additionally they show that class diagrams obtained semi-automatically are similar to those obtained manually and more precise than those provided usually.

5.2.1.1 PTIDEJ

PTIDEJ (Pattern Trace Identification, Detection, And Enhancement in Java) is a reverse engineering tool suite to build class diagrams from static and dynamic models of Java programs semi-automatically. In [13], the author describes the tool and provides a concrete example of JHotDraw framework's reverse engineering. Then, he compares the class diagram given in this framework's documentation and the diagram obtained by reverse engineering using Ptidej. They show that the tool creates very precise class diagrams. They also do a review of related work, a review of the reverse engineering tools CHAVA, WOMBLE and the CASE tools ArgoUML and Rational Rose. They analyze the weaknesses of these tools.


Ptidej UI (MACR) v0.92
CH.ifa.draw.standard.StandardDrawingView
<<interface>> CH.ifa.draw.framework.Figure
<<interface>> CH.ifa.draw.framework.Handle
CH.ifa.draw.standard.AbstractHandle
CH.ifa.draw.standard.LocatorHandle
CH.ifa.draw.standard.NullHandle
<<interface>> CH.ifa.draw.framework.Tool
CH.ifa.draw.standard.CompositeFigure
CH.ifa.draw.standard.StandardDrawing
New project
Load project
Load extra information
Add class file
Add jar file
Add package
Save project

- [ ] Ghost entities display
- [x] Fully qualified names
- [ ] Method names
- [ ] Field names
- [x] Hierarchy display
- [ ] Hierarchy names
- [x] Container Composition display
- [ ] Container Composition names
- [x] Composition display
- [ ] Composition names
- [x] Container Aggregation display
- [ ] Container Aggregation names
- [x] Aggregation display
- [ ] Aggregation names
- [ ] Association display
- [x] Association names
- [ ] Use display
- [x] Use names

Figure 3-5.1 Top view of JHotDraw core classes in PTIDEJ and their concrete relationships [13]

Class diagrams created by Ptidej contain most of the UML constructs such as classes, interfaces, and relationships among them like use, composition and aggregation.

The tool uses three different models to represent static and dynamic data about the Java programs. The *static model* uses class files composing Java programs. Java class files contain all data program architecture and runtime behavior statically. The *dynamic model* uses traces as models of the runtime behavior of Java programs where a trace is a history of execution events: Field accesses-modifications; Class loads-unloads; Method, constructor, and finalizer entries-exits: Program end. A program has one static model but can have many (or an infinite number of) dynamic models. Finally, the *class diagram model* is a metamodel, PADL (*Pattern and Abstract-level Description Language*) [14], to describe programs as class diagrams. PADL offers constituents, such as Model, Class, Method, Relationship, which enables building class diagrams representing programs. It also offers methods to manipulate class diagrams easily and to generate other representations of class diagrams, using the Visitor design pattern.

The tool is comprised of three parts: PADL ClassFile creator, Caffeine and PTIDEJ. Static models are analyzed using the PADL ClassFile Creator tool, dynamic models using the Caffeine tool. PADL ClassFile Creator and Caffeine compute values of four minimal properties (exclusivity, lifetime, multiplicity, and invocation site) that they use to formalize the use, association, aggregation, and composition relationships. The authors also exemplified the Ptidej tool suite on a simple document description program and detailed its application on the JHotDraw framework. Additionally they showed that the class diagram obtained semi-automatically for the JHotDraw framework is more precise than the class diagram provided with the documentation from the authors. Figure 3-5.1 shows a UML class diagram obtained by using PTIDEJ.

For more information and details, refer to [13].

5.2.2 Design Patterns

Design patterns identification from the source code is a very hot topic. Design patterns provide accepted solutions to well known problems of the software systems. Using design patterns in design and implementation offers easier handling of the source code and application of proven solutions.

In [15] design patterns and constraints are used to automate the detection and correction of inter-class design defects. For this, the authors use a meta-model to describe design patterns, exploiting the descriptions to infer sets of detection and transformation rules. A constraints solver with explanations uses the descriptions and rules to recognize groups of entities with organizations similar to the described design patterns. A transformation engine modifies the source code to comply with the recognized distorted design patterns. As an example, they apply these guidelines on the Composite pattern using Ptidej, their prototype tool that integrates the complete guidelines they state in the paper.

The master thesis work of Zannier [16] presents complex refactorings based on existing tool-supported refactorings, knowledge of the application to be changed, knowledge of design patterns, and the capability to generate code. In her work, a proof of concept of tool support for complex refactoring to design patterns is detailed and empirical results in favor of such a tool are given.

In [17] again the design patterns are the main focus of research. The authors of the paper present a set of tools and techniques to help OO software practitioners design, understand, and re-engineer a piece of software, using design-patterns. A first prototype tool, Patterns- Box, provides assistance in designing the architecture of a new piece of software, while a second prototype tool, Ptidej, identifies design patterns used in an existing one. These tools, in combination, support maintenance by highlighting defects in an existing design, and by suggesting and applying corrections based on widely-accepted design patterns solutions.

A state of the art in UML-based static reverse engineering is given in [18]. This paper will be described in more detail in the next subsection. In this paper the reverse engineering capabilities of two industrial UML CASE tools as well as two research prototypes are analyzed. The results show that the static reverse engineering of the tools are still poor for the reasons which will be explained.

Most of the UML tools that support reverse engineering support it at the level of class diagrams. Only analysis of the static structure of the source code is done and this information is modeled in class diagrams. Some of the tools also support interaction diagram reverse engineering, but it is not widely supported. The runtime behavior of the objects can not be directly understood by parsing the source code. Execution of the system is necessary. Debugging is a special kind of execution. But if the debugging is adopted as part of reverse engineering, the reverse engineering process gets very slow and difficult to handle.

5.3 State of the Art for Static Reverse Engineering

In this section a comparison of the reverse engineering of the major CASE tools will be given. The tools include Rational, TogetherJ, Fujaba[19] and IDEA[20]. The compared reverse engineering includes only class diagram reverse engineering. Most of the UML CASE tools support only reverse engineering of class diagrams.

This comparison is based on the comparison given in [18]. The class diagrams in UML only show the static structure. It provides no information on behavior of the objects. The implementation of the methods is not part of the diagrams. This even hinders the manual learning of the behavior. The representation of the UML constructs in Java and vice versa is not always trivial as analyzed in Chapter 6. UML constructs like aggregation, composition and association do not have a direct mapping in Java. This makes the reverse engineering of the Java systems difficult. Generally the tools do not create any aggregation or composition relationships during reverse engineering. It is very difficult and subtle to analyze the source code for specific UML constructs. The tools are generally only limited to forming models with classes, interfaces, methods, attributes, generalization and implementation. Although many limitations exist in static reverse engineering, however the created UML model still helps a lot in understanding the static structure of the programs.

Nowadays, CASE tools increase their support for design patterns recovery from source code. But it has limitations.

The tools do not allow influencing the interpretations of the source code. The reverse engineering algorithm is completely buried in the tool's source code. This can cause misunderstanding of the reverse engineering. Especially if the tool also supports code generation, it should apply the same rules and concepts used for reverse engineering and code generation. For example a specific code pattern which is reversed as dependency should be followed in code generation of dependencies in a UML class diagram model.

The comparison based on [18] is done on a subject system Mathaino. The same criteria are used for all compared UML tools. The extracted model is compared by modeling tool TED.

5.3.1 Metrics used for Comparison

The following model properties are used as criteria for comparison:

- Number of Classes - can be counted in various ways (e.g. inclusion of nested classes or container classes)
- Number/Types/Multiplicities/Roles of Associations -- associations (directed, bi-directional), aggregation, composition
- Handling of Interfaces - which classes realize or require?
- Handling of Java Collection Classes - An implementation-specific means to handle collections of objects that don't generally show up in design
- Inner classes - implementation-specific, not reflected in UML
- Class Compartment Details - level of detail when resolving method signatures.

5.3.2 Results of Comparison

A summary is given for the results of the comparison in the following table, Table 3-5.1.

Tool / **Metric**	**Rose**	**Together**	**FUJABA**	**IDEA**
Number of Classes (39)	39	39	39	39
Named Inner Classes (3)	3	3	3	3
Anonymous Inner Classes (42)	42	42 (using *.class files)	0	0
Class Details	Name of the actual class and its package; attribute and operations names, types, and visibility (public, protected, or private). For each operation, the parameter types are also given.			

Interfaces	4	4	Yes (amount not specified)	4
Interface Realization	All	All	Yes (amount not specified)	Not specified.
Interface Dependency	None	None	Metrics reported by model not updated to reflect dependency	Not specified.
Associations	83 (45 represent relationship between nested class and its parent) Always directed.	16 (using *.java files)	None found because Mathaino doesn't use expected coding standards	47
Associations - Role Names	Yes - on object end of association.	Not supported	None found because Mathaino doesn't use expected coding standards	Yes - no details
Associations - Multiplicity	Not supported.	Not supported.	Non-primitives get 0..1 and containers get 0..n	Yes- no details
Aggregation	0	0	None found because Mathaino doesn't use expected coding standards.	No
Composition	0	0	None found because Mathaino doesn't use expected coding standards.	No
Java Collection Classes	Written to attribute compartment of the class.	Written to attribute compartment of the class.	None found because Mathaino doesn't use expected coding standards.	16

Table 3-5.1 Common elements found by the CASE tools [21]

All of the examined tools succeeded in recognizing the basic UML features like classes, interfaces and associations. Only in one case TogetherJ failed to recognize a part of the plain associations. At this level, the results could be compared easily using metrics like number of classes (NOC) and number of associations (NOA). The numbers were generally identical for all tools and the occurring differences could be explained by the different approaches or ways to count. For example, anonymous inner classes and package-external classes are handled differently by each tool. Similar differences existed for associations. Concluding, the creation of a correct implementation-view representation could be handled by all four tools. [18]

The industrial tools Together and Rational are very poor at recognition of advance features and getting abstract representation. The extracted model does not contain elements like multiplicities, inverse associations and container resolution. The support for these elements is better at the research prototypes Fujaba and IDEA. If the source code is conforming a predefined template code the prototypes follow, they can produce those elements.

The static reverse engineering support for arbitrary code is still an open issue. With newer versions of the industrial tools no improvements are done. The reason for this is mainly the subtlety in the mapping between UML and Java. Some newer tools like Omondo provide better reverse engineering support for source code having Javadoc tags describing the functionality of the UML model in the source code.

5.4 Summary

This chapter has described techniques and proposals in the field of reverse engineering. After defining reverse engineering, a classification into two parts can be made: Static vs. Dynamic reverse engineering. The focus in this thesis work is static reverse engineering; therefore dynamic reverse engineering is not analyzed in detail. Static reverse engineering is part of the roundtrip engineering process. The recognition of the static structure of the source code during roundtripping is the half of the roundtripping process. The other half is analyzed in the chapter, Code Generation.

Although the static reverse engineering with class diagram extraction is only a subset of the reverse engineering process, it is still not very well supported by CASE tools. Some research prototypes analyzed [19, 20] support reverse engineering better. To some extent, they also abstract the source code as a result of reverse engineering. But they highly depend on the specific patterns in the source code. The reverse engineering of arbitrary source code is not well supported in industrial tools.

Although the reverse engineering is incomplete for almost all available tools, it should be stated that it helps in understanding the static structure of the Java programs as well as as part of roundtripping. The generalization and implementation relationships can be well represented and reverse engineered. The extracted model is supposed to be complete with respect to the inheritance hierarchy reverse engineering.

Industrial tools that support full roundtripping between Java source code and UML class diagrams like TogetherJ save the UML model information in the source code when generating code from UML models. This allows for the full reversing of the source code without losing UML model details. But this is only a special case. The full complete reverse engineering for arbitrary code is not possible.

Mapping problems

The main reason for the missing support in static reverse engineering is the problem that UML and Java constructs do not have a 1-1 mapping. Some constructs such as aggregation, composition or associations do not have Java mappings (at the language level). Aggregation and composition are implemented similarly although they are completely different concepts in the UML. Most of the CASE tools analyzed produce no aggregation or composition relationships in the reverse engineering. Another big problem is collections. In Java, collections can have heterogeneous elements: one element of type `Car` and another of type `House` is totally valid in Java. This introduces problems to the static reverse engineering. The Java collections are generally represented as attributes of the owning class although at the high level design, they represent the associations with ends having multiplicities of *many (*)*. When defining attributes of type collections in a Java class, at the time of the definition the type of the objects that can exist in the collection is not known. A deeper analysis of the source code is necessary for getting the type of collection elements. Many of the CASE tools omit this step. Java 1.5 has typed collections and seems to have solved some of the problems met in static reverse engineering regarding collections. Java 1.5 is not yet finalized and released, and by no means is it supported yet in reverse engineering tools.

6 UML™ and Poseidon for UML™

This chapter concentrates on the concepts that should be understood to follow the rest of this work. As this work concentrates on the synchronization between UML and Java, it is necessary to understand UML well. This chapter provides an overview of the UML, more details on UML in the context of synchronization will be described in the following chapters.

6.1 UML (The Unified Modeling Language)

The UML is a family of graphical notations, backed by a single meta-model, which helps in describing and designing software systems, particularly object-oriented software system. [22]

UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems. [23]

With UML static as well as dynamic information of software systems can be modeled. The UML is an informal language where the emphasis is on the graphical notation for the representation of the software designs. The UML is easy to understand and learn.

The work on UML was started by Grady Booch and James Rumbaugh at Rational Software in 1994.The goal they had was to unite the Booch method and the OMT-2 method of which Rumbaugh was the leading developer. Later in 1995 Ivar Jacobson also joined them and the work on UML went on fast. Version 1.0 of the UML was released at January 1997. The "three amigos" aimed to make the UML a de facto standard, but as the OMG made a request to adopt it as a standard modeling language, it became more and widely adopted by software industry.[24] So, The UML is a relatively open standard, controlled by the Object Management Group, an open consortium of companies.

As a modeling language the UML offers the users possibilities as complex software and system design and combining the ideas and exchange them. The designers need to understand the modeling language, but not the software process. The UML is not part of a software process; it can be adopted by any process life cycle.

For the description of designs specialized at specific parts of the system, the UML provides many different diagram types as class diagrams, sequence diagrams or component diagrams. The different views UML provides enable the user to see the system by selecting the focused view.

6.1.1 Diagrams in UML

The UML defines 12 types of diagrams w.r.t models:

- *Use case diagram:* Use case diagram shows a set of use cases, actors and their relationships. It shows view of static use case, which is important for organizing and defining system activities.
- *Class diagram*: Class diagrams show relationships among a set of classes, interfaces, and collaborations. It is the mostly used diagram type in the UML. It shows the static design of a system.
- Behavior diagrams:

  - *State chart diagram*: It shows a state machine that is made up from states, transitions, activities and events. Dynamic view of a system is shown by state chart diagrams.
  - *Activity diagram*: It is a special form of state chart diagrams that shows sequence of one activity to another activity in a system. Control flow is emphasized.
  - interaction diagrams:
    - *Sequence diagram*: Sequence diagram and collaboration diagrams show the interactions between objects. The sequence of the messages sent between the objects is emphasized.
    - *Collaboration diagram*: It emphasizes the collaborations between the objects.
- Implementation diagrams:
  - *Component diagram*: Component diagrams show organization and dependencies among a set of components that relate to class diagrams.
  - *Deployment diagram*: A deployment diagram shows configuration of a node as well as its components.

Although other names are sometimes given to these diagrams, this list constitutes the canonical diagram names.

These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees, although the UML and supporting tools will provide for a number of derivative views.[23]

In this work, the concentration is on the class diagrams. The synchronization approaches analyzed are based on class diagram reverse engineering and code generation from class diagrams.

6.1.2 UML Metamodel

The UML in its current state defines a notation and a metamodel. The notation is the graphical facade of models; it is the graphical syntax of the modeling language. For instance, the notation for the class diagrams in UML defines how concepts such as class, interfaces, relationships, associations, etc. are represented. The UML metamodel defines the UML elements. The UML metamodel is especially important for those who use the *UML as a programming language*[6] as the metamodel defines the abstract syntax of the language.

The architecture of the UML is based on a *four-layer metamodel* structure, which consists of the following layers: user objects, model, metamodel, and meta-metamodel.

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture. This architecture is a proven infrastructure for defining the precise semantics required by complex models. There are several other advantages associated with this approach:

- It refines semantic constructs by recursively applying them to successive metalayers.
- It provides an architectural basis for defining future UML metamodel extensions.
- It furnishes an architectural basis for aligning the UML metamodel with other standards based on a four-layer metamodeling architecture, in particular the OMG Meta-Object Facility (MOF).[25]

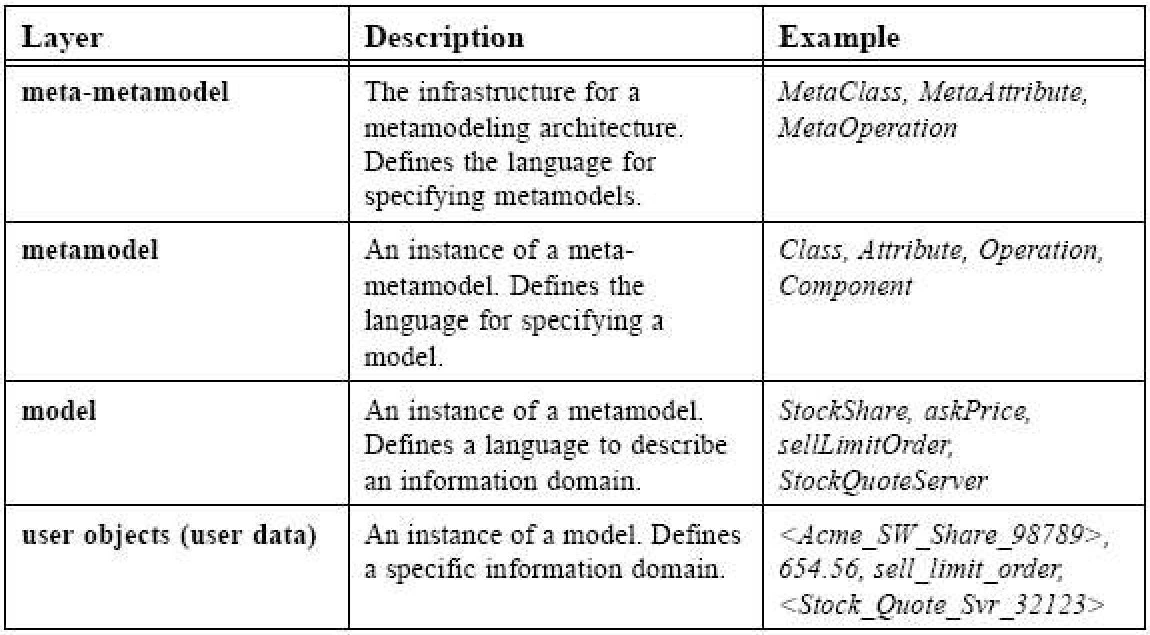
[6] See [22],UML Distilled, page 3

The UML language is defined using the MOF. MOF is used to define metadata, such as the structure of an object repository or a modeling language. The MOF is an OMG standard defining a common, abstract language for the specification of metamodels. MOF is an example of a meta-metamodel, or model of the metamodel. The UML language, including each UML model element such as class, attribute or dependency, is defined in a document called the UML metamodel. The UML metamodel is described using MOF. Actually, MOF is similar to the core subset of UML, what is usually called UML class diagrams. Tools that understand the MOF standard can manipulate any MOF-based modeling language, including UML and future versions and extensions to UML.

The generally accepted framework for metamodeling is based on architecture with four layers:

- *meta-metamodel*
- *metamodel*
- *model*
- *user objects*

The functions of these layers are summarized in the following table.



Layer	Description	Example
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	*MetaClass, MetaAttribute, MetaOperation*
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
model	An instance of a metamodel. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*
user objects (user data)	An instance of a model. Defines a specific information domain.	*<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>*

Figure 4-6.1 Four layer Metamodeling architecture[23]

The meta-metamodeling layer forms the foundation for the metamodeling architecture. The primary responsibility of this layer is to define the language for specifying a metamodel. A meta-metamodel defines a model at a higher level of abstraction than a metamodel, and is typically more compact than the metamodel that it describes. A meta-metamodel can define multiple metamodels, and there can be multiple metametamodels associated with each metamodel. While it is generally desirable that related metamodels and meta-metamodels share common design philosophies and constructs, this is not a strict rule. Each layer needs to maintain its own design integrity. Examples of meta-metaobjects in the metametamodeling layer are: MetaClass, MetaAttribute, and MetaOperation. A metamodel is an instance of a meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. Metamodels are typically more elaborate than the meta-metamodels that describe them, especially when they define dynamic semantics. Examples of metaobjects in the metamodeling layer are: Class, Attribute, Operation, and Component.

A model is an instance of a metamodel. The primary responsibility of the model layer is to define a language that describes an information domain. Examples of objects in the modeling layer are: StockShare, askPrice, sellLimitOrder, and StockQuoteServer. User objects (a.k.a. user data) are an instance of a model. The primary responsibility of the user objects layer is to describe a specific information domain.

Both the UML and the MOF are based on a four-layer metamodel architecture, where the MOF meta-metamodel is the meta-metamodel for the UML metamodel. Since the MOF and UML have different scopes and differ in their abstraction levels (the UML metamodel tends to be more of a logical model than the MOF meta-metamodel), they are related by loose metamodeling rather than strict metamodeling. As a result, the UML metamodel is an instance of the MOF meta-metamodel. Consequently, there is not a strict isomorphic instance-of mapping between all the MOF meta-metamodel elements and the UML metamodel elements. In spite of this, since the two models were designed to be interoperable, the UML Core package metamodel and the MOF meta-metamodel are structurally quite similar.

6.1.3 Class Diagrams

In this section the class diagram elements that are most relevant to this thesis work will be described.

Class diagrams are structural diagrams of the UML. They take a central place during object-oriented modeling process. The structure of the systems is described using classes, interfaces and the relationships between them.

A class diagram is composed of classes, interfaces, operations and attributes of them, the relationships between classes and interfaces as associations, dependencies, generalizations, implementations, compositions and aggregations. Stereotypes and tagged values are also part of class diagrams.

A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. The UML uses the term **feature** as a general term that covers properties and operations of a class.[22]

Now the main elements of a class diagram will be described:

Class

They are the central elements of a class diagram. A class defines internal data of an object by attributes and operations. In UML, a class has separate compartments describing its operations, attributes and the name compartment.

Interface

Interfaces define the structure of an object without implementation details. An interface is a class that has no implementation. All features of interfaces are abstract. Classes implementing interfaces contain the implementation for the operations defined in the interfaces. Only the signature of the operations is defined in the interfaces.

In UML interfaces are expressed with the standard class symbol and the stereotype `<<Interface>>` describing it. The interfaces contain only two compartments, the name compartment and the operations compartment.

Package

A *package* is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain subordinate packages as well as other kinds of model elements. All kinds of UML model elements can be organized into packages.[23]

In class diagrams, packages are used to group classes, interfaces and packages themselves. It defines a namespace for these elements.

Generalization

Generalizations define inheritance relationships between classes or interfaces. This relationship defines an inheritance relationship between a general and a specialized class.

A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information.[23]

Realization

Realization relationship exists between an interface and the class implementing the interface.

Association

Associations represent communication connections between objects. When two classes are connected with each other, the association is said to be *binary*. In this work, only binary associations are considered.

An association declares a connection (link) between instances of the associated classifiers (e.g., classes). It consists of at least two association ends, each specifying a connected classifier and a set of properties that must be fulfilled for the relationship to be valid. The multiplicity property of an association end specifies how many instances of the classifier at a given end (the one bearing the multiplicity value) may be associated with a single instance of the classifier at the other end. A multiplicity is a range of nonnegative integers. The association end also states whether or not the connection may be traversed towards the instance playing that role in the connection (isNavigable), for instance, if the instance is directly reachable via the association. An association-end also specifies whether or not an instance playing that role in a connection may be replaced by another instance.[23]

Associations notate the properties of a class. Much of the same information that you can show on an attribute appears on an association.[7]

Associations are described with more details in the coming chapters as it is emphasized on this work. The multiplicities and types of associations are described later.

Aggregation and Composition

Aggregation is the part-of relationship. Composition defines a is-a relationship. It says, although a class may be a component of many other classes, any instance must be a component of only one owner. The "no-sharing" rule is the key to composition. An assumption on compositions is that, when an owner object instance is deleted, also the objects that are owned by it by composition relationship should also be deleted.

Dependency

[7] See [22]. UML Distilled, page 37

A dependency relationship between different classifiers[8] emphasizes the dependencies between them. It expresses that a classifier depending on another classifier can be affected if the classifier it depends on changes.

A dependency exists between two elements if changes to the definition of one element (the supplier) may cause changes to the other (the client). With classes, dependencies exist for various reasons: One class sends a message to another; one class has another as part of its data; one class mentions another parameter to an operation. If a class changes its interface, any message sent to that class may no longer be valid.[9]

Stereotypes

Generally a stereotype represents a usage distinction for UML metamodel elements. A stereotyped element may have additional constraints on it from the base metamodel class.

6.2 Poseidon for UML: The UML Modeling Tool

Poseidon for UML, shortly called "Poseidon", is a modeling tool which is developed based on open source project ArgoUML[10] by Gentleware AG.

Poseidon has several editions (Community, Standard, Professional, Enterprise und Embedded). They answer different type of clients' needs. The current version of Poseidon is 2.5[11] and it supports all diagram types of UML version 1.4. Support of UML 2.0 is one of the main future plans.

In the next sections the components of Poseidon will be shortly described. For detailed information on Poseidon, please refer to [26] or the product itself.

6.2.1 Graphical User Interface

Poseidon has 4 main panels which are shown in Figure 4-6.2: The navigation panel, the details panel, the diagram panel and, the last one, the overview panel.

Navigation Pane

The top left corner of the Poseidon tool is the Navigation panel where all of the UML model elements can be seen in form of a tree. This panel is used to access all of the main parts of a model by presenting the elements of the model in various tree structures. The navigation panel offers many different view types which are specified in UML specifications. The UML model elements can also be edited directly by the tree nodes seen in this panel.

Diagram Pane

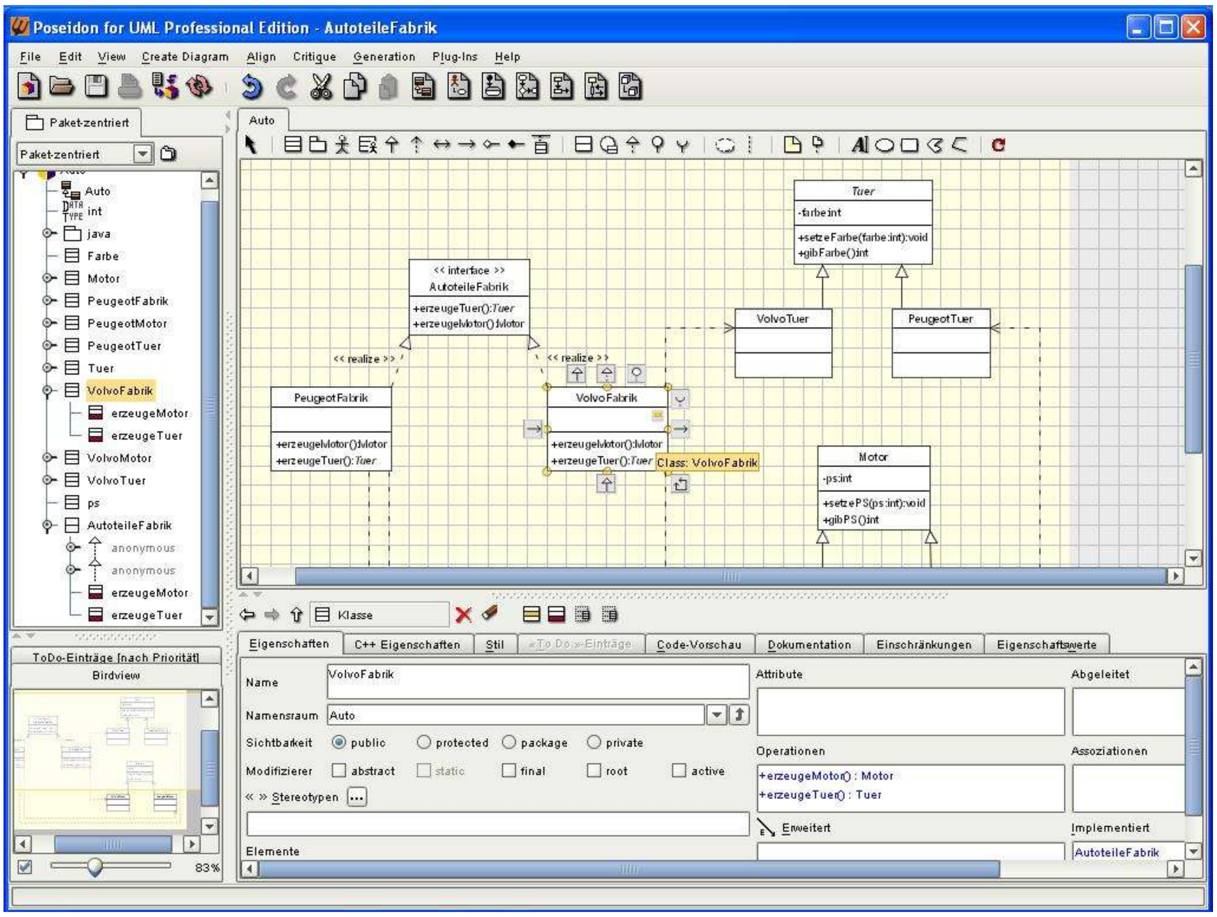
The diagram panel is the main panel of Poseidon. It is on the top right corner. This panel enables the users to design the UML models with the model elements and notations defined in the specification.

[8] Classifier is the superclass of Class, Interface and DataType in UML metamodel.
[9] See UML Distilled, page 47
[10] argouml.tigris.org is the web site of this project.
[11] September 2004

Details Pane

The Details pane provides access to all of the aspect of the model elements. It is on the bottom-right corner. Within this pane, the user can view and modify properties of the elements, define additional properties, and navigate between elements

The *properties* tab in the details pane is the most important one. It is selected by default. The Properties tab looks a little different for each different type of model element as Class, Interface, Package, Operation, etc.



Name: VolvoFabrik
Namensraum: Auto
Sichtbarkeit: [✓ public] protected package private
Modifizierer: abstract static final root active
« » Stereotypen
Elemente
Attribute
Operationen: +erzeugeMotor() : Motor, +erzeugeTuer() : Tuer
Erweitert
Abgeleitet
Assoziationen
Implementiert: AutoteileFabrik

Figure 4-6.2 Poseidon - Professional edition

Overview Pane

The overview pane is on the bottom-left corner of Poseidon. It offers the users to see the snapshots of the active diagram edited. It has a zoom possibility for the active UML diagram, by which the level of details to be seen in the diagram can be adjusted.

6.2.2 Main Components and Functionality

Poseidon is a UML tool completely written in Java. The UML models that are designed with Poseidon conform to the UML metamodel found in the UML specification [23].

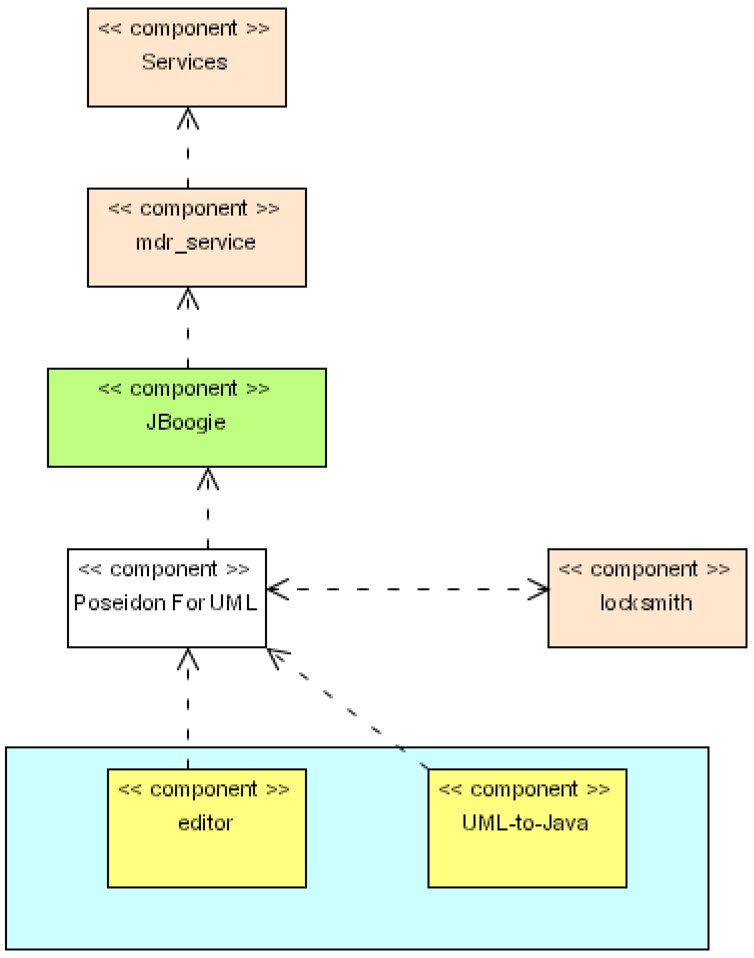


Figure 4-6.3 Poseidon Framework

The main components Poseidon depend and use are **MDR** and **JBoogie** framework as seen in Figure 4-6.3. The components **mdr_service** and **Services** offer the functionality to manage UML data.

The UML 1.4 specification can be written down in XML using the XMI DTD. Poseidon writes XMI 1.2 and reads XMI 1.0, 1.1 and 1.2. UML 1.4 can be formally defined using MOF (Meta-Object Facility). This, in combination with JMI (Java Metadata Interface), defines the shape of the Java interfaces representing the UML metamodel. MDR (Metadata Repository) complies with the XMI, MOF, and JMI standards. MDR is the component that Poseidon uses for management of the UML data. [27]

MDR implements the OMG's MOF standard based metadata repository. The UML metamodel is M2 level of MOF levels. The UML models the users create are on M1 level. The UML metamodel[12] persistence as well as UML models' persistence, shortly UML data management is done by MDR component in Poseidon. UML models are instances of UML metamodels. To create, edit, delete model elements from UML models, UML1.4 API generated by MDR is used.

JBoogie framework is implemented as part of the thesis work [28]. It handles all diagram-related functionality. As part of the work done in [28], a metamodel to define UML diagrams is developed. This metamodel is now in the UML2.0 specification. *Diagram Interchange* (DI) [29] metamodel is the mentioned developed metamodel. It has model elements as diagram, size, color, etc. specifying all aspects of UML diagrams. The main aim of this metamodel is to enable exchange of UML semantic model as well as UML diagram

[12] Up to UML1.4 metamodel as the report is written

information. To support this aim, the UML data and diagram data are saved in XMI (XML Metadata Interchange) [30] format.

The component **locksmith** handles main licensing operations for Poseidon versions and editions.

The components **editor** and **UML-to-Java** shown are plugins for Poseidon. Poseidon can be extended via the open API it defines. The core functionality of Poseidon is independent of plugins. Most of the plugins are code generation components. Plugins that offer UML documentation of Java source code as well as reverse engineering are also available.

6.2.3 UML Profiles in Poseidon

Profiles extend the UML through Stereotypes that are already defined. These stereotypes form a vocabulary and define a subset of UML metamodel. Within the UML standard, the specific way to define the use of the UML can be defined in a UML profile. A UML profile defines a subset of the UML model elements, specializations of UML concepts, limitations and specific requirements for the used concepts and finally the limitations and the requirements for the concepts that are used. Profiles can also define extra attributes which can be added to the UML models.

Poseidon offers in total 9 profiles for different programming languages and standards through plugins. Java, C++, C#, Delphi, SQL, PHP, etc., are some of the profiles that can be found. These profiles define data types, classes etc., that exist for the specified programming language or the standard. The user can then directly use them.

The profiles in Poseidon application can be enabled and disabled through *Profiles* panel in *Plugins* menu.

7 Java™ and the Eclipse™ Platform

7.1 What is Eclipse

The Eclipse Platform is designed for building integrated development environments (IDEs) that can be used to create applications as diverse as web sites, embedded Java^{TM} programs, C++ programs, and Enterprise JavaBeans^{TM}. The Eclipse Platform is an IDE for anything and for nothing in particular.

The Eclipse Platform which has a lot of built-in functionality can be extended by additional tools to enhance the platform. The platform provides many extension points that plugin developers can extend. The whole Eclipse platform is based on plugins. Even most of the platform core functionality is implemented as plugins. The mechanism for discovering, integrating and running modules is done via plugins.

The Eclipse Platform is designed and built to meet the following requirements (from []):

- Support the construction of a variety of tools for application development.
- Support an unrestricted set of tool providers, including independent software vendors (ISVs).
- Support tools to manipulate arbitrary content types (e.g., HTML, Java, C, JSP, EJB, XML, and GIF).
- Facilitate seamless integration of tools within and across different content types and tool providers.
- Support both GUI and non-GUI-based application development environments.
- Run on a wide range of operating systems, including Windows$^{®}$ and Linux^{TM}.
- Capitalize on the popularity of the Java programming language for writing tools.

Eclipse is a collection of places where additional functionalities can be inserted. These are called extension points and the functionality it provides is called extensions. The role of the Eclipse Platform is mainly to provide tool mechanisms to seamlessly integrate to it.

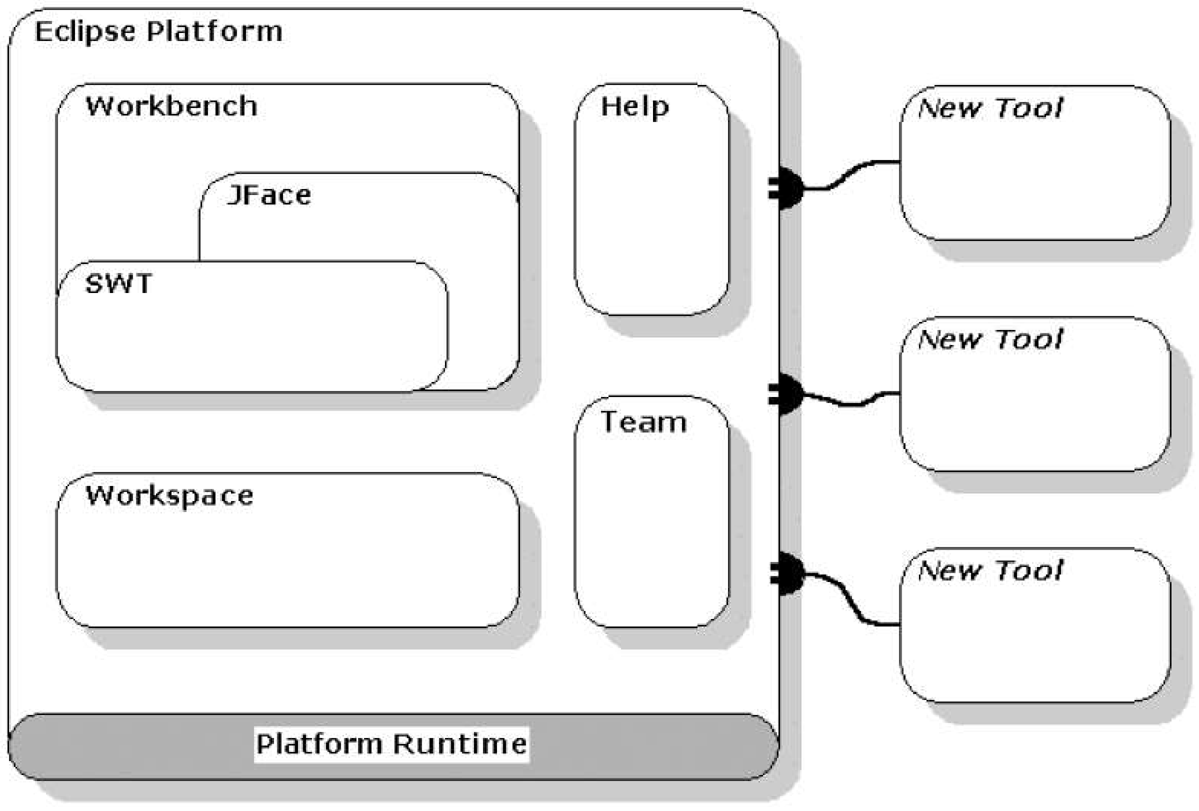


Figure 5-7.1 Eclipse Platform Architecture[31]

Figure 5-7.1 shows the Eclipse Platform architecture. As it can be seen in the figure, the Eclipse Platform's main component is the Platform Runtime. The Eclipse Workbench, Workspace and other parts of the platform are based on the Platform runtime. New tools can plug into the platform by extending the right extension points.

7.2 Platform RunTime and the Plugin Architecture of Eclipse

A *plugin* is the smallest unit of Eclipse Platform function that can be developed and delivered separately. The Eclipse platform itself is also composed of many plugins. Only the main functionality provided by Platform Runtime is not implemented as plugins. It is the kernel of the Eclipse Platform. The Platform Runtime component defines the plugin infrastructure.

The Eclipse plugins are written in Java and have a manifest file which declares the relationships and dependencies to other Eclipse plugins. The manifest file is called *plugin.xml*. As Eclipse defines extension points and plugins extend them, plugins themselves can also define new extension points and make them available for other plugin developers. This enables the platform to support unlimited plugins.

On start-up, the Platform Runtime discovers the available plugins and reads their manifest files. All plugins of Eclipse reside in a directory */plugins* relative to Eclipse installation. Any installed plugin can be found there. After reading the manifest files, the Platform Runtime builds a plugin registry which stays in memory. The plugins in Eclipse are only loaded if the functionality they provide is executed. The plugin registry which resides in memory after start-up is available through APIs. Whenever a new plugin is installed, the Eclipse Platform should be restarted. ***Plugins cannot be added after start-up.***

"Lazy Loading Rule: Contributions are only loaded when they are needed."

The dependencies of a plugin to other plugins are defined explicitly in the manifest file. Besides, a plugin defines the visibilities of the classes in its libraries in the manifest file.

7.3 Basic Eclipse Concepts and Components

In this section, the goal is to introduce the basic Platform components, libraries and concepts.

Workspaces

The Workspace manages one or more projects of the highest level. A project is composed of resources like files and folders. The plugins added to the Eclipse Platform operate on a user's workspace. A project maps to a single root directory on a file system.

Workbench and UI Toolkits

The Workbench defines the overall structure of the Eclipse user interface, i.e. its editors, views, perspectives etc., which are extensible.

The workbench API and implementation are built from the toolkits SWT and JFace. SWT is a widget set and graphics library which is integrated with the native window system. Still it has an OS-independent API. The JFace toolkit is implemented by SWT itself. It offers a higher level API than the SWT API for the plugin developers.

As the workbench is the central place where Eclipse and its plugins meet, it is known as the Eclipse Platform UI. The window seen when Eclipse runs is the Workbench. The workbench is completely implemented with SWT[13] and JFace.

Standard Widget Toolkit (SWT)

The SWT provides the graphical functions and widgets to the plugins. The API is independent of the operating system (OS) as Swing library. But the independency means is different for SWT and Swing. SWT uses the native GUI widgets of the OS as long as they exist. Whenever they don't exist, SWT internally implements them and provides to the users. On the other hand, Swing is completely independent of the native widgets. As Java programming language was designed, the main aim was the platform (OS) independency. The user interface library is also respectively designed and implemented. All of the widgets Swing provides are implemented by the Swing library, i.e., no use of OS native widgets is required.

Up to Eclipse version 3.0, the tools that plug in to the platform had to use SWT, but since version 3.0 the tools can also be written with Swing and AWT. Eclipse now provides a mechanism to integrate Swing-based plugins to the platform.

JFace

JFace is a UI toolkit and framework designed for handling common UI tasks. It is based on the SWT API and is window-system-independent. The JFace API provides classes for tasks as font registries dialog and preference windows, progress bars, etc.

The main components of the Eclipse Platform excluding the Java Development Tools (JDT) are described in this section. JDT is an extensible Java IDE. The details are in the next sections.

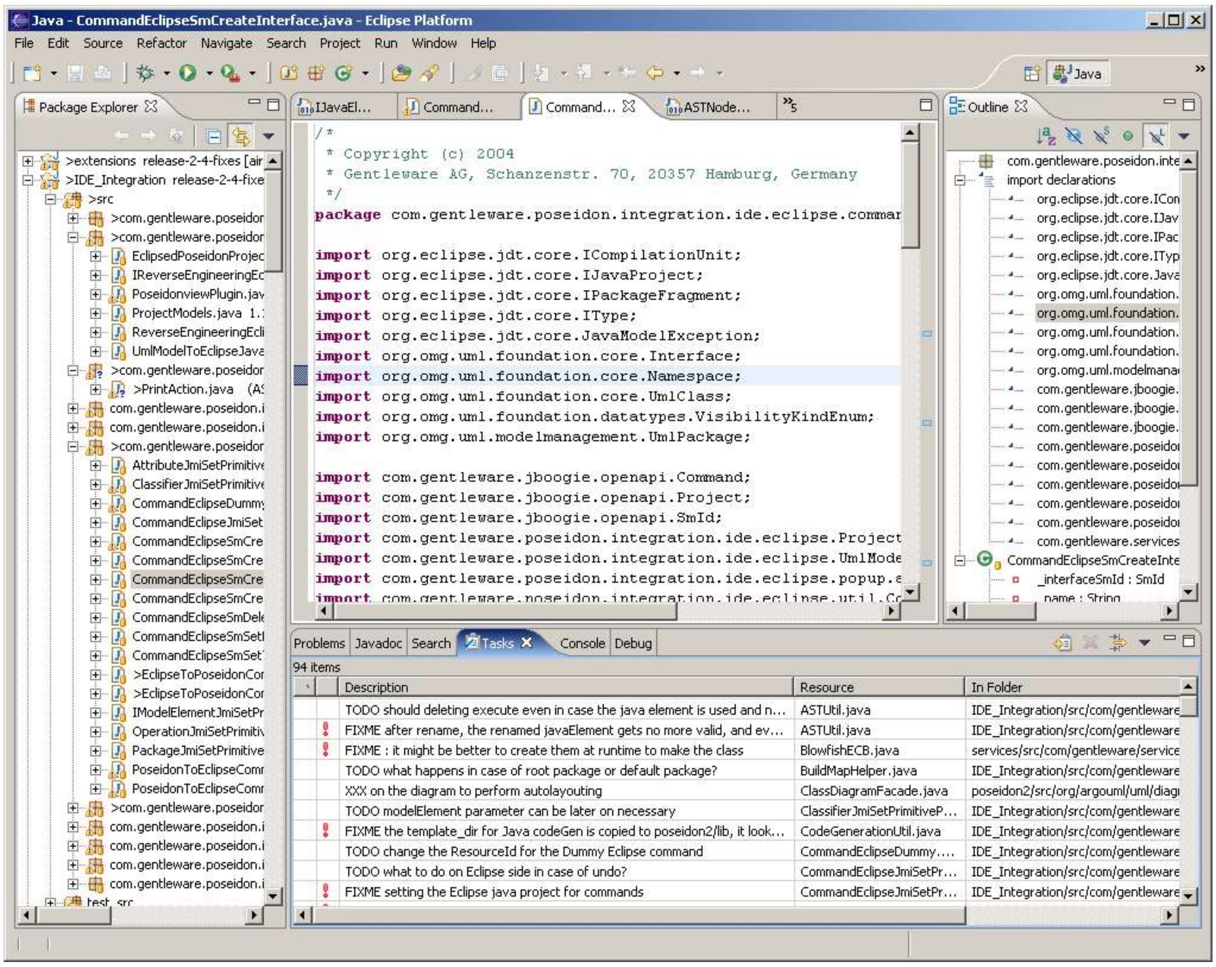
[13] SWT is implemented without the Java Swing library. So, the Eclipse Platform UI is Swing-independent.

7.4 Java Development Tools (JDT)

The Java Development Tools is the tool which is part of the Eclipse platform enabling users to develop Java programs as well as extend the JDT itself. It is an integrated development environment (IDE) for Java in the Eclipse Platform.

JDT environment can be seen in Figure 5-7.2. The Outline view provides the general outline of the active Java editor, showing the method signatures, fields, import statements etc. The Package Explorer view provides an overview of the existing Java projects in the Eclipse workspace. Resources belonging to the projects can be found under the project tree where the project name is the root node of the tree. Various Java source files, class files and many more resources can be project resources.

The JDT provides many useful features for Java developers such as refactoring, searching, browsing, comparing, running and debugging. Refactoring capability is one of the most useful and popular features for any Java development environment.


Java - CommandEclipseSmCreateInterface.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
IJavaEl... Command... Command... ASTNode...
Outline
Problems Javadoc Search Tasks Console Debug

```
/*
 * Copyright (c) 2004
 * Gentleware AG, Schanzenstr. 70, 20357 Hamburg, Germany
 */
package com.gentleware.poseidon.integration.ide.eclipse.commar

import org.eclipse.jdt.core.ICompilationUnit;
import org.eclipse.jdt.core.IJavaProject;
import org.eclipse.jdt.core.IPackageFragment;
import org.eclipse.jdt.core.IType;
import org.eclipse.jdt.core.JavaModelException;
import org.omg.uml.foundation.core.Interface;
import org.omg.uml.foundation.core.Namespace;
import org.omg.uml.foundation.core.UmlClass;
import org.omg.uml.foundation.datatypes.VisibilityKindEnum;
import org.omg.uml.modelmanagement.UmlPackage;

import com.gentleware.jboogie.openapi.Command;
import com.gentleware.jboogie.openapi.Project;
import com.gentleware.jboogie.openapi.SmId;
import com.gentleware.poseidon.integration.ide.eclipse.Project
import com.gentleware.poseidon.integration.ide.eclipse.UmlMode
import com.gentleware.poseidon.integration.ide.eclipse.popup.a
import com.gentleware.poseidon.integration.ide.eclipse.util.Cc
```

94 items

	Description	Resource	In Folder
	TODO should deleting execute even in case the java element is used and n...	ASTUtil.java	IDE_Integration/src/com/gentleware
!	FIXME after rename, the renamed javaElement gets no more valid, and ev...	ASTUtil.java	IDE_Integration/src/com/gentleware
!	FIXME : it might be better to create them at runtime to make the class	BlowfishECB.java	services/src/com/gentleware/service
	TODO what happens in case of root package or default package?	BuildMapHelper.java	IDE_Integration/src/com/gentleware
	XXX on the diagram to perform autolayouting	ClassDiagramFacade.java	poseidon2/src/org/argouml/uml/diagi
	TODO modelElement parameter can be later on necessary	ClassifierJmiSetPrimitiveP...	IDE_Integration/src/com/gentleware
!	FIXME the template_dir for Java codeGen is copied to poseidon2/lib, it look...	CodeGenerationUtil.java	IDE_Integration/src/com/gentleware
	TODO change the ResourceId for the Dummy Eclipse command	CommandEclipseDummy....	IDE_Integration/src/com/gentleware
	TODO what to do on Eclipse side in case of undo?	CommandEclipseJmiSetPr...	IDE_Integration/src/com/gentleware
!	FIXME setting the Eclipse java project for commands	CommandEclipseJmiSetPr...	IDE_Integration/src/com/gentleware

Figure 5-7.2 The Eclipse Workbench showing the Java Perspective[14]

The JDT is not only used for Java development, it also provides extension points to extend the JDT itself or to offer the functionalities of it for the plugin developers. The JDT makes use of many of the platform extension points and frameworks the Eclipse Platform provides.

The JDT is implemented by a group of plugins, with the user interface in a UI plugin and the non-UI infrastructure in a separate core plugin. This separation of UI and non-UI code

[14] Eclipse version 3.0

allows the JDT core to be used in configurations of the Eclipse Platform without GUI, and by other GUI tools that incorporate Java capabilities but do not need the JDT UI.

Figure 5-7.3 shows the main connections between JDT UI and the Eclipse platform. The JDT is a collection of plugins that extend the Eclipse workbench by providing perspectives, editors, views, wizards, action sets, property pages and preference pages for Java environment.

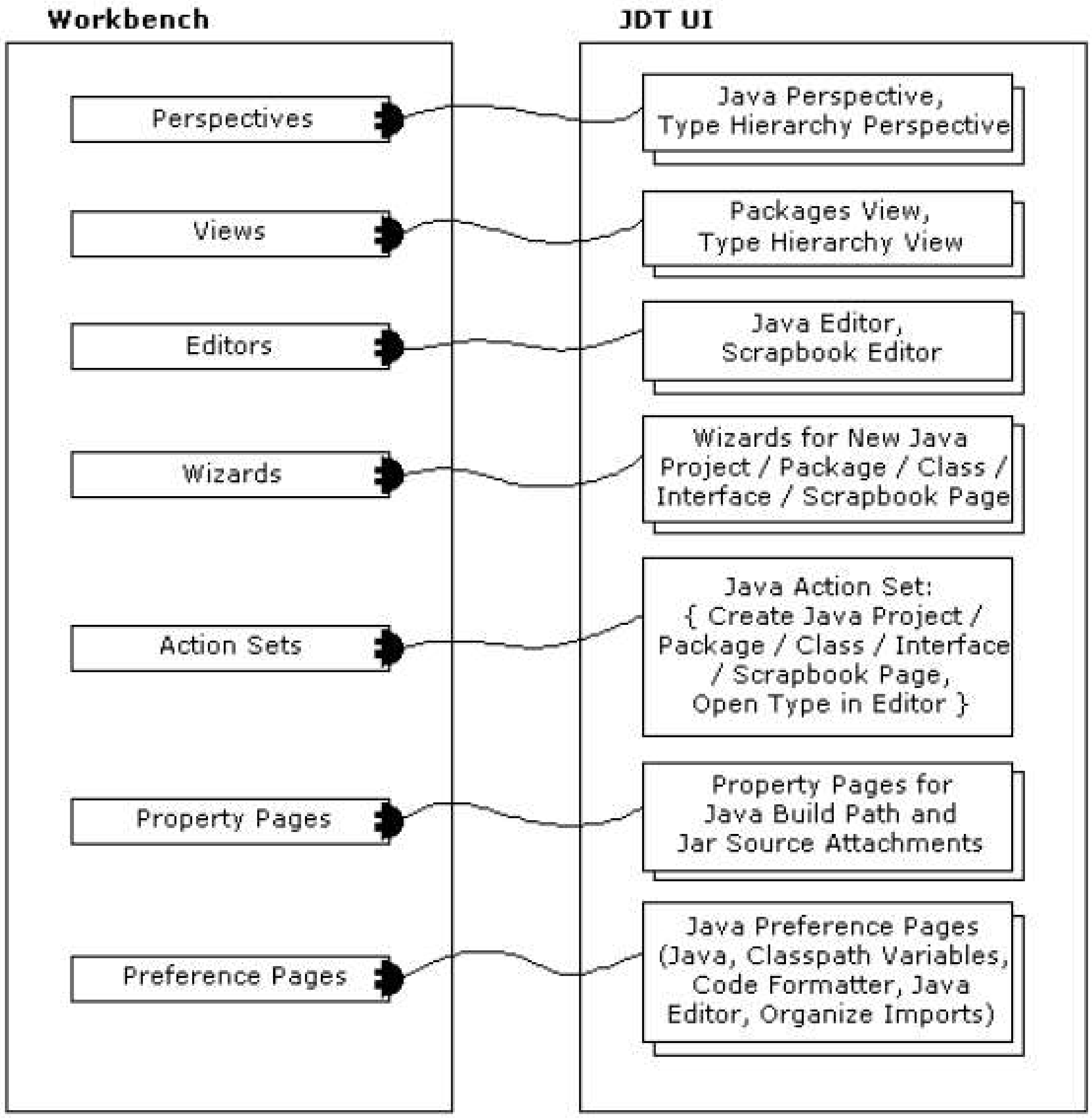

Workbench
JDT UI
Perspectives
Views
Editors
Wizards
Action Sets
Property Pages
Preference Pages
Java Perspective, Type Hierarchy Perspective
Packages View, Type Hierarchy View
Java Editor, Scrapbook Editor
Wizards for New Java Project / Package / Class / Interface / Scrapbook Page
Java Action Set: { Create Java Project / Package / Class / Interface / Scrapbook Page, Open Type in Editor }
Property Pages for Java Build Path and Jar Source Attachments
Java Preference Pages (Java, Classpath Variables, Code Formatter, Java Editor, Organize Imports)

Figure 5-7.3 Main Connections between JDT UI and the Eclipse Platform [31]

Figure 5-7.4 shows the main connections between the JDT Core and the Eclipse platform. The JDT core plugin extends the Eclipse Workbench by adding Java project nature, builder and problems for extension points, project natures, builders and marker types.

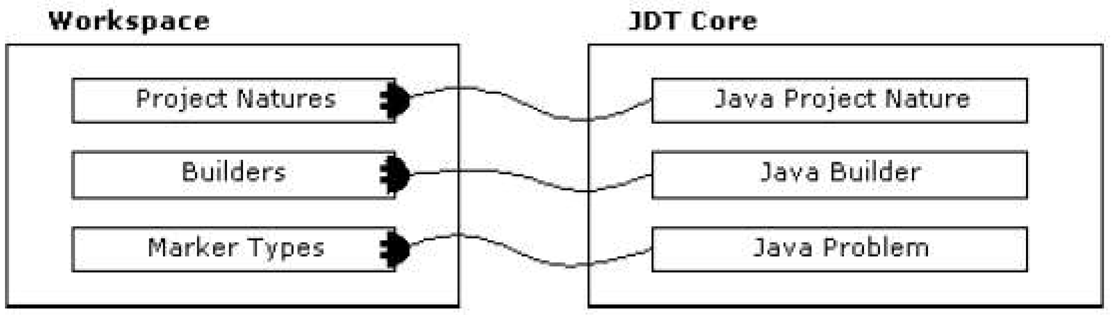


Figure 5-7.4 Main Connections between the JDT Core and the Eclipse Platform [31]

JDT allows plugin developers to develop plugins. A plugin that interacts with Java programs or resources needs to do some of the following:

- Programmatically manipulate Java resources, such as creating projects, generating Java source code, performing builds, or detecting problems in code.
- Programmatically launch a Java program from the platform
- Add new functions and extensions to the Java IDE itself
- …

The JDT API allows the plugin developers to manipulate the Java programs by the JDT Core, JDT UI and the JDT Debug components. JDT Core API comprises of the headless infrastructure for compiling and manipulating the Java source code where as JDT UI is comprised of the extensions which provide the Java IDE JDT. The JDT Debug component is for program launching and debugs support.

7.4.1 The Java Model

The Java model is a set of classes that model the objects associated with creating, editing, and building a Java Program.[32] These classes implement Java specific behavior for resources and they decompose these Java resources into Java elements.

Most plugins need the capabilities offered by the JDT core plugin. The JDT core plugin is composed of classes which represent the Java source and class files structure and which enables getting the information about the Java programs as the class names, methods, fields, import declarations etc. Without need for extra parsers for Java, the plugin developers can access and get information about the Java files and resources available in the workspace.

The *Java model* provides an API for navigating the *Java element tree*. The Java element tree represents the projects with Java nature with these following types of elements:

- Package fragment roots corresponding to a project's source folders and JAR libraries.
- Package fragments corresponding to specific packages within a package fragment root.
- Compilation units and binary classes corresponding to individual Java source (*.java) and binary class (*.class) files.
- Various types of Java declarations that appear within a compilation unit or class file:
- Package declarations.
- Import declarations.
- Class and interface declarations.
- Method and constructor declarations.
- Field declarations.

- Initializer declarations. [31]

The element types are summarized in the following Table 5-7.1.

Element	**Description**
`IJavaModel`	Represents the root Java element, corresponding to the workspace. The parent of all projects with the Java nature. It also gives the developer access to the projects without the java nature.
`IJavaProject`	Represents a Java project in the workspace (it is a child of `IJavaModel`)
`IPackageFragmentRoot`	Represents a set of package fragments, and maps the fragments to an underlying resource which is either a folder, JAR, or ZIP file (it is child of `IJavaProject`)
`IPackageFragment`	Represents the portion of the workspace that corresponds to an entire package, or a portion of the package (it is child of `IPackageFragmentRoot`)
`ICompilationUnit`	Represents a Java source file (child of `IPackageFragment`)
`IClassFile`	Represents a binary (compiled) type. (Child of `IPackageFragment`)
`IImportContainer`	Represents the collection of package import declarations in a compilation unit (child of `ICompilationUnit`)
`IImportDeclaration`	Represents a single package import declaration. (Child of `IImportContainer`)
`IType`	Represents either a source type inside a compilation unit, or a binary type inside a class file.
`IField`	Represents a field inside a type (child of `IType`)
`IMethod`	Represents a method or constructor inside a type (Child of `IType`)
`IInitializer`	Represents a static or instance initializer inside a type (Child of `IType`)
`IPackageDeclaration`	Represents a package declaration in a compilation unit. (Child of `ICompilationUnit`)

Table 5-7.1 Java Interfaces representing the Java Model in JDT [32]

The Java Model is based on a resource structure representation of Eclipse. The clients traversing the Java API need an API which differs from Eclipse's resource interfaces. This is a typical setup for the ***adapter pattern*** []. The adapter pattern says: *Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces*. In this example, `IJavaElement`[15] interface is considered. It plays the adapter role of the adapter pattern. The roles and the relationships can be seen in Figure 5-7.7[16].

[15] IJavaElement is the base class for all Java elements.

[16] IResource interface seen in the figure is from Eclipse resource API.

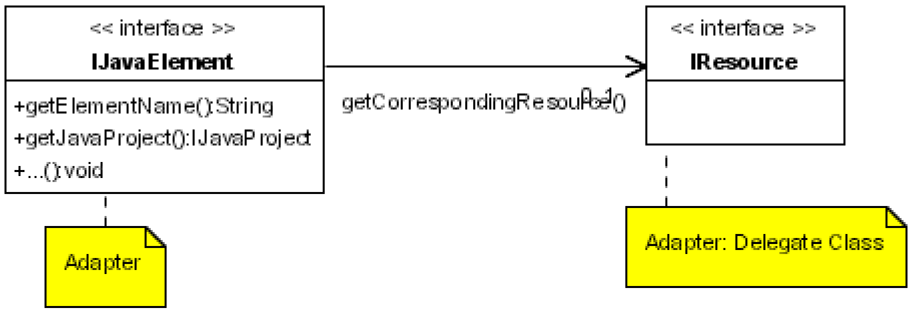


Figure 5-7.5 IJavaElement adapts IResource[ ]

The resource instance `IJavaElement` can be retrieved by `getCorrespondingResource()` method of `IJavaElement`. The `IJavaElement` enables the navigation from a Java element to its corresponding resource. It is also possible to get the Java element from a resource instance. The `JavaCore` class provides `create (IResource): IJavaElement` methods. Figure 5-7.6 shows the roles with respect to patterns and the relationships between the mentioned classes.

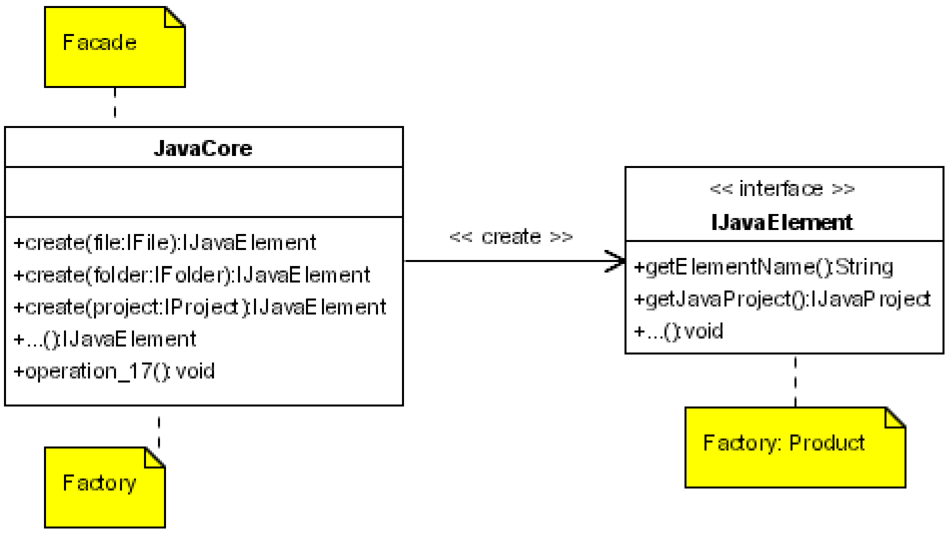


Figure 5-7.6 JavaCore is a facade and a factory []

The simplified UML class diagrams representing the Java model can be seen in Figure 5-7.7, Figure 5-7.8 and Figure 5-7.9.

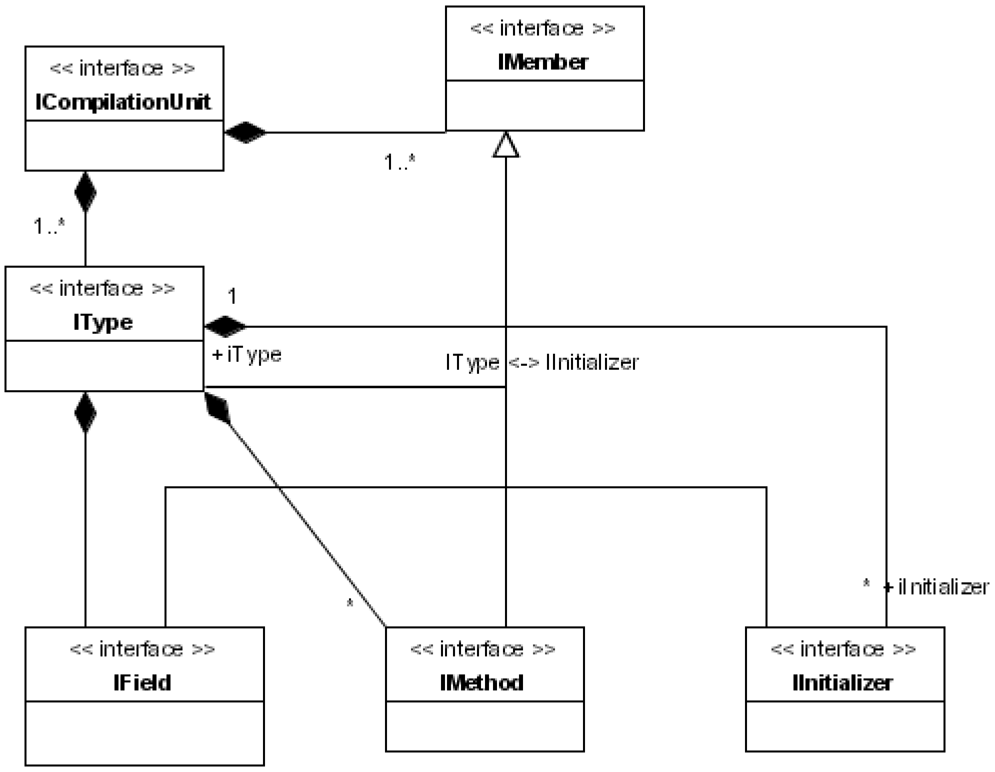


Figure 5-7.7 Java Type Representation

Java classes and interfaces are modeled with `IType`. The simplified class diagram is as Figure 5-7.7. The methods, fields, initializers and Java types (classes and interfaces) themselves are subtypes of `IMember`. The modeling of the types is compliant with the Java language specifications. An `ICompilationUnit` instance represents a Java source file (.java file) in a Java project. The `IClassFile` interface which is not shown in the figure for simplification reasons represents the class files.

Figure 5-7.8 shows the Import and Package declaration representation in the Java model of JDT. An `IImportContainer` is a child of `ICompilationUnit`. There can be at most one instance of `IImportContainer` for a specific `ICompilationUnit` instance (a Java source file).

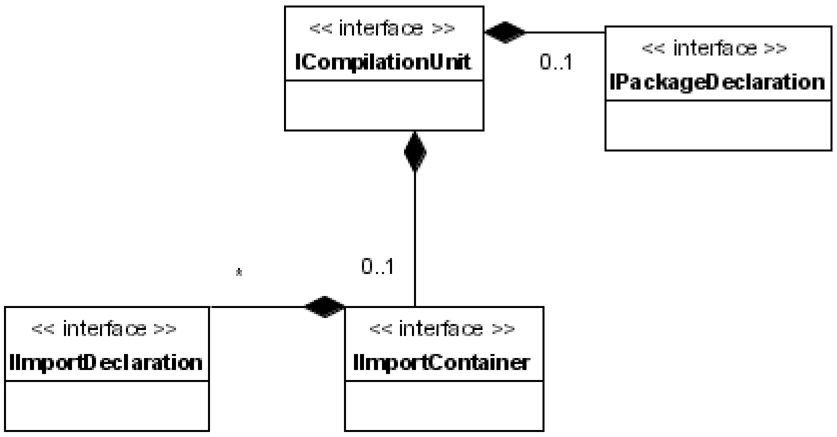


Figure 5-7.8 Import and Package Declaration Representation

IImportContainer can have any number of import declarations as it can be seen in Figure 5-7.8. An IImportDeclaration instance represents an import declaration in a Java class. A Java class can have at most one package declaration. The package declaration is modeled with the interface IPackageDeclaration.

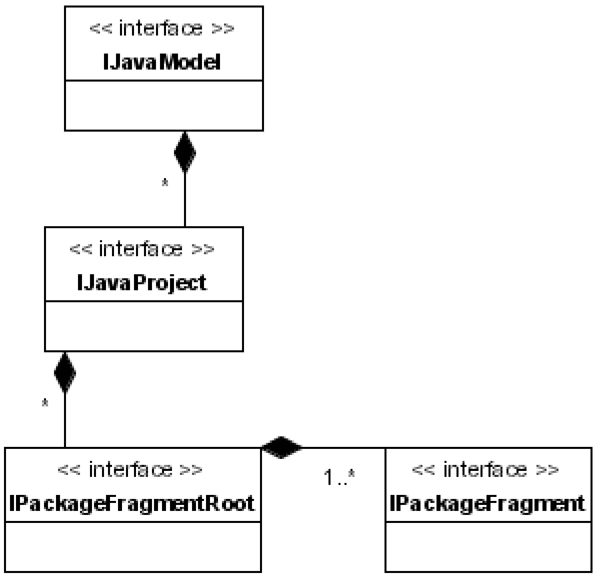


Figure 5-7.9 Java Package Structure Representation

And finally Figure 5-7.9 shows the class diagram for Java model representation of Java packages and subpackages. An IPackageFragmentRoot represents a source folder or a JAR file or a ZIP file. A package fragment root can have any number of IPackageFragment instances. An instance of IPackageFragment matches to an entire package or a portion of a package. A package fragment root belongs to the owning Java Project. A Java Project can have various package fragment roots. As all the Java projects and their resources are modeled with the root IJavaModel instance, the Java projects are children of IJavaModel.

The following figure, Figure 5-7.10, shows some of the Java elements in the Package Explorer view of JDT.

The Java Model of the JDT is more appropriate for traversing than manipulating. Getting information about classes, its members as methods and fields or types information are all available through the interfaces and classes in the org.eclipse.jdt.core package. But for manipulating the Java source files the Java Model is not appropriate, or even it is not enough. It does not provide the necessary API for manipulation. The API focuces on getting information about the Java model. One brute force way to manipulate the source files or parts of the files representing a method or field or type etc. is to directly change the file buffer or file contents. But it is cumbersome; it is not the best way for manipulation. Abstract Syntax Trees for Java files are the rescue for this problem.

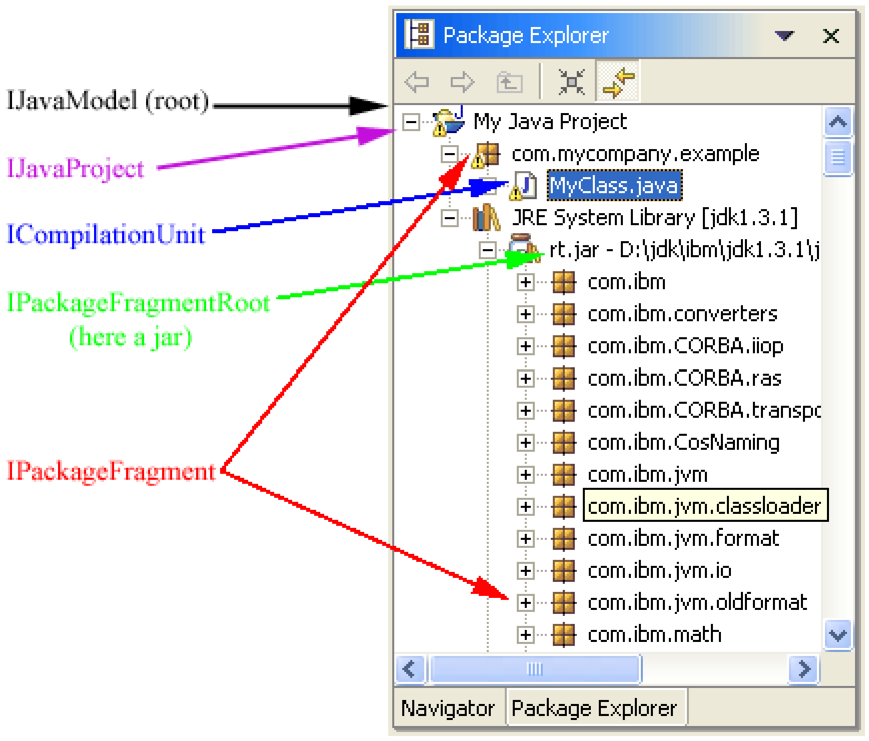


Figure 5-7.10 Java Elements in Packages View

For most Java-specific tools (including the Java UI) navigating and operating on a Java project via the Java model is more convenient than navigating the underlying resources. ***Java elements are represented by adaptable objects so that other parties can extend their behavior***. The Java element tree for a Java project is defined by its underlying resources and its classpath file. As keeping the complete Java element tree in the memory is neither efficient nor feasible, portions of the Java element tree are built on demand. For faster access to the element tree, an internal cache is kept.

Notification of Java Element Changes

Changes on Java elements are notified by `JavaCore`. The clients who want to get notified register the listener to the `JavaCore` Façade. The `ElementChangedEvent` contains the information about the changes on the elements. This event contains the instance of `IJavaElementDelta` which has the changes done on the Java elements in a tree form. It is a tree, because whenever a change in a source code is done, or classes are deleted, etc., not only one element gets changed. For example for a change in the method signature, the element method as well as the containing type change. The `IJavaElementDelta` changes start from the root element affected and go up to the leaf, most fine-grain change.

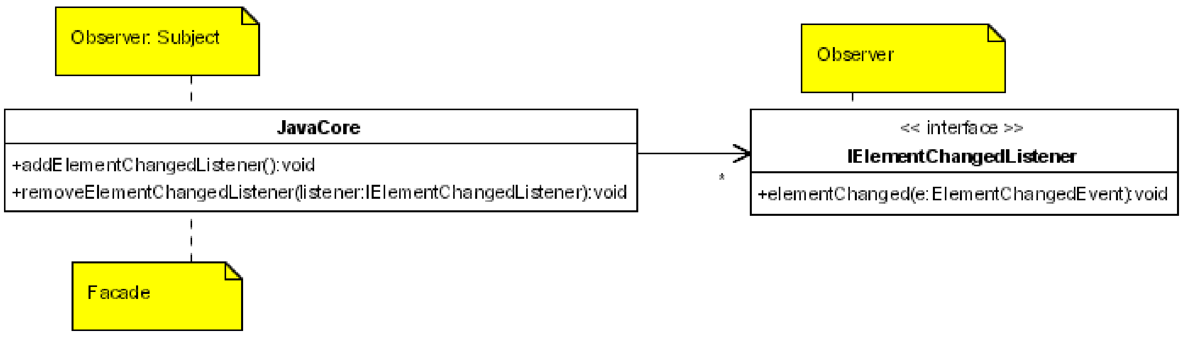

Observer: Subject
Observer
JavaCore
+addElementChangedListener():void
+removeElementChangedListener(listener:IElementChangedListener):void
*
<< interface >>
IElementChangedListener
+elementChanged(e:ElementChangedEvent):void
Facade

Figure 5-7.11 IElementChangedListener observes the Java Model []

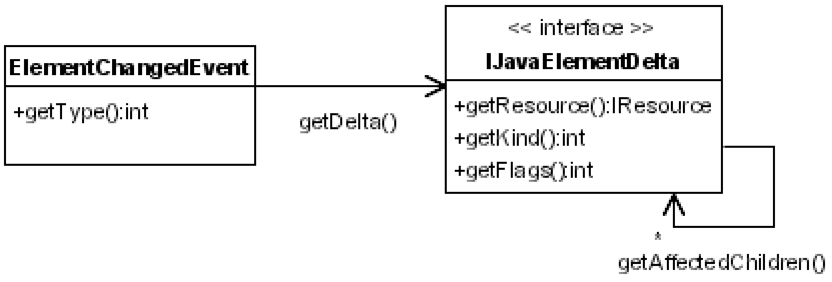


Figure 5-7.12 IJavaElementDelta builds as IResourceDelta a tree of changes []

The changes on Java elements are notified and saved in `IJavaElementDelta` as mentioned. The Java element changes found in the delta are only valid during the notification. The described relationships between the classes can be seen in figures Figure 5-7.11 and Figure 5-7.12.

The type hierarchies are not part of the change delta. For listening to the type hierarchies, Java model offers `ITypeHierarchyChangedListenern` listener, too.

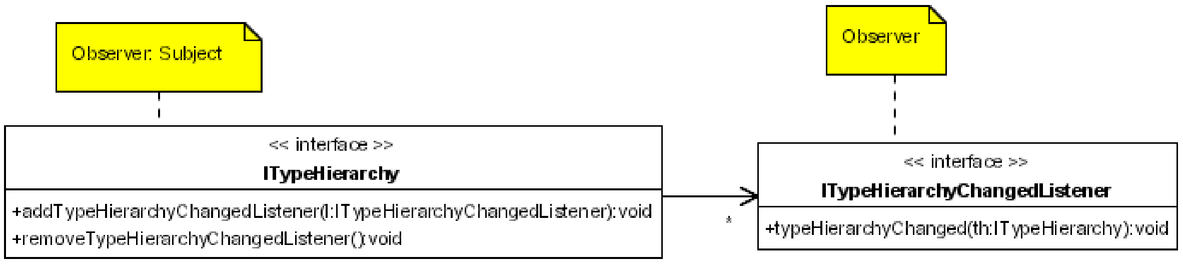

Observer: Subject
Observer
<< interface >>
ITypeHierarchy
+addTypeHierarchyChangedListener(l:ITypeHierarchyChangedListener):void
+removeTypeHierarchyChangedListener():void
*
<< interface >>
ITypeHierarchyChangedListener
+typeHierarchyChanged(th:ITypeHierarchy):void

Figure 5-7.13 ITypeHierarchyChangedListener observes ITypeHierarchy []

Besides Java Model specific listeners, the clients can also implement and register instance of `IResourceChangeListener`. This interface provides methods to analyze many modifications on Eclipse Workspace resources. Saving resources, deleting resources, compiling resources, etc., are some of the actions on resources which are provided by the Eclipse Platform.

The Java Model also provides `IBufferChangedListener` which contains the changes done on the editors in JDT. Any single change on the buffer is notified to the clients after registration. For analyzing the meaning of the changes, the client herself needs to design algorithms.

So, if we summarize the listener interfaces through which the clients get informed of the Java element changes, we see the following:

1. `IElementChangedListener`
2. `IBufferChangedListener`
3. `IResourceChangeListener`
4. `ITypeHierarchyChangedListener`

7.4.2 Abstract Syntax Tree: AST

The Java Model supports navigation through the Java element tree. But, the model is not detailed enough for fine-grained code analysis and modification. Java Core provides the Abstract Syntax Tree (AST) access of any compilation units. An AST represents the result of the parsing and analysis of the compilation unit.

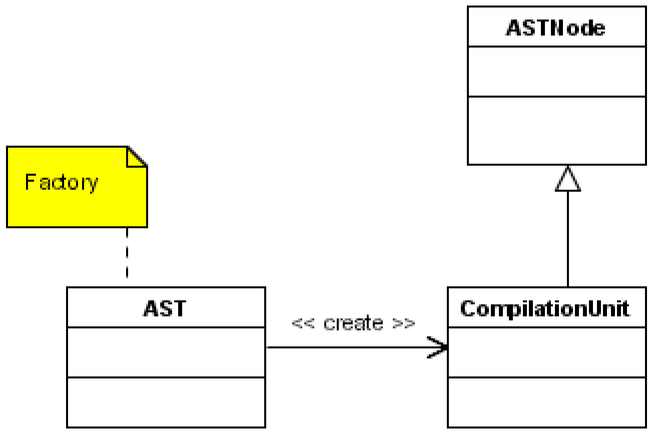
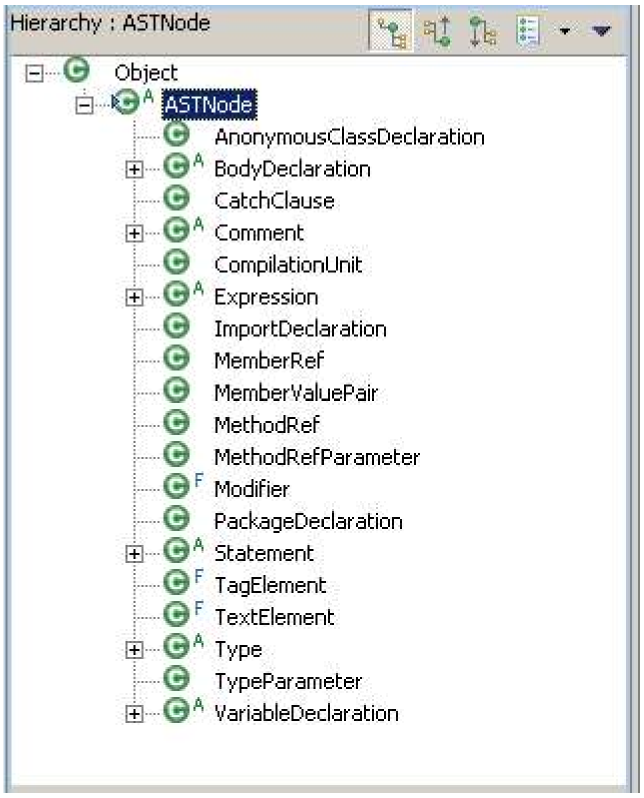


Figure 5-7.14 AST creates ASTNode Objects [33]

Every node of an AST represents an element of the program and it contains the source range of that element. The AST nodes are defined through a hierarchy where `ASTNode` is the base class. `AST` class creates all of the AST nodes. Figure 5-7.14 shows the node creation for AST. The node object `CompilationUnit` instance is created by `AST` and this class is a subclass of `ASTNode`.


Hierarchy : ASTNode

```
Object
  ASTNode
    AnonymousClassDeclaration
    BodyDeclaration
    CatchClause
    Comment
    CompilationUnit
    Expression
    ImportDeclaration
    MemberRef
    MemberValuePair
    MethodRef
    MethodRefParameter
    Modifier
    PackageDeclaration
    Statement
    TagElement
    TextElement
    Type
    TypeParameter
    VariableDeclaration
```

Figure 5-7.15 ASTNode subclasses

Figure 5-7.15 shows all subclasses of `ASTNode`.[17] An AST is created when detailed analysis of a compilation unit or modification on the compilation unit is needed. The AST analysis is the typical application of the *Visitor* pattern. If the visitor pattern is applied, it is important to know that the class hierarchy should be stable. Whenever the class hierarchy changes, the visitor instance used for traversing should be modified considering the changes in the hierarchy. In Figure 5-7.16 the class relationships between `ASTNode` and `ASTVisitor` can be seen. `ASTVisitor` provides all the methods for visiting any kind of AST nodes. This class is available through Java Model API.

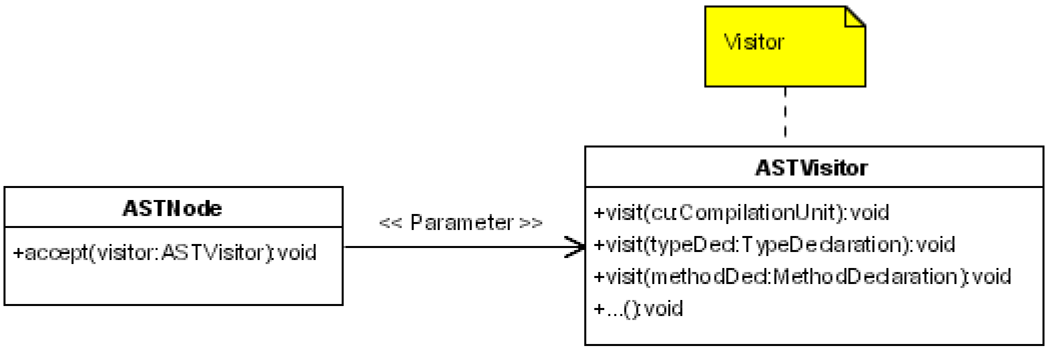

Visitor
ASTNode
+accept(visitor:ASTVisitor):void
<< Parameter >>
ASTVisitor
+visit(cu:CompilationUnit):void
+visit(typeDed:TypeDeclaration):void
+visit(methodDed:MethodDeclaration):void
+...():void

Figure 5-7.16 ASTVisitor visiting ASTNode Objects [33]

More details on Eclipse, JDT and Java model can be found in [31],[32] and [33].

[17] For Eclipse 3.0 JDT API

7.5 Swing/SWT Integration

The Eclipse Platform is a universal tool development and integration platform. It is completely written with SWT library. At the times Eclipse project started, IBM thought that Swing and AWT were too slow and cumbersome for graphical user interfaces. At those times, AWT was really slow and very buggy, having a large memory footprint.

As the Eclipse project become open source and individual developers and firms could write their own plugins and integrate to Eclipse, SWT became a main problem. Many Java programmers had experience with Swing and AWT. As time passed and new Java libraries were released, the Swing library got more and more professional. The reason IBM invented SWT became invalid. But as the Eclipse codebase is implemented with SWT, it is still a problem. The Java community has two UI libraries, where Swing is the mostly used one, but at the same time, one of the most successful open source project Eclipse does not use it.

It is a big problem for plugin development especially for the applications that need to be integrated to Eclipse but that are already implemented with Java Swing library. If integration to Eclipse is desired, the application GUI needs to be rewritten from scratch, a very time consuming and costly procedure.

Sun Microsystems and Eclipse worked together to integrate Swing applications into Eclipse. As the result of this work, the Eclipse 3.0 release makes it possible to integrate Swing applications into Eclipse Platform.

The details of the threading problems will be explained in System Design and Implementation chapter.

7.6 Summary

In this chapter Eclipse, its plugin architecture and the necessary tools and API are described that is used during the implementation for integrating Poseidon with Eclipse.

The aim of this thesis work is the analysis and review of the work done on code generation, reverse engineering, i.e. altogether roundtrip engineering as well as integrating the Poseidon tool to the Eclipse environment. Integration introduces the need for good understanding of Eclipse Platform where Java Core and the model are the main points of focus. The integration is especially demanded by Java developers who want to work with UML and Java together in the same environment. This is a feature bringing big productivity changes for the Java developers.

At the beginning of the chapter, the Eclipse platform was introduced. The aim is only to introduce the platform features that are necessary to understand the case study of roundtrip engineering between Poseidon and Eclipse Java Development Tooling.

After Eclipse Platform introduction, its plugin architecture is described. The Eclipse plugins are loaded lazily. The registry mechanism for the plugins was introduced.

The most important component of the Eclipse Platform in this thesis work is Java Core. The Java Core API offers many features for the plugin developers. The most important of all is the traversing and manipulating of the Java project tree with the resources as Java source files etc. The plugin developer does not need the hard and handy task of parsing, understanding, initializing, instantiating the Java programs. The JDT and better said Java core does this for the plugin developers. The adapters for the Java resources are available as Java Core API.
The listeners on Java sources and Java editor are very important at the process of roundtrip engineering. They are explained and analyzed in some detail. The problems will be described in the implementation chapter.

8 Code Generation

Object-oriented software development has been widely accepted as it contributes to a seamless transition to the development process. To describe the different aspects of a system, many specifications based on the models are obtained in the software design process. But still there is a gap between the high level modeling language and the programming languages. In this chapter Java code generation from UML models will be explained and previous studies will be reviewed. By generating code from UML models, the gap gets smaller. From the analyzed approaches in the current and previous studies the best matching approach will be chosen for use in integrating of Poseidon with IDEs.

In order to better design and manage complex systems, software teams turned to modeling languages which make the general handling of software projects easier. With help of the design understanding the problem area, finding the bugs are easier. The UML is one of the modeling languages used in software industry. But most important it is the de facto standard for modeling. Today most of the modeling of software systems is done with UML. Many companies have adopted UML as company's modeling environment and language.

The graphical design lets the team members grasp the details of the systems more easily. The communication between the team members is also improved by the usage of the many different diagram types of UML. For example to describe the static structure of the programs, class diagrams are used. The object-oriented view for the static structure of the programs can be seen and grasped very easily. Many presentations for software topics on object interaction in systems or general class structure now include UML diagrams.

Design is used for various reasons by developers, ranging from providing a description of the implementation of the system to modeling the architecture of the system independent of the implementation language or style choices as stated in [34]. In the process of designing there is always a choice of which level of abstraction of the system is to be modeled. The level of abstraction of the design represents the level of details represented in the modeling.

When the chosen level of abstraction for modeling is the implementation level, i.e. the level at which the source code is kept, it is not difficult to create the implementation code from the models compared to a model which has a higher level of design, with more abstraction. Many UML tools provide skeletal code from the UML model descriptions [35, 36, 37, 38, 39, 40,41]. In this kind of code generation, the designer is limited to using some of the UML constructs which match to the programming language chosen for implementation. For example, the UML provides multiple inheritance at designs but the programming language Java has no direct multiple inheritance. The implementation level UML model descriptions can not have multiple inheritance if the target language is Java. Otherwise the code generated is not a valid code in Java, i.e. it is not compliable.

Code generation is the task of creating the source code for a programming language from existing design and models. At the context of this thesis work, as mentioned, only generation of Java source code from UML models is considered. The UML models contain instances of many diagram types and the model information residing in these diagram instances. The source code can be generated directly from the user's UML models. Code generation can be done for models in class diagrams, sequence diagrams or state diagrams. [18]

Some of the metamodel elements of UML metamodel have directly matching Java constructs. The UML class instances can be assimilated to Java classes, as well as UML attributes with Java fields and UML operations with Java methods, thus making creation of code from these static UML elements straightforward. But this is not the case for UML constructs such as composition, aggregation or association.

[18] and in future may be other UML diagrams will be used for Code Generation

In code generation there has been wide research mainly resulting in the Model Driven Architecture (MDA) [42]. Problems have been found out after extensive research work. The analysis in the previous studies will be given in next subsections. Some basic problems remain: e.g. mapping of the UML aggregation and composition, multiplicities, and stereotypes.

Although there are well known problems in generation of code from implementation level UML model descriptions (problems are analyzed in section 8.3), the main focus in this work is code generation from implementation level designs.

8.1 Narrowing the Design-Implementation Gap

Some concepts in UML and Object-Oriented Programming Languages (OOPL) have direct or easy-enough mappings. Table 6-8.1 shows a mapping from an Object-oriented programming language to UML concepts.

OO PL Prope rty	*UML Concept (s)*
Assignments	Actions in State Diagram Action states in Activity Diagram
Control Flow	Branch in Activity Diagram Guard Condition in Interaction Diagram Guard Condition in State Diagram
Primitive Data Types	User Defined Stereotypes in Class Diagram
Functions and Variables	Action states in Activity Diagram Declare static method in utility class Declare functions in the activity states in Activity diagram
Comments	Notes in all Diagram
Operators	Not mentioned in UML but can be used in the action state in activity diagram
Classes Declarations	Declare as class in class diagram
Data Encapsulations	Declare as class in class diagram
Access Control	Visibility in class diagram
Inheritances	Inheritances relationships in class diagram
Polymorphisms	Supported since inheritance can be represented in UML
Assertions	Constraint in OCL
Messages	Messages in Interaction Diagrams
Interfaces	Interface in class diagram
Class Templates	Class templates in class diagram
Control name Space Conflict	Packages in component/class diagram

Table 6-8.1 Mapping from Object-oriented Programming Language to UML []

The table tries to show the points where the gap between the OOPLs and UML is seamless or narrow. The concepts in this table can be used at code generation from UML models.

There are three main approaches to code generation as stated in [43]. The *structural approach* is based on using the models of the object structure. Most of the UML tools generate skeletal code from class diagrams. This is an example for this approach. This approach has drawbacks as no behavior expression exists in the created code. The generated code with this approach is not complete.

The second approach is the *state machine approach*. This approach uses the state-machine of objects plus the object structures. The object structure and the state-machine enable creation of complete code which includes the object behavior. The main disadvantage for this approach is that the developers must express all behaviors by state machines which are not practical and sometimes even not meaningful.

The third and the last approach is the *translative approach* which is based on application and architecture which are based on the Shlaer and Mellor Method [44]. In this approach a complete model of the application and architecture is developed. After the development of the model, based on the rules found in the model, code is created. The disadvantage of this approach is that it is not the same as the Unified Software Development Process. [45]

The first approach will be analyzed for Java code generation. The other approaches are outside the context of this thesis. The main mappings used for almost all code generation algorithms and approaches for "*UML to Java*" are as seen in Table 6-8.2.

UML	***Java***
Package of classes/component	Package
Dependency between components and packages in Component Diagram	Import
Class Component	Class or defined component
Class	Class
Interface	Interface
Realization between classes and interfaces	Implements
Generalization between classes or interfaces	Extends
Association between classes	Reference Attributes on both classes[19]
Attribute	Attribute
Operation	Method
Properties on classes (Abstract, Final)	Class Modifiers (Abstract, Final)
Properties on attributes	Attribute modifiers
Visibility	Visibility
Implementation access	Package level visibility

Table 6-8.2 UML to Java Translation [43]

The main steps at code generation for class diagrams are (the algorithm from [43]):

- Packages will be transformed into Java packages. The classes and interfaces belonging to them will be also generated.
- Classes and interfaces in the diagram are transformed into Java code. The operations and attributes of them will be also generated in the Java code.
- For each class the invariants will also be generated as code.
- The relationships between the classes will be also generated and represented in the Java code.
- If pre/post conditions for operations exist, they will be also translated to source code.

[19] Mapping associations is complex and it will be described in the following sections.

For invariants and relationships the code generation needs a deeper analysis which is to be found in the next sections. The way code is generated depends on the approach taken for generating code from them.

8.2 Generation of Code from High Level Designs

The designers of a system do not have any specific technology or programming language in mind when analyzing the problem area and the system. The system is expressed independent of them. Such a high level model which does not consider the technologies or specific programming languages is more appropriate for showing system architecture and the relationships between its components. A high level model is clearer because the system architecture is independent of the implementation level details. This model can be mapped to many different implementation level models respectively programming languages or technologies like Enterprise Java Beans (EJB) or Microsoft's .NET.

Most commercial modeling tools make publicity for design model at the same level of abstraction as implementation. This approach is a way of viewing the source code in a graphical notation with the help of UML. Especially if the roundtrip for the source code is also used, the UML model contains repeated instances of design patterns that have been used in the implementation.

Although the system architecture is clear and flexible for high level designs, there are also problems specific to these models. These models are no more directly applicable to programming languages and it is difficult to keep the code and the model in sync. Generally the model gets out-of-date.

8.2.1 Approaches

This section gives main approaches on code generation from high level models that are found in reviewed papers.

8.2.1.1 Harrison Approach in [34]

The paper [34] analyses the benefits and problems met in code generation from high level, implementation-independent models. They present *a mapping method that preserves expressive freedom for the designer by allowing the specification of abstract models that make few assumptions about the underlying implementation*. A high level implementation is generated by the mapping they suggest. This high level implementation shields the implementers from worrying about the low level details. One example is *cursors* which abstracts the associations and eases the management and navigation of the associations. SAGE (Scalable Adapter Generator) is the tool that they implemented as a prototype for their suggestions. The tool SAGE is used at IBM Watson Research Center.

Objectives in Mapping High Level UML Designs

In a system being designed there can be several unfortunate consequences [46] if the system is designed without considering concerns [47], e.g. the design becoming clumsy shortly after as a result of the many design elements scattered and tangled together. The better way of designing is to separate the concerns in a system and making them explicit at the design level. For example security, graphical user interface, networking of a system are highly independent from each other.

The proposed design process in [34] involves different roles. *System designer, concern designer, object designer, behavior implementer* and *representation implementer* are the roles. The objectives to be held during high-level designing are:

- Separate design from implementation
- Separate behavior from representation
- Maximize type-safety
- Avoid Roundtripping
- Support for Designing with concerns
- Support for Generalizations (supporting multiple inheritance)
- Support for Associations
- Behavioral access for Associations and Attributes
- Unified set of idioms for Accessing/Navigating the model
- Promote code reuse

Mapping High-level Design and Implementation

After the objectives are also stated, the next step consists in defining the mapping of the UML high level designs to Java code.

The UML constructs from which code is generated are marked with stereotype **"entity"** with their associations, relationships, attributes, operations, generalization relationships and association relationships. The classes without the stereotype **"entity"** are thought to be implementation specific and will not be generated with the framework suggested in this paper. An example class hierarchy which is generated from a UML class diagram can be seen in Figure 6-8.1.

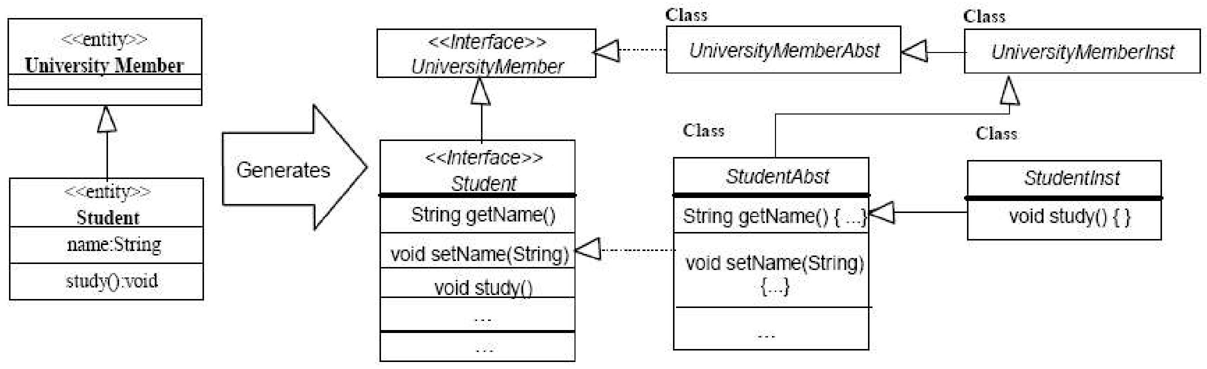


Figure 6-8.1 Class hierarchy generated from a UML class diagram [34]

Each entity is mapped into an interface, and two implementing classes: An abstract class and an instantiable class. The code generated for the interface and the abstract class is not to be changed by the implementers. The specific implementation for operations can be done by the implementers in the instantiable class. In Figure 6-8.2 an example class hierarchy is shown for generation of UML constructs having stereotype **"entity"**. The details about the mapping can be found in [34].

Associations' code generation also has an abstraction, *cursors*. The semantics of an association and its representation in code is separated. The cursor encapsulates the complexity of associations. The following figure shows the high level UML model and the generated classes' implementation level UML model where high level model contains associations.

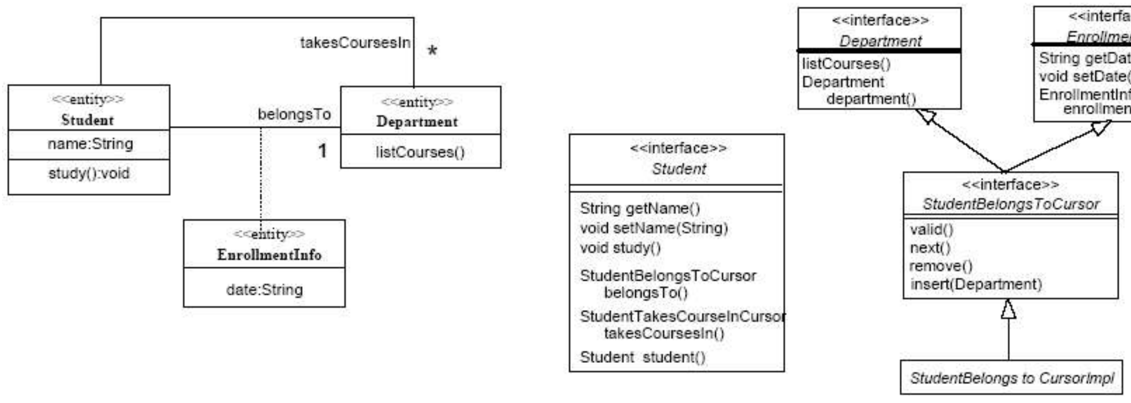


Figure 6-8.2 Example of generated code for associations [34]

For the belongsTo relationship, methods in the Student interface are generated as seen. The implementation of the generated methods will be in StudentAbst class which is not shown in this figure.

The takeCoursesIn association has a * multiplicity. Respective code for this association is also generated similarly. An extra method is added to the Student interface for getting the list of departments where the student takes courses. The methods generated return instances of type StudentBelongsToCursor for belongsTo association and respectively for takesCoursesIn association. This class is generated specifically for representing and implementing the associations.

For more details on code generation approach proposed in [34], please refer to [34].

8.2.1.2 A Language to Describe Software Textures

The paper [48] uses a similar approach for code generation from high level design models. Their motivation conforms to [34] where the model is independent of the implementation language. In this way more than one programming language can be used for the same abstract high level model. The difference of the level of abstraction between the UML model and the implementation code corresponds to a productivity increase.

The abstract UML models can be used by further teams for further projects. Generation of code in an appropriate way can make the re-use of the same abstract models possible.

The proposed modeling and code generation has several steps. The UML modeling is divided in 3 levels:

- *Design model*
- *Architecture model*
- *Implementation model*

In the *design model* stereotypes are used for indication of the roles of individual UML classes as in Figure 6-8.3.

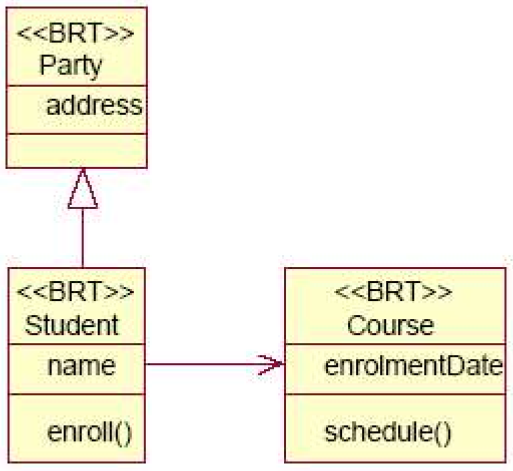


Figure 6-8.3 Design Model [48]

In the *architecture model*, multi-level stereotyping is used. For example to represent all classes of stereotype <<Foo>> in the design model, <<Stereotype>> stereotype is used in the architecture model. In the UML each model element can have one stereotype, however here multi-level stereotyping is necessary. Tagged values could be used for simulating the multi-level stereotyping but they are already used for reasons like fine-tuning of code generation. For this the solution they propose to use a <<Allowable>> stereotype with operations, links, and attributes to denote items in the architecture and implementation models that are allowable constructs. The items that do not have this stereotype are representing mandatory constructs.

Similar techniques are used for implementation model as in Figure 6-8.4 to the effect that all UML diagrams have a one-to-one corresponding UML diagram at architecture level. The mapping of implementation model diagrams and architecture model diagrams are achieved by usage of a simple strict set of mapping rules for model element names.

The modeling and metamodeling techniques suggested in [48] lead to highly compact and precise implementation patterns. The extension mechanisms of standard UML *by stereotyping and tagged values* are insufficient for this technique. As a result they invented their own mechanism.

The actual mapping of the architecture model to implementation model uses the ways similar to the patterns described in [34]. As in [34], most classes from architecture model correspond to a Java interface, an abstract Java class and a concrete Java class and a Java collection class for implementing associations.

For mapping of the software textures in the architecture model to textures in the implementation model the UML has a *traces* relationship but this is insufficient in this context. It can be only used for UML classes. To solve this problem, they invented a new diagram notation *texture diagram notation* specially designed for the mapping purposes. The notational elements of this new notation can be seen in Figure 6-8.5.

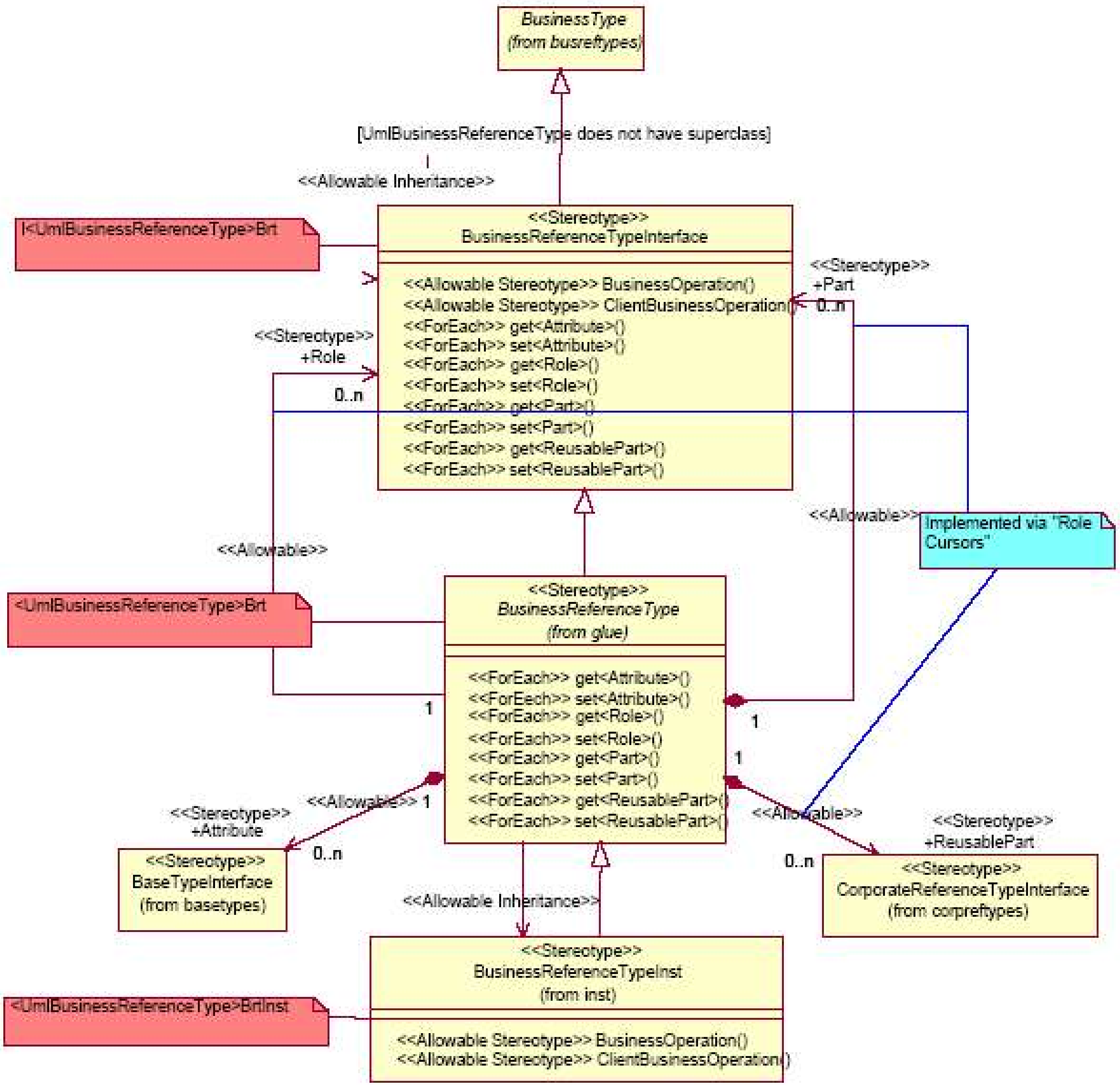

BusinessType
(from busreftypes)
[UmlBusinessReferenceType does not have superclass]
I
<<Allowable Inheritance>>
I<UmlBusinessReferenceType>Brt
<<Stereotype>>
BusinessReferenceTypeInterface
<<Allowable Stereotype>> BusinessOperation()
<<Allowable Stereotype>> ClientBusinessOperation()
<<ForEach>> get<Attribute>()
<<ForEach>> set<Attribute>()
<<ForEach>> get<Role>()
<<ForEach>> set<Role>()
<<ForEach>> get<Part>()
<<ForEach>> set<Part>()
<<ForEach>> get<ReusablePart>()
<<ForEach>> set<ReusablePart>()
<<Stereotype>>
+Part
0..n
<<Stereotype>>
+Role
0..n
<<Allowable>>
<<Allowable>>
Implemented via "Role Cursors"
<UmlBusinessReferenceType>Brt
<<Stereotype>>
BusinessReferenceType
(from glue)
<<ForEach>> get<Attribute>()
<<ForEech>> set<Attribute>()
<<ForEach>> get<Role>()
<<ForEach>> set<Role>()
<<ForEach>> get<Part>()
<<ForEach>> set<Part>()
<<ForEach>> get<ReusablePart>()
<<ForEach>> set<ReusablePart>()
1
1
1
<<Stereotype>>
+Attribute
<<Allowable>> 1
0..n
<<Allowable>>
<<Stereotype>>
+ReusablePart
0..n
<<Stereotype>>
BaseTypeInterface
(from basetypes)
<<Allowable Inheritance>>
<<Stereotype>>
CorporateReferenceTypeInterface
(from corpreftypes)
<UmlBusinessReferenceType>BrtInst
<<Stereotype>>
BusinessReferenceTypeInst
(from inst)
<<Allowable Stereotype>> BusinessOperation()
<<Allowable Stereotype>> ClientBusinessOperation()

Figure 6-8.4 Implementation Model [48]

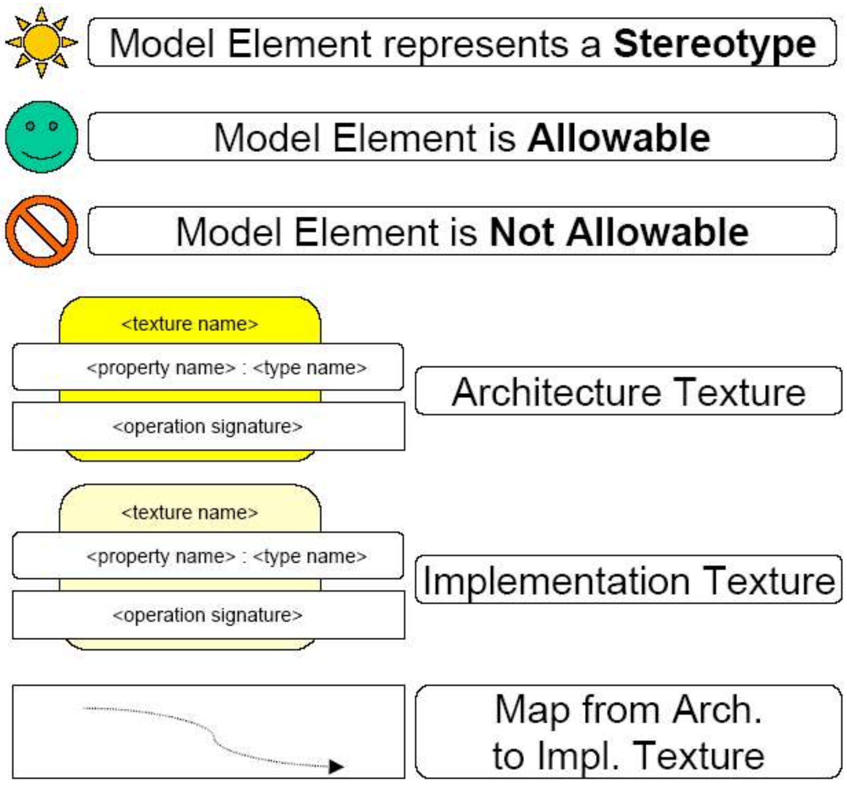

Model Element represents a Stereotype
Model Element is Allowable
Model Element is Not Allowable
<texture name>
<property name> : <type name>
<operation signature>
Architecture Texture
<texture name>
<property name> : <type name>
<operation signature>
Implementation Texture
Map from Arch. to Impl. Texture

Figure 6-8.5 Notational Elements Chosen [48]

For additional details and information, please refer to [48].

8.2.1.3 Aspect code generation from UML models

In [49] the research carried out concentrates on generating Aspect code from high level UML models. This paper presents a concept for aspect-oriented (AO) design and a seamless integration of AO design and implementation. They suggest a design notation for AO designs based on standard UML. The notation suggested *separates clearly the reusable programming language independent design of Aspect code and base (business logic) code from the language dependent cross-cutting parts*. They suggest a mapping from aspect design model to implementation language.

In [49] they extend the UML by standard extension mechanisms like *stereotypes, tagged values and constraints* without changing the UML metamodel which allows using any UML design tools. They claim that their work can be seen as a step towards the UML standardization on defining aspects at design level of Aspect Oriented Software Development (AOSD). They offer a terminology that keeps aspect and base elements apart which enables reuse of the aspect and base elements in any other model.

This paper analyzes the problem of Aspect code introduced to the software projects at very late steps, i.e. at implementation. The Aspect Oriented Programming (AOP) relies on aspects which are scattered through all code unlike the common behavior analyzed in classes in object-oriented programming. The AOP focuses on the behavior scattered through the classes, like security. To solve the problem that AOP is only introduced at the implementation phase of a software engineering process, a new approach is suggested. The design phase the modeling is done considering aspects in a system and they are modeled with base elements. The base elements and the aspects are modeled separately by holding to considerations of

Separation of Concerns (SOC). By this way, the models for aspects and base elements can be reused.

They address the specification of crosscutting concerns at design level to maintain the separation of concerns earlier in the lifecycle. A terminology based on the core concepts of **AspectJ** [35][36] and **UFA**[37][50] is adopted.

The work addresses the aspect-oriented development process from design to code. Aspect code generation of validated aspect design models is provided.

As mentioned before, the standard extension mechanisms of UML is used to design aspect models. The generated code from the aspect models is AspectJ code. AspectJ is an aspect-oriented extension to the Java language, which is one of the most common aspect-oriented languages.

Modeling Aspects in UML

The notation adopted or chosen should consider the concerns that aspects are scattered through all classes in the system, e.g. a security aspect. A *concern* itself can consist of several classes. Therefore, to model concerns, a module construct is chosen.

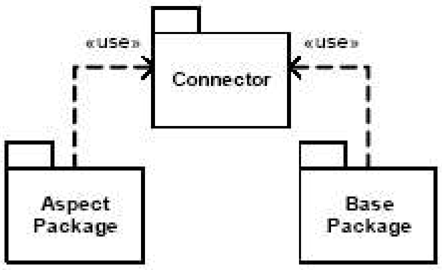


Figure 6-8.6 Package Level Decomposition [49]

The underlying concepts of UFA are adopted and some additional ones are added for AspectJ concepts. The following figure, Figure 6-8.6, shows an example model with the chosen constructs and concepts.

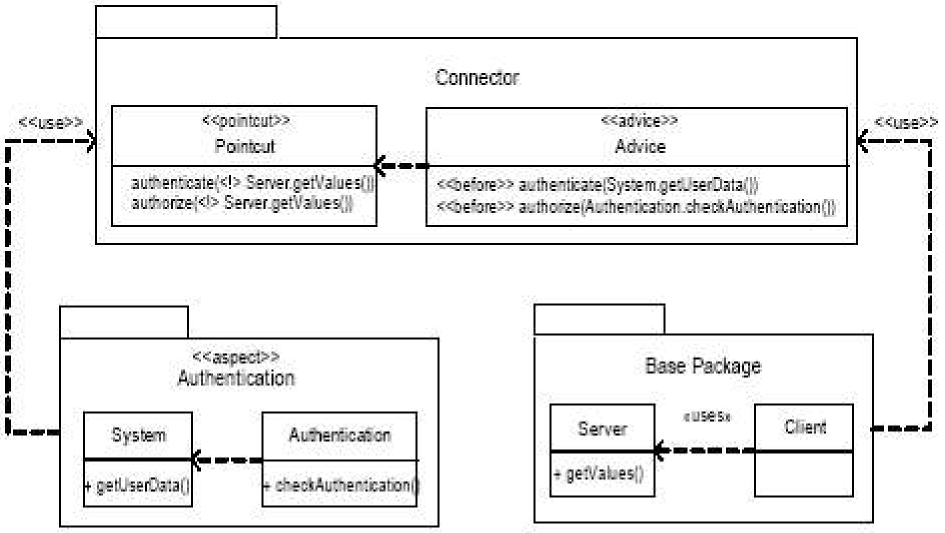


Figure 6-8.7 Security Design Example [49]

The notation includes a base package (containing the business logic), an aspect package (containing the crosscutting concerns) and a connector which links the aspect and the base elements. This separation allows reusability, and also the connectors encapsulate the underlying technologies resulting in easy replaceability of the connector code. [20]

The connector link contains the main concepts from the AOP like specifying of execution points in a program, the actions to be performed at those points as well as type-modification constructs. The connector defined for AspectJ includes concepts for AspectJ itself. For other AOP technologies different connector types should be used. For example the connector for AspectJ includes pointcuts, advices and introductions corresponding to AspectJ core concepts. All these elements are defined at methods of the classes that they belong to.

After fine-tuning of the Aspect model, the next step is code generation. The final design for which aspect code is to be generated is as seen in Figure 6-8.7.

Further details can be found in [49].

8.2.1.4 Summary of the analyzed approaches

The Harrison approach starts by describing the problems met during mapping a high level UML model to source code. The UML models are implementation-language independent. They define roles for the software development phase such as system designer, object designer, behaviour implementer and etc. For the mapping of the high-level designs, main objectives are summarized in this paper. Some of the objectives are separating the design from the implementation, separating the behaviour from the representation, avoiding roundtripping (most often, when high-level models are mapped to code, roundtripping is avoided, for example MDA also advises avoiding roundtripping), support for associations, etc.

In this approach, example mapping is given for a high-level UML model. The generated source code is Java source code. UML classes are often stereotyped which differentiates the elements to be generated or which describes what generation template to use. An example for this is the stereotype "**entity**". The UML classes with this stereotype are generated as one interface and two implementing classes. The implementing classes are an abstract class and an instantiable class. The person with role *implementer* can implement the specific implementations for methods in the instantiable class. Rest of the source code is not assumed to be changed.

The associations are also generated with a specific code template. For any association in the high-level model, an abstraction *cursor* is used. The cursors encapsulate the complexity of the associations. For the multiplicities bigger than 1, a list is generated with the respective list manipulation methods such as `remove`, `next`, `add` and `insert`.

The second approach analyzed has very similar concepts to the Harrison's approach. They reference the work of the Harrison in their paper. In this paper, the models from which code will be generated are design models, i.e. high-level UML models. For generation of code, very similar templates and algorithms are used. The UML models in this approach can be in one of these 3 levels: *Design* model, *architecture* model and the *implementation* model. In these levels, stereotyping is very often used. A way of multi-level stereotyping is used which is not allowed in the standard UML. For representation of models, they use their own extension mechanism of the UML. The standard extension mechanism stereotyping and tagged values to extend the UML are not enough for their purposes.

The generated code in this approach is very similar to the code generated by the Harrison's approach. The diagram elements for their models contain special notational elements they designed for representing the models in the mentioned levels of models. The notational elements are a notation representing model elements with a Stereotype, model

[20] From page 3 of [49]

elements which are allowable, model elements which are not allowable, architecture texture and implementation texture and mapping from architecture to implementation texture.

The last approach analyzed for code generation of high-level model uses Aspect UML models. The generated code and the UML model show aspects scattered through the system. Aspects are unlike the object-oriented classes can not be modeled with single classes. The behaviour can not be represented easily. The code representing aspects are generally in all of the system classes. For example an aspect that is scattered through systems is security issue. In this paper describing this approach, Aspect code is generated from high-level UML models. The paper presents a concept for aspect-oriented design and a seamless integration of aspect-oriented design to the implementation. They suggest a new notation for modeling UML models with Aspect design. The notation they suggest separates the part in a model which are reusable and which are not. The reusable programming language independent design of Aspect code and the language dependent base code is separated. Again in this approach they extend the UML. The extension is through standard way, i.e *stereotypes*.

The approaches presented in this work are not the only possible ways for code generation from high-level UML models. But the approaches presented give a basic summary of the possible ways or algorithms. For example, stereotypes are very often used for deciding on a code generation template. Often, UML is extended either through standard mechanisms or mechanisms that are not standard. The mechanism is dependent of the algorithm in this case.

One another common aspect in code generation from high-level models is avoidance of roundtripping. As the model elements in the UML models might have different mapping ways to source code, the roundtripping is not easy. It is very error-prone and needs high analysis.

8.2.2 Comments and Problems analyzed on Code Generation from High Level Designs

Generating code from a model and executing the code following with the running of the system is a dream shared by many. Whether it is realizable is a main question. Up to now, the approaches reviewed only create a portion of code for specific needs. The generated code in most cases still needs to be customized and handled by the programmers and the developers. Still one can not argue that an automated process of code generation is not useful.

Code generation is not widely adopted by the software industry for various reasons: missing standards, inconsistencies between the modeling languages and programming languages and the inadequacy of the UML metamodel of current version 1.4. The new version 2.0 is to be released, whose benefits and uses will be appreciated with time. The major uses of code generation in software systems is in Object-relational mapping (O-R), Object-XML mapping where mainly the behavior of the program is not seen, but the representation of the domain model. In O-R mapping for example, the same domain data has different representation and views: as objects or as database tables. The code to be generated is mainly syntax change code.

In code generation of UML models, there are more specific problems corresponding to mainly inconsistency between UML as a modeling language and Java as a programming language. UML constructs like association, dependency or multiple generalization does not have a direct mapping in Java. But those problems are mainly the problems encountered at code generation of low-level UML models.

Problems

- Composition and Multiple Inheritance

- Arbitrary UML high levels models can not be generated
- Not suitable for synchronization-roundtrip engineering
- Almost always one-way code generation

A main problem is the problem of multiple inheritance at both levels of design: High level and the low level UML designs. The multiple inheritance existing in UML is not directly mappable to Java. In Java a class can not extend more than one class. But a UML class can have any number of superclasses. A UML class can be a *generalization* of many super classes.

The multiple inheritance problem is analyzed in [51].The multiple inheritance in UML models can be in a way simulated in Java by *single inheritance*. Even this is not without new problems introduced. The way of expressing multiple inheritance in Java will be explained soon.

Multiple inheritance is one of the difficult issues in high level designs in the UML. The question is how to deal with them. Inheritance is a technique for specifying the reuse of implementations. There are two entirely different reasons for use of inheritance: *differential development and composition*. Differential development occurs when we want to say "x is like superX except that features as ...". In the programming languages the concept refers to *single inheritance*. On the other hand, composition is the concept behind "this one is both a one of these and a one of those". The programming language concept multiple inheritance corresponds to composition in high level design.

Multiple inheritance in programming languages causes various problems when programming. C++ and Smalltalk-80 have multiple inheritance. The Java creators omitted the multiple inheritance from the language to get rid of the complexity it introduces to the language. For this reason there was a need for extralinguistic support for composition. Hyper/J [52] and AspectJ are composition tools for Java.

Generalization and Multiple Inheritance in High Level Design

In UML the multiple inheritance is used for generalization whereas in the programming languages multiple inheritance is mainly used for code-reuse.

The tool Tengger [53] generates stylized Java code by using merge-by-name composition of collections of high-level UML designs.[21] The paper's authors have analysed the mapping choices [34] of Tengger for UML's multiple inheritance onto Java's single inheritance and they identified many problems. For use in larger-scale composition based projects Tengger was unnecessary, and it could be replaced by Hyper/J's composition separating the design and implementation more clearly.

- Arborization of mapping multiple inheritance in UML to single inheritance in Java is faulty:

The problem will be shown with a small example where the multiple inheritance in the high-level design will be mapped to single inheritance in the low-level design.

[21] from page 2 of [51]

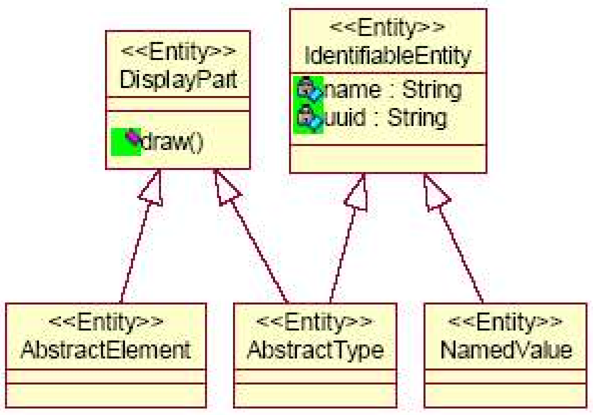


Figure 6-8.8 Multiple Generalization[51]

The entity AbstractType generalizes both DisplayPart and IdentifiableEntity. This can not be directly mapped to Java with single inheritance. The class diagram in Figure 6-8.9 represents the high-level design model with multiple-inheritance mapped to high-level design with single-inheritance.

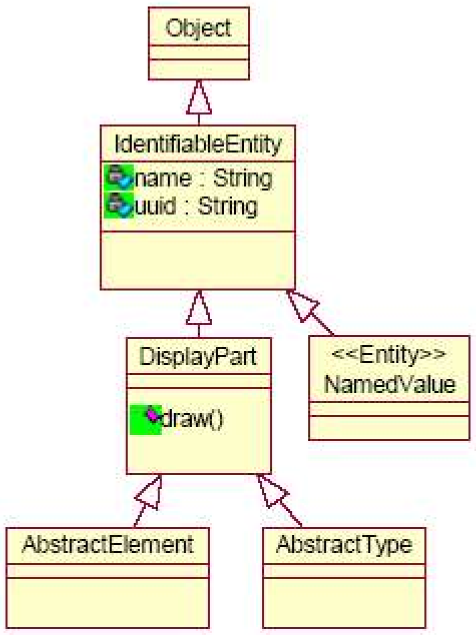


Figure 6-8.9 Single-Inheritance Arborization [51]

But even this mapping has various problems. Here, there are some potential false inheritances. For example "equals" method from IdentifiableEntity would be inherited to AbstractElement although this is not the case in the multiple-inheritance high level class diagram. The solution for this false inheritance is *method renaming*. If the UML-Java mapper knows that IdentifiableEntity implements "`equals`", then it could change the calls "`equals`" in AbstractElement to more directly targeted calls to solve the name clashing. But which class implements which method can be just retrieved from the implementation.

- Design/Implementation Tension

In fact, working with the implementations, Hyper/J performs the much more sophisticated method renaming and superclass redirection needed to fix the false inheritances. But recognizing that the implementation notes did not belong in the design and that they could not appear in the naïve Java code led us to observe that there is a somewhat deeper problem here. In the design, multiple generalization is quite appropriate and meaningful and, in mappings like those used by Tengger that create an interface corresponding to each entity, they imply a multiple-inclusion structure among the interfaces. Most languages with interfaces allow multiple-inclusion for interfaces, making for a natural design/implementation map. But a mapping that uses the design to determine single-inheritance relationships among implementation classes stands on very shaky ground. Inheritance is an implementation reuse mechanism, not a high-level design structure mechanism. [22]

What is taking place is an attempt to infer the implementation from a design structure that contains insufficient information, and that the information does not lie in the base-level code either. Recognizing this crystallizes the realization that we need to use some other development model artifact for the specification. With Tengger's feature-oriented development model, or more generally with Hyper/J's hyperslices, another artifact is at hand that could be employed. [23]

The solution will not be presented for place reasons. For the solution of Tengger to the design-implementation tension problem, refer to [51].

8.3 Generation of Code from Implementation Level Designs

Code generation from implementation level design refers to UML class diagrams (in our context) code generation. The model and information contained in these class diagrams are a visual representation of the implementation code in terms of UML model elements. Generation of code from low-level design UML model elements is trivial. The UML construct class directly maps to Java classes, the UML construct interface maps to Java interfaces, the UML attributes maps to Java fields, the UML operations maps to Java instance methods etc. But it is not the whole scenario. The constructs dependency, associations, composition, aggregation are difficult to represent directly in Java code. Any representation suffers from loss of information because of the inconsistent concepts between UML and Java.

In this section, trivial code generation for UML model elements class, interface, operation etc. will not be explained. The analysis and explanations are for problem areas. The main problem areas are the constructs (notions) of UML that do not have a direct mapping in the Java language. Therefore, these notions should be implemented by using patterns. Figure 6-8.10 gives an idea of the constructs that do not exist in Java.

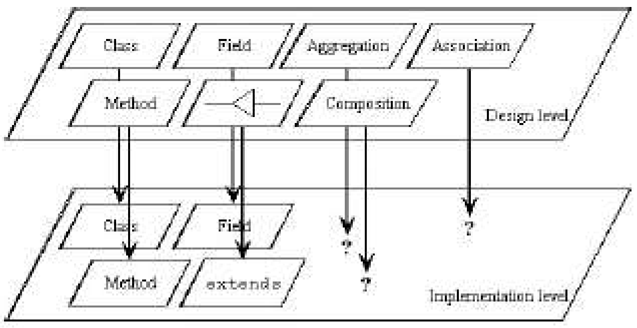


Figure 6-8.10 UML-Java concepts' mapping []

[22] from page 4 of [51]

[23] from page 5 of [51]

The UML notions that are mostly encountered in designs and that do not exist in Java are binary relationships such as associations, composition, aggregation or stereotypes (the list is not exhaustive). This introduces a discontinuity between design and implementation. This discontinuity prevents the creation of software implementations as well as posing extra difficulties to the understanding of software implementations.

Ideally, the UML definitions given for binary class relationships should help narrowing of the gap. Unfortunately the definitions are given in natural language, and they have several ambiguities. Also, there is no hint on implementation of these UML constructs in UML specifications.

8.3.1 Multiple Generalizations

One of the major problems is the multiple generalization capability of UML (corresponding to inheritance in programming languages). Multiple inheritance is by no ways directly mappable to Java. Even the indirect ways analyzed in the previous sections have their own problems. As multiple generalization of UML should not occur in any low-level design representing Java code, this problem is omitted in the context of this thesis.

8.3.2 Associations

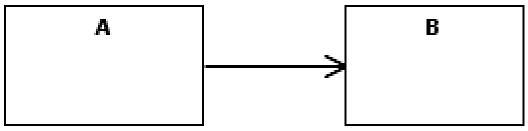
An association declares a connection (link) between instances of the associated classifiers (e.g., classes). It consists of at least two association ends, each specifying a connected classifier and a set of properties that must be fulfilled for the relationship to be valid. The multiplicity property of an association end specifies how many instances of the classifier at a given end (the one bearing the multiplicity value) may be associated with a single instance of the classifier at the other end. A multiplicity is a range of nonnegative integers. The association end also states whether or not the connection may be traversed towards the instance playing that role in the connection (isNavigable), for instance, if the instance is directly reachable via the association. An association-end also specifies whether or not an instance playing that role in a connection may be replaced by another instance [23].

Associations are like composition, aggregation or stereotype some UML constructs that can not be directly mapped to Java code. For the various problems that are solvable, the implementation suggested by authors is also shown.

An association at the design level is a conceptual link between two classes. Each class can have multiple instances involved in the relationship. At the implementation level, [54] defines the association relationship as:

> *A binary class relationship involves the instances of two classes. A binary class relationship is oriented, irreflexive, anti-symmetric at the instance and class level, and asymmetric at the instance level. An association between two classes A and B is the ability of an instance of A to send a message to an instance of B, with the possibility of mutual associations between the instances. The subclasses inherit the association relationship between classes A and B, because in class-based programming languages, subclasses inherit the structure and behavior of their superclasses. For example, if an association relationship exists between two classes A and B, an association relationship also exists between A and SubB, where SubB is a subclass of B.*

The following figure shows an association:



The following piece of code represents an association relationship between two classes A and B:

```
public class A {
   public void operation(B b) {
       b.operation();
   }
}

public class B {
       public void operation() {
       }
}
```

The method `void operation(B)` introduces the association relationship between classes A and B. The piece of code shown here is not the only example representing associations.

8.3.2.1 Multiplicity Problem

The multiplicity of a binary association, placed on an association end (the target end), specifies the number of target instances that may be associated with a single source instance across the given association, in other words, how many objects of one class (the target class) may be associated with a given single object from the other class (the source class).[55]

The potential multiplicities in UML extend to any subset of natural numbers. It does not need to be single interval as (2, *), it can be an interval as (1..4, 7..9, 11..*). Although the second interval is a valid one, it is not used in models. The most usual multiplicities in UML are (0..1), (1..*), (*), (1). The interval can be represented with *(min, max)*.

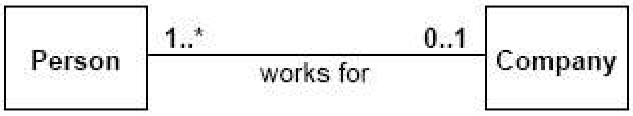


Figure 6-8.11 An association example []

The Figure 6-8.11 shows an association where a Person instance can have 0 or 1 Company instance where as a Company instance can have many Person instances working.

Multiplicity constraint is an invariant which means that it is a condition that must be satisfied by the system. The general practice is that this constraint is not checked always, just at some cases where the programmer decides. Generally it is bound to run-time check. If the programmer forgets to check the conditions, the system can result in an inconsistent state.

8.3.2.1.1 Optional and mandatory associations

An **optional** association is an association for the class on the opposite end where the minimum value for the association is 0. For example in Figure 6-8.11 works for association is an optional one for the Person class. If the value is 1 or greater, it is a **mandatory** association. Conceptually an object taking part in a mandatory association should always be linked with the object of the opposite end; otherwise the system should be in a wrong state. For example in the figure, the Company instances should always be associated with a Person instance.

- An instance of Company is created by an instance of Person and linked to its creator
- An instance of Company is created with an instance of Person supplied as a parameter
- An instance of Company is created and it issues the creation of an instance of Person

In the 3rd case, if the Person has also mandatory associations, handling of the association poses some extra problems. It is not clear whether to create the Company instance first or the Person instance. If the Person instance is created first, till the Company instance creation finishes the Person instance is in an invalid state and vice versa. For the all cases of the inconsistency, see [56].

After the analysis of the problems with the mandatory associations, the authors state that the mandatory associations pose unsolvable problems regarding the creation and deletion of instances and links. They suggest they cannot achieve with a few primitive operations that a mandatory association is obeyed *at any time*, and they cannot isolate, inside atomic operations, the times when the constraint is not obeyed. Therefore the implications of mandatory associations for the implementation have to be relaxed as [34] does. They propose "*not to check the minimum multiplicity constraint when modifying the links of the association (mutator methods or setters), but only when accessing them (accessor methods or getters)*".

8.3.2.1.2 Single and multiple associations

The maximum value for a multiplicity in an association end can be 1 or more. If the maximum value is 1, it is called a *single* association for the class on the opposite end, if greater than 1 it is called a *multiple* association for the class on the opposite end.

Storing the single value of the association is simple to implement. Generally it is implemented as *an attribute of the class at the opposite end*. Multiple associations are more difficult to handle. Some type of collections should be used for storing the values. But the returned value will be no more a type of the class at the other end, but type of the collection such as `java.util.List`.

8.3.2.2 Navigability Problem

The directionality, or navigability, of a binary association, graphically expressed as an open arrow at the end of the association line that connects two classes, specifies the ability of an instance of the source class to access the instances of the target class by means of the association instances (links) that connect them. If the association can be traversed in both directions, then it is *bidirectional* (two-way), otherwise it is *unidirectional* (one-way).[24]

A navigable association end, which is referenced by its rolename, defines a pseudoattribute of the source class, so that the source instance can use the rolename in

[24] from page 142 of [56]

expressions in the same way as it uses its own attributes. [25]An instance can communicate (by sending messages) with the connected instances of the opposite navigable end, and it can use references to them as arguments or reply values in communications. Similarly, if the association end is navigable, the source instance can query and update the links that connect it to the target instances.

The Figure 6-8.12 shows examples for the unidirectional and bidirectional associations. The association seen in Figure 6-8.12(a) is a unidirectional association whereas the one seen in (b) is bidirectional. Key instances have access to the Door instance it opens, but the Door instance does not have the Key instance. On the other hand, a Man instance has access to the *wife* Woman instance, and a Woman instance has access to the *husband* Man instance.

The UML specification says that the arrowheads in Figure 6-8.12(b) can be omitted. But this leads to an ambiguity in the meaning. The omitted arrowheads can also mean that the navigability is not decided or known yet. So, if the arrowheads are omitted, it can not be decided whether it is a bidirectional association or an association with an unspecified navigability.

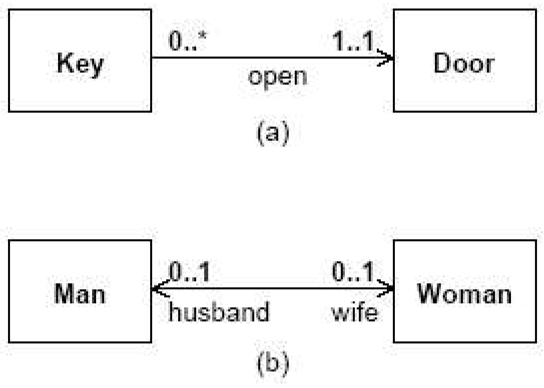


Figure 6-8.12 Unidirectional and bidirectional association examples [56]

8.3.2.2.1 Unidirectional associations

A *single unidirectional association* is very similar to a single valued attribute in the source class, of the type of the target class: an embedded reference, pointer, or whatever you want to call it. The equivalence, however, is not complete. Whereas the *attribute value* is "owned" by the class instance and has no identity, an *external referenced object* has identity and can be shared by instances of other classes that have a reference to the same object [[57]] (see Figure 6-8.13). Anyhow, the equivalence is satisfactory enough to serve as a basis for the implementation of this kind of associations. In fact, in Java there is no difference at all: except for the case of primitive values, attributes in Java are objects with identity, and if they are public you cannot avoid them to be referenced and shared by other objects.[26]

[25] from page 354 of [55]

[26] page 143 in [56]

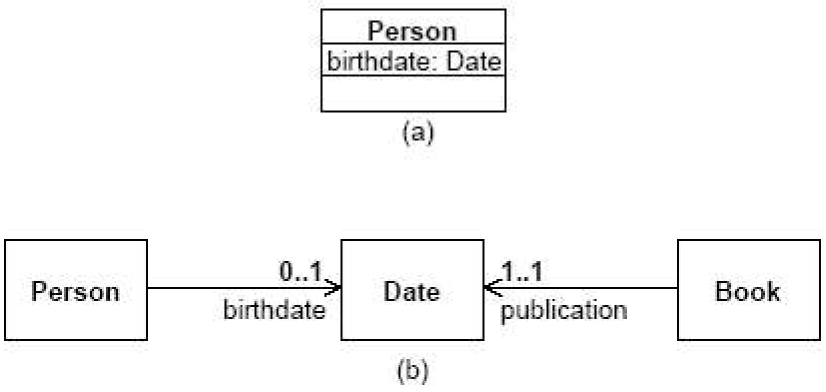


Figure 6-8.13 Association example with Java class [56]

A *multiple unidirectional association* is very similar to multi-valued attribute. It can be implemented in the same manner using collections of Java.

It is suggested in [56] that the multiplicity constraint in a design model can be specified only *for a navigable association end*. The multiplicity constraint should be evaluated within the context of the class that owns the association end; if that class knows the constraint, then it knows the association end, that is, the end is navigable. The number of objects connected to a given instance can not be restricted unless this instance has some knowledge of the connected objects, that is, unless the association end is made navigable. Therefore when it is necessary to have a multiplicity other than (0..*), the association should be also navigable. *In consequence, the associations which are not navigable but have multiplicities other than (0..*) should not be accepted at code generation*.

8.3.2.2.2 Bidirectional associations

The partial equivalence between attributes and unidirectional associations is not any more found among bidirectional associations. Instead, an instance of a bidirectional association is more like a ***tuple of elements***[27]. Combining the multiplicities in both association ends, there are three possible cases: *single-single*, *single-multiple*, and *multiple-multiple*.[56]

Single-single bidirectional associations can be implemented by having two synchronized unidirectional associations forming the bidirectional association. The example seen in Figure 6-8.14 is an example for this.

[27] UML 1.4 Specification, in [23] p. 2-19

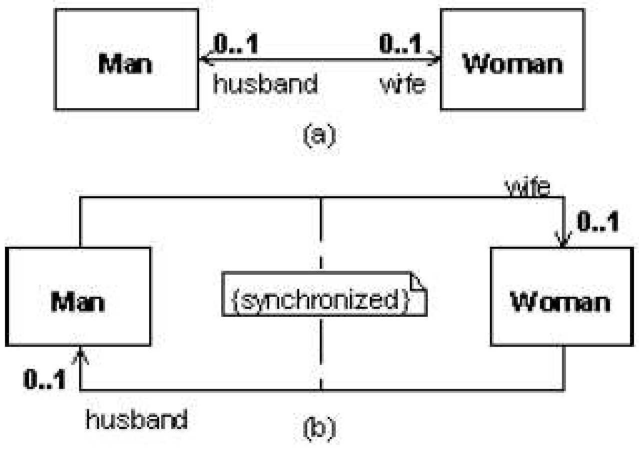


Figure 6-8.14 Bidirectional association example [56]

A single-multiple association can be implemented in a way similar: 2 synchronized unidirectional associations where one is single association and the 2nd a multiple association. It is the normal behavior that the synchronization gets more difficult.

A multiple-multiple association is implemented with two synchronized unidirectional multiple associations. The multiple associations are implemented with collections as specified before. Synchronizing the multiple-multiple collections poses new problems as analysed in [56].

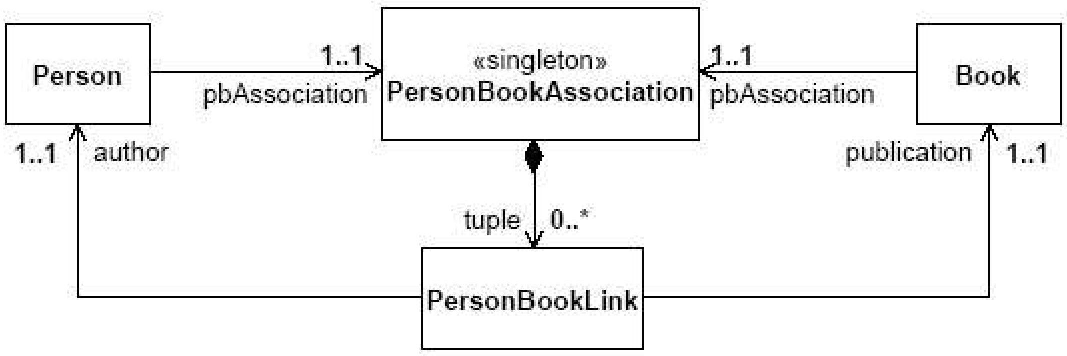


Figure 6-8.15 Multiple-multiple association example [56]

The UML specification says, an association is defined as a "set of tuples" meaning that one can not have twice the same tuple in the collection of links of an association. The implementation explained already ensures this. The associations can be also implemented in another way, by using an extra class defined for representing the associations with both ends. By this way, there is no need to synchronize any instances on classes. The unidirectional association ends can already be stored in this association class.

In Figure 6-8.15, the class PersonBookAssociation represents the multiple-multiple association between Book and Person. This scheme reifies the links for the association and makes them objects of their own as in [58]. The links or the collections of the associations can be stored in the singleton instance PersonBookAssociation. But this way of implementing the associations does not guarantee the UML constraint "uniqueness of the each tuple" for associations specified before. It is also heavier and more expensive to implement the associations in this manner.

8.3.2.3 Visibility Problem

The UML standard states for the visibility of an association end that "It specifies the visibility of the association end from the viewpoint of the classifier on the other hand". There are four kinds of visibilities in UML:

- `public` : Other classifiers may navigate the association and use the rolename in expressions similar to the use of a public attribute.
- `protected`: Descendants of the source classifier may navigate the association and use the rolename in expressions, similar to the use of a protected attribute.
- `private:` Only the source classifier may navigate the association and use the rolename in expressions, similar to the use of a private attribute
- `package:` Classifiers in the same package (or a nested subpackage) as the association declaration may navigate the association and use the rolename in the expressions.

As can be seen clearly, the choices of the attributes in UML are not by coincidence same as Java. The default access control level in Java is *package* visibility.

The visibility of an association introduces no special problems in case of unidirectional associations. The only class who has access is already only the opposite end of the association end. But for the bidirectional associations the situation is quite different about visibilities. In principle, an association with one or both ends of private visibility should also not cause any extra problems. For example a Bank-Client association with both private ends wouldn't let the other end of the association to access. Although the expected behavior of association ends with visibilities is not to cause problems, this is not the case. The problems will be explained with an example.

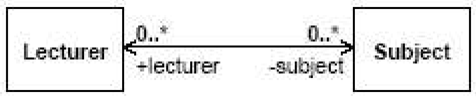


Figure 6-8.16 Lecturer-Subject Association [56]

In Figure 6-8.16, there is a private association end *subject* and a public association end *lecturer*. The association end *lecturer* can be accessed by any model element, whereas the association end *subject* can only be accessed by Lecturer class. According to the associations between Subject and Lecturer with the existing visibilities, there is an implication that Subject class do not know about the *subject* association end. The class Subject knows that it is associated with Lecturer class, but it does not know that the Lecturer class is associated with it in return. There is an ambiguity whether this association is really a bidirectional association. The two-unidirectional association ends' synchronized way of representing the bidirectional association does not work in this case because only Lecturer can access the association end subject.

For the reasons explained above, a public-private bidirectional association can be managed only from the class that owns the private end, and other classes, including the class on the other end of the association, can have only indirect access if this class provides the adequate public methods. A private-private bidirectional association, on the contrary, can not be managed at all. In consequence, bidirectional associations with visibility other than public or package in both ends must be rejected in code generation. The authors' thoughts are that the result encountered here is not only a bias of the particular implementation, but a real semantic difficulty of the definition of visibility in bidirectional associations. Visibility in

UML is not specified for associations but for association ends and it is assimilated to the visibility of attributes.[28] In UML a definition of visibility that fits better with the concept of bidirectional association is necessary. [56]

8.3.3 Aggregations

Wien [[59]] states aggregations are used to build composite objects, where aggregations need a special construct called containment constructs. An aggregate object is an object with references to dependent objects and independent objects.

Rational Rose considers aggregation entirely conceptual and states "it only distinguishes a whole form a part. Simple aggregation does not change the meaning of navigation across the association between the whole and its parts nor does it link the lifetimes of the whole and its parts".

An association may represent an aggregation; that is, a whole/part relationship. In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations.[29]

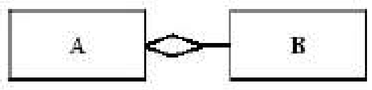


Figure 6-8.17 Aggregation in UML

The Figure 6-8.17 shows an aggregation relationship between the classes A and B. The definition of aggregation at the implementation level from [54]is:

An aggregation relationship exists when the definition of one class contains instances of the other class. It distinguishes a whole from a part. The aggregate class must define a field (or an array field, or a collection) of the type of the aggregated class. Instances of the aggregate class send messages to the referenced instance of the aggregated class.

The following code snippet taken from [54] is an example showing aggregation relationship:

```
public class A {
     public B b;
     public A(B b) {
        this.b = b;
     }
     public void operation(){
        this.b.operation();
     }
}

public class B{
     public void operation(){
     }
}
```

Here the method `void operation()` and field `B b` introduces the aggregation relationship between classes A and B.

[28] page 2-23 in UML Specification [23]
[29] page 122 in UML Specification [23]

8.3.4 Compositions

Composition is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. This means that the composite object is responsible for the creation and destruction of the parts. In implementation terms, it is responsible for their memory allocation. If a composite object is destroyed, it must destroy all of its parts. It may remove a part and give it to another composite object, which then assumes responsibility for it. If the multiplicity from a part to composite is zero-to-one, the composite may remove the part, and the part may assume responsibility for itself, otherwise it may not live apart from a composite. [23] [30]

The definition given at [54] for the composition at implementation level is:

A composition is an aggregation between two classes, with a constraint between the lifetime of the instance of the whole and the lifetime of the instances of the part and a constraint on the ownership of the instance of the part by the instance of the whole.

The whole may take direct responsibility for creating the part or it may accept an already created part. A whole may pass a part to some other whole, which then assumes responsibility for it. When a whole is deleted, all its parts are deleted as well.

An instance of the whole owns the instance of its part. The instance of its part must not belong to any other whole (either through an aggregation relationship or a composition relationship). The instance of the part is exclusive to the instance of the whole.

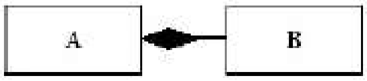


Figure 6-8.18 Composition

The Figure 6-8.18 shows a composition relationship between classes A and B. The Java code snippet shown below represents one of the ways for expressing the composition at code level. Here the means how composition is represented is more complex than the ones for aggregation. Not only it has the static structure in the program, but also runtime behavior of the code is introduced the composition. The method `void attach(B)`, `void operation()` and field `private B b` are the static portion. The dynamic portion comes from the garbage collector. The JVM garbage collector destructs `b` before instance of `A` which ensures the conditions on lifetimes of classes having a composition relationship.

```
public class A {
	private B b;
	public void attach(B b) {
		this.b = b;
	}
	public void operation() {
		this.b.operation();
	}
}

public class B {
	public void operation() {
	}
}
```

[30] Page 291 in the referenced documentation

8.3.5 Stereotypes

A stereotype is, in effect, a new class of metamodel element that is introduced at modeling time. It represents a subclass of an existing metamodel element with the same form (attributes and relationships) but with a different intent. Generally a stereotype represents a usage distinction. A stereotyped element may have additional constraints on it from the base metamodel class. It may also have required tagged values that add information needed by elements with the stereotype. It is expected that code generators and other tools will treat stereotyped elements specially. Stereotypes represent one of the built-in extensibility mechanisms of UML. [23]

Stereotypes in low-level designs are not reflected to Java code. The generation of Java code for a stereotyped UML metamodel element depends on the context where the stereotype is used. Therefore, the generation of the code also depends highly on the design model in which the stereotype is created. A stereotype "unknown" can have different meanings in different modeling environments. The stereotypes are mainly used for code generation from high level design models whose usages are given in examples in section 8.2.

8.3.6 Summary

Implementation level UML models are a visual representation of source code. There is no difference in the level of abstraction between the model and the source code it represents. Such models are often too crowded. These models are very often results of reverse engineering from source code, or roundtrip engineering. They are very often used in roundtrip engineering process. The changed code is reverse engineered to update the models and the changes on the models propogate to the source code.

Not all of the UML concepts are well-representable in the programming languages. For example, in case of the programming language Java, multiple inheritance which exists in the UML is not representable. The language does not support multiple inheritance. When the code is generated from implementation level UML diagrams, multiple inheritance is often rejected. When not rejected, custom code templates are used to represent the multiple inheritance in the code.

The only problem in mapping is not the multiple inheritance. Composition and aggregation which exist in the UML are not very well defined. They are only defined in natural language in the UML specifications. The code generated for them always has some level of loss of information. Most often, for not losing the UML model details during code generation; comments describing the UML model are added to the source code. Especially at roundtripping, these added comments are very important.

Another very big problem during mapping UML models into source code is the associations. Associations are only a UML concept. In no programming language, there exists a similar concept. The associations have multiplicities, navigability, association ends, visibilities at the association ends and rolenames which need to be represented in the source code. To generate code for associations with multiplicities of *, often a collection is used. The visibilities of association ends are at some cases not generatable in the source code as the visibilities do not allow accesses. Navigability is also a similar issue. Most of the code generation tools do simply omit the model details during code generation.

8.4 Eclipse Modeling Framework (EMF)

EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications. [60]

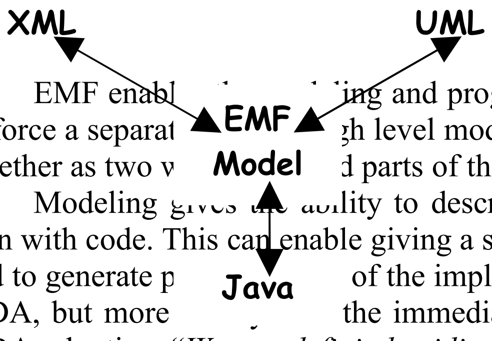
Developing applications generally starts with considering the design model, after that the graphical user interface and the rest is taken care of. The Eclipse Modeling Framework is designed to ease the design and implementation of a structured model. The Java framework provides a code generation facility in order to keep the focus on the model itself and not on its implementation details. The key concepts underlying the framework are: meta-data, code generation, and default serialization. [61]

At the beginning, EMF is started as a MOF of OMG implementation and with time it evolved. EMF is an enhancement of MOF2.0. EMF is open source code that enhances the MOF 2.0 Ecore model and restructures its design in a way that is easy for the user.

The Eclipse Modeling Framework is part of the Model Driven Architecture and it is current implementation of a portion of the MDA in the Eclipse family tools. In MDA, the model itself is described in a metamodel. Then, by using mappings, the model is used to generate software artifacts, which will implement the real system.

Two types of mappings are defined: *Metadata Interchange*, where documents like XML, DTD, and XSD are generated; and *Metadata Interfaces*, which target Java or any other language and generate IDL code.

EMF unifies the following three technologies, Java, XML and UML. Regardless of which one is used to define it, an EMF model is the common high level representation that glues them together.



EMF enabl ing and programming to be thought as one thing. It does not try to force a separat gh level modeling from low level implementation. It brings them together as two v d parts of the same job.

Modeling gives the ability to describe how an application is supposed to more easily than with code. This can enable giving a solid, high-level way both to communicate the design and to generate p of the implementation code. EMF is a technology on the way to MDA, but more the immediate widespread adoption. The EMF creators say for MDA adoption *"We are definitely riding the bike, but we don't want to fall down and hurt ourselves by moving too fast. The problem is that high-level modeling languages need to be learned and since we're going to need to work with (for example, debug) generated Java code anyway, we now need to understand the mapping between them. Except for specific applications where things like state diagrams, for example, can be the most effective way to convey the behavior, in the general case, good old-fashioned Java programming is the simplest and most direct way to do the job."* [62][31]

The EMF developers call the EMF as being in the middle of two extreme views of modeling, those who say "I don't need modeling" and those who say "modeling rules". EMF is thought to be the least common denominator of those two views; it mixes the right amount of programming and modeling at the current software technologies. An EMF model is the class diagram subset of the UML. It has a simple model of classes, or data, of the application.

[31] On page 13, paragraph 4

8.4.1 EMF Model inputs

EMF is driven by a metamodel that can be in the form of:

- Ecore model
- Rose .mdl model
- XSD Schema (provides automatic serialization according to the schema)
- MOF 2 EMOF
- Annotated Java

8.4.2 Ecore

Ecore is the metamodel of EMF. It is used to represent models in EMF. The EMF models are defined using Ecore metamodel. Ecore is also defined by Ecore itself. The simplified class diagram for Ecore metamodel is as seen in Figure 6-8.19. The Ecore metamodel is a subset of the UML class diagrams. EMF Ecore metamodel and UML metamodel are on the same levels, *M2* w.r.t MOF metalevels. Besides the difference in the metamodel, another main difference is that, to define UML metamodel MOF meta-metamodel is used. But in EMF Ecore, Ecore M3 level meta-metamodel is used.

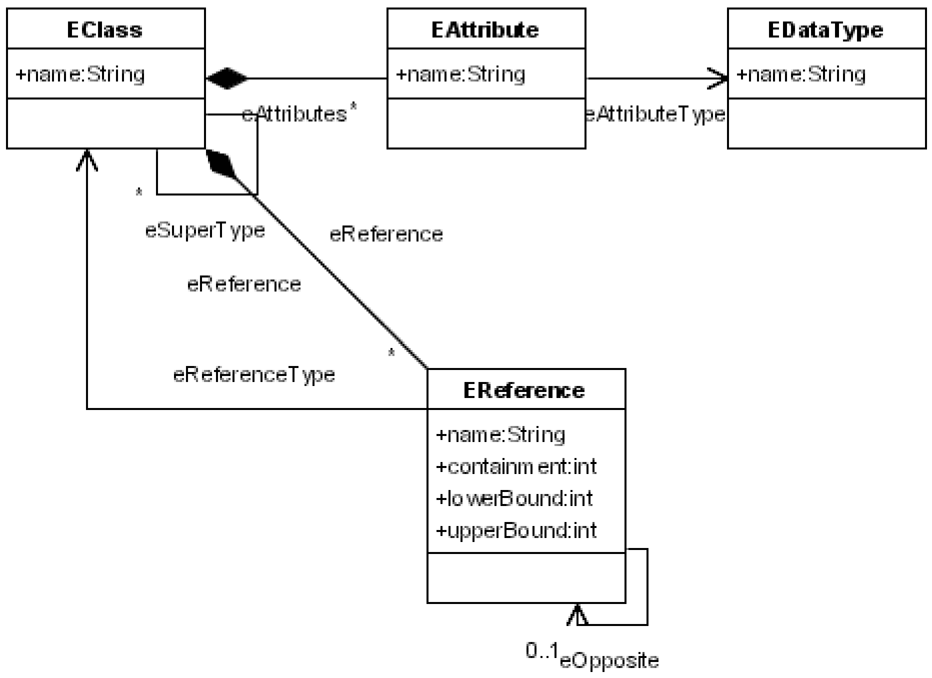


Figure 6-8.19 Ecore Metamodel of EMF

8.4.3 Model

An EMF model is specification of an application's data. It consists of object attributes, relationships (associations) between objects, operations available on each object and some simple constraints (e.g. multiplicity) ob objects and relationships.

An example model defined with Ecore can be written in many formats.

- XMI format is:

```
<eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
        <eReferences name="items" eType="#//Item" upperBound="-1"
containment="true"/>
           <eAttributes name="shipTo" eType="ecore:EDataType
http:...Ecore#//EString"/>
          <eAttributes name="billTo" eType="ecore:EDataType
http:...Ecore#//EString"/>
</eClassifiers>
```

Table 6-8.3 The model in XMI format

- The UML diagram for this model is as follows:

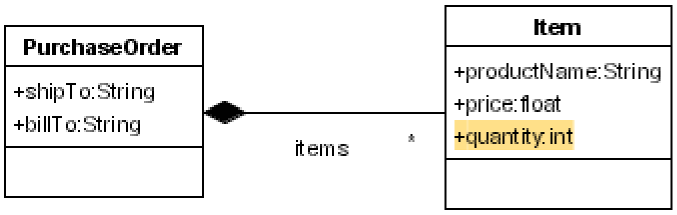


Figure 6-8.20 UML Diagram representation of the Model

- XSD schema as seen in Table 6-8.4:

```
<xsd:complexType name="PurchaseOrder">
    <xsd:sequence>
        <xsd:element name="shipTo" type="xsd:string"/>
        <xsd:element name="billTo" type="xsd:string"/>
        <xsd:element name="items"  type="PO:Item"
               minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Item">
     <xsd:sequence>
        <xsd:element name="productName"
type="xsd:string"/>
       <xsd:element name="quantity" type="xsd:int"/>
       <xsd:element name="price" type="xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
```

Table 6-8.4 XSD schema of the Model

8.4.4 Code Generation

The EMF is composed of three main parts:

- **EMF** - The core EMF framework includes a meta model (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.
- **EMF.Edit** - generic reusable classes for building editors for EMF models.
- **EMF.Codegen** - capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked.

Code generation with EMF has three levels:

1. **Model** - provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package (metadata) implementation class.

2. **Adapters** - generates implementation classes (called ItemProviders) that adapt the model classes for editing and display.

3. **Editor** - produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors and serves as a starting point from which to start customizing.

All generators support regeneration of code while preserving user modifications. The generators can be invoked either through the GUI or headless from a command line.

The EMF code generation is only for class diagrams, the static structure of an application, for its metamodel only comprises the class diagrams part of UML. The generation patterns or algorithms are buried in the framework, they can not be customized. For example, the code generated for an *EClass* instance has a predefined structure, an interface and an implementing class.

For further information on EMF and EMF code generation, please refer to [60, 62, 61]. EMF is explained as it is another view on using common models and generating code of them for XSD, Java and XMI.

8.5 Domain Specific Languages (DSL): Replacement for UML?

Domain Specific languages are the languages that instead of being focused on a particular technological problem such as programming, data interchange or configuration, are designed so that they can more directly represent the problem domain which is being addressed. DSLs are often called *modeling languages*, and are used to build models of the domains they address. DSL is a language which is targeted to solve a particular kind of problem, rather than a general purpose programming language. DSLs are not a new kind of languages; it exists since long, almost as old as the computing.

The UNIX community uses DSLs a lot. The most common Unix-style approach is to define language syntax and then either use code generation to a general purpose language, or write an interpreter for the DSL. UNIX provides many tools for making these tasks easier and faster.

Domain-specific languages can be created for numerous problem domains: Some well known applications are in telecommunications, investment banking, public transport, space exploration, and in many other areas.

DSLs have been used by Smalltalk and Lisp communities, but their approach of using it is different than UNIX. Rather than creating a new DSL, they insert and adapt it into the existing language.

DSLs are the Microsoft's choice for model driven development. DSLs will be used by Microsoft instead of adopting the MDA technology. For the reasons explained in [63], the MDA standard is not adopted by Microsoft. But this does not mean that they are not focusing on model driven development. As the technologies get more and more complex and the fact that software developers need to know many technologies, nowadays system development is getting more complex and the industry understands the importance of model driven development better. The future of Microsoft view of DSLs is not clear yet, as the future of MDA. Currently it can not be said that MDA is a standard accepted by the software industry. Although parts of it is used and adopted, the idea of composing a running system from a model is not clear to many. The inadequacy of MOF is another reason that MDA is not adopted.

For the time being, Microsoft plans to use the DSLs in model driven development for distributed service-oriented applications. In future, they plan to span this to areas like code visualization, business modeling, and opening up the tool-building environment itself, as well as developing industry partnerships through which other domains will be supported.

How Microsoft will use DSLs at model driven development has not been clear yet. A summarized view of usage from [63] is as follows:

They will be taking advantage of the UML to the extent it provides recognizable notation for well understood concepts. Whenever they find areas where UML notation is not enough, they will enhance it by their own means. For example, as a UML class can not directly represent a C# class, they will be providing a new UML class for C# classes. It will be similar for other non-represented concepts. The enhancement done will allow full roundtripping which is very important for customers, namely software developers. The diagrams representing .NET models will not be UML-standard compliant. For any domain that UML does not address problems or provide notations, again they will be creating their own conventions. An example domain for this development of models of logical data centers, which is used at Microsoft to model the deployment of distributed applications.

The reasons MOF is not supported by Microsoft are as follows according to [63]:

1) it is not yet a single stable standard;
2) using it as the language for designing our tools would have far reaching practical consequences on those tools that we are unwilling to accept;
3) addressing the missing elements of the MOF required by commercial grade implementations (e.g., notation, transactions, events, etc.) will continue to introduce major changes into the MOF specification; and
4) MOF's approach to model serialization fails to meet its goals.

As the time of this writing, Microsoft and JetBrains[64] are the only software vendors which focus on DSLs for model driven development. The future will tell us what will happen to the MDA standard and DSLs.

Grady Booch's view on DSLs are as follows: *DSL's absolutely have a place at MDA, DSLs are basically at the PIM level that's what a DSL is, but it can be done with standard UML, and it should be able to be exchanged with UML and that's our position*.[32]

[32] It is taken from an interview found at http://www.codegeneration.net/tiki-read_article.php?articleId=59

DSLs are now, as stated before, mainly used by Microsoft and JetBrains in model driven development. Microsoft aims at creating models which have full roundtripping support. On the other hand, JetBrains has a view where the model driven development should be completely one-way, namely code generation. The user will not be allowed to change the code, if she changes, the tool is not expected to behave correctly. The project Fabrique™ [65] which is currently under development is a Rapid Application Development platform for developing sophisticated web and enterprise applications. It uses DSLs for code generation.

DSLs are explained in this thesis, because it is a big competitor for UML as well as UML vendors, and MDA. Although the comments of Grady Booch on DSLs are so that they have definitely part in MDA, this is not totally clearly now. In MDA, UML and MOF standards are the main standards the MDA is based on. But DSLs are just mentioned because Microsoft started working with them for model driven development. Which lives longer or better said, which will be surviving will be understood and seen with time.

8.6 Summary

In this chapter, there was a detailed analysis of code generation from UML models. The problems are analyzed and solutions proposed in papers are given. Code generation is analysed with respect to two different views: *Generation of code from high level UML models* and *generation of code from implementation level UML models*.

Main problems analyzed are at mapping generalization and association (containing aggregation and composition) relationships in UML. Multiple generalization is not directly mappable to Java. Solutions proposed in the literature are analysed.

At the end of the chapter, EMF overview is given. EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model.

Domain Specific languages are the languages that instead of being focused on a particular technological problem such as programming, data interchange or configuration, are designed so that they can more directly represent the problem domain which is being addressed. In context of this thesis, Code generation from UML models is analysed and important. But as review is done, the importance of DSLs is understood. They are used at Microsoft and Jetbrains and seem to be a big competitor to UML at code generation.

9 Integrating *Poseidon* with the Eclipse Platform: Design and Implementation

All the review done in the previous chapters was part of the main aim of this thesis: Integrating Poseidon with the Eclipse Platform. Roundtrip engineering which means synchronizing the source code and the design model at any moment of time can be best obtained with a tool specialized for UML and an environment for programming, i.e. Integrated Development Environment.

For the reasons explained in Chapter 5, Eclipse is the most suitable environment for integrating any tool with an IDE. There are hundreds of plugins available for the Eclipse environment. When a developer needs to edit ".xsd" files, she only needs to find the appropriate Eclipse plugin(s) for XML and XSD editing. For XML, Databases, Web Development, UML, etc., numerous plugins are available.

In this chapter, an overview of the design and the architecture of the Eclipse plugin implemented for integrating Poseidon with Eclipse as well as implementation details are given.

9.1 System Design

9.1.1 Architecture of the System

The system is composed of 3 main components as seen in Figure 7-9.1: Poseidon, Poseidon Plugin and Eclipse. Poseidon component is the Poseidon tool which is integrated to the Eclipse platform via platform extension points.

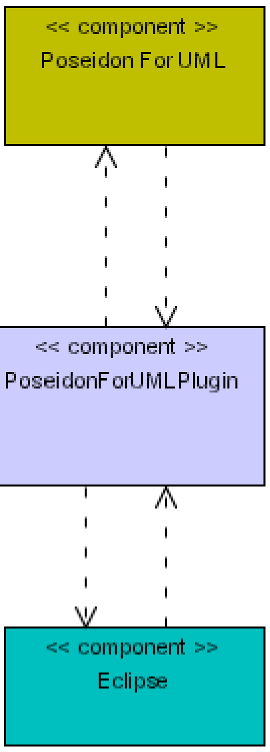
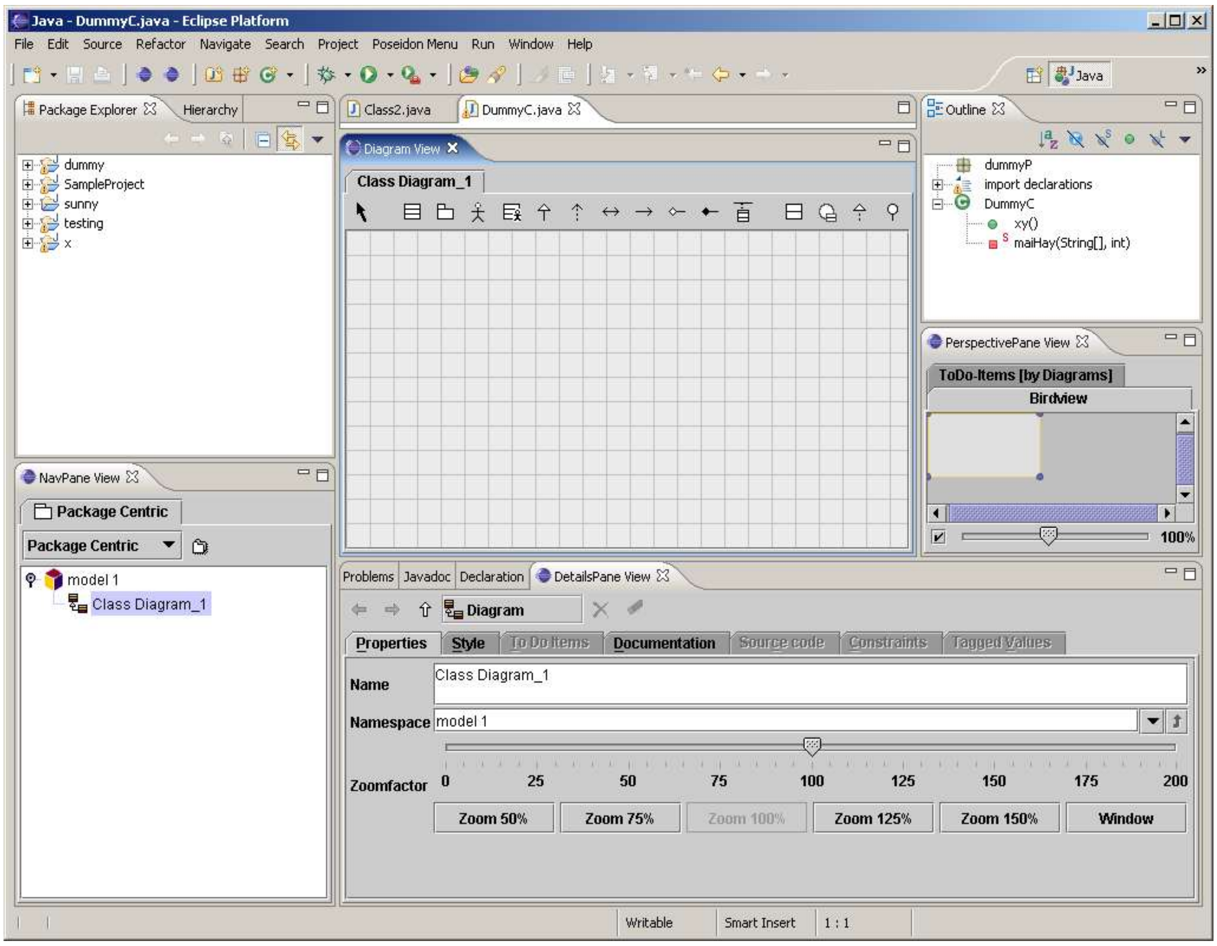


Figure 7-9.1 System Architecture

The component Poseidon for UML plugin is the main contribution to the Eclipse platform. Poseidon component is integrated to the platform by this plugin. It contributes to the Eclipse platform new views, actionsets and popup menus. It is a bridge between the Eclipse platform and Poseidon.

The component Eclipse is the platform to which Poseidon gets connected. Eclipse component provides many extension points by which the platform can be extended. The Eclipse Platform is a huge component.



Name: Class Diagram_1
Namespace: model 1

Figure 7-9.2 Eclipse Platform with Poseidon Plugin [33]

Figure 7-9.2 shows the Poseidon views in the Eclipse Platform, better said, in the Java perspective of JDT.

9.1.2 Visual Integration

The visual integration between Poseidon and Eclipse platform is achieved through a number of implemented extension points. The *views*, *popup menus* and *actionsets* extension points are extended by new extensions.

The component diagram shown in Figure 7-9.3 gives an overview on visual integration. The views, popup menus and actionsets use mainly the SWT GUI library, as the interfaces to extend these extension points depend on SWT. The views indirectly use Swing GUI library as

[33] Integration with Eclipse 3.0 Version

seen in the figure. Poseidon panels are inserted to views which use Swing GUI for GUI programming.

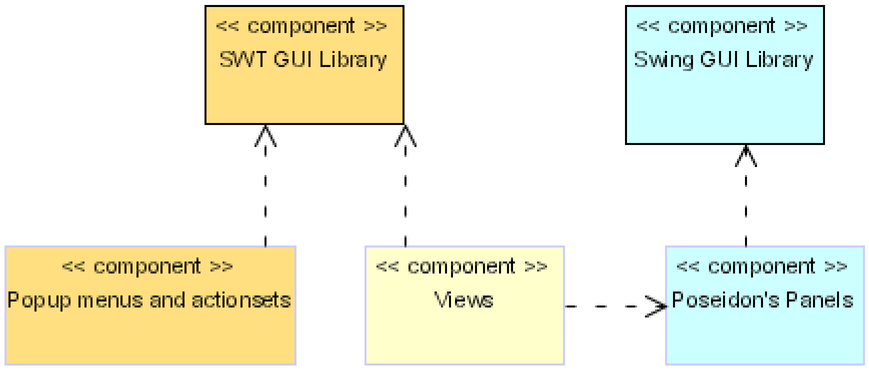


Figure 7-9.3 GUI library dependencies

9.1.2.1 Views

Poseidon plugin provides 4 Eclipse views for Poseidon windows. They are:

- Diagram view
- Details Pane view
- Navigation Pane view
- Perspective Pane view

The diagram view corresponds to the Poseidon's *Diagram panel* in the standalone version. All UML diagrams can be created and seen in this view.

Details pane view corresponds to the *Details Pane panel* of the Poseidon tool. It shows all the details for any UML model element found in the UML model. The model element properties such as name, visibility, modifiers, attributes, operations, associations, implements list, extends list as well as diagram style, UML documentation and constraints and tagged values information is available in this window.

The Navigation pane view corresponds to the *Navigation panel* of the Poseidon tool. The Navigation Pane enables the traversal of the UML model elements. The users can also create, delete or edit UML model elements from the navigation pane.

The perspective pane view is the *perspective panel* of the Poseidon tool. The bird view of the active UML diagram is shown by this panel. The user can determine the level of details she wants to see in the active UML diagram by adjusting the percentage of the bird view of it.

To implement the Poseidon panels as views in Eclipse, the respective platform extension points were implemented. The extension point which needs to be implemented for creating a new view in Eclipse is the *"org.eclipse.ui.views"* extension point. The views are added to the Java perspective in Eclipse. Another possibility is to create a new perspective for the UML interaction in Eclipse and add the views to this perspective. This is not implemented. It can be a candidate for future work.

9.1.2.2 Popup Menus

Another extension point extended by Poseidon plugin is popup menu extension point. To handle project-related actions, popup menu for `IJavaProject` instances, i.e. Java projects in the workspace, is extended by a new menu "**UML**". This menu contains 4 menu items. They are *Create Poseidon-UML Model*, *Open Poseidon-UML model*, *Import Poseidon-UML model* and *Reverse engineer*.

9.1.2.3 ActionSets

A plugin can contribute menus, menu items, and tool bar items to the workbench menus and toolbar by using the "*org.eclipse.ui.actionSets"* extension point. The Poseidon plugin contributes by "**Poseidon Menu**" to the workbench menu. This menu currently contains only very simple Poseidon actions such as printing the active UML diagram or saving the active UML diagram as graphics.

9.1.3 Synchronization of UML Models and Java Code

The aimed synchronization between UML models and Java source code is achieved by the following procedure:

When Eclipse platform is started and the Poseidon plugin gets activated, the main Poseidon application starts. This initializes all UML-related features.

For any Java project in the workspace with the Java nature, the user can create the UML model representation if the model does not exist yet. When the UML model for a Java project is first created, reverse engineering of all of the Java project source code is done. If the model already exists, the user can open the existing model and edit it.

When the UML model for the Java source code is created by reverse engineering or opened, the UML project containing the model becomes the active project in the Poseidon. The views for Poseidon in the JDT hold information about this current model. Any interaction done after opening the UML model or creating the UML model triggers synchronization. For example, if the UML model is edited or new UML model elements are created or existing UML model elements are deleted, then necessary functionality to update the source code of the mapping Java project is executed. Whenever a new UML class is created, a Java class with the same name is added to the project with the respective namespace. Whenever the source code is edited in the JDT, the UML model is updated accordingly.

Figure 7-9.4 shows the role of the Poseidon's own plugins in the overall architecture. Poseidon can also be extended similar to Eclipse. Most of the code generators are added to Poseidon as plugins. UML-Java generator plugin creates the Java source for the UML model elements that can be generatable such as UML classes, interfaces, attributes, associations etc. This generator does *one-way code generation*. Every time UML model gets changed, the source code needs to be generated again if the file system is the target. Poseidon tool provides a *"Source code"* tab. This tab provides an editor with selectable target language as Java, C++, etc., which shows the code generated for the active selected UML model element.

The Reverse Engineering plugin can reverse engineer Java source code. It can reverse engineer arbitrary code. The Eclipse plugin for Poseidon is dependent on this plugin as it is used in synchronization of source code and UML models. This component can only reverse engineer single files as the smallest Java unit. It can not reverse engineer a portion of a file. Therefore, the use of this plugin is limited to cases where complete source files or Java projects are reverse engineered. At our specific case, it is used at Create Poseidon UML Model action (see 9.2.3.1). Any further reverse engineering functionality is implemented using Java Model API.

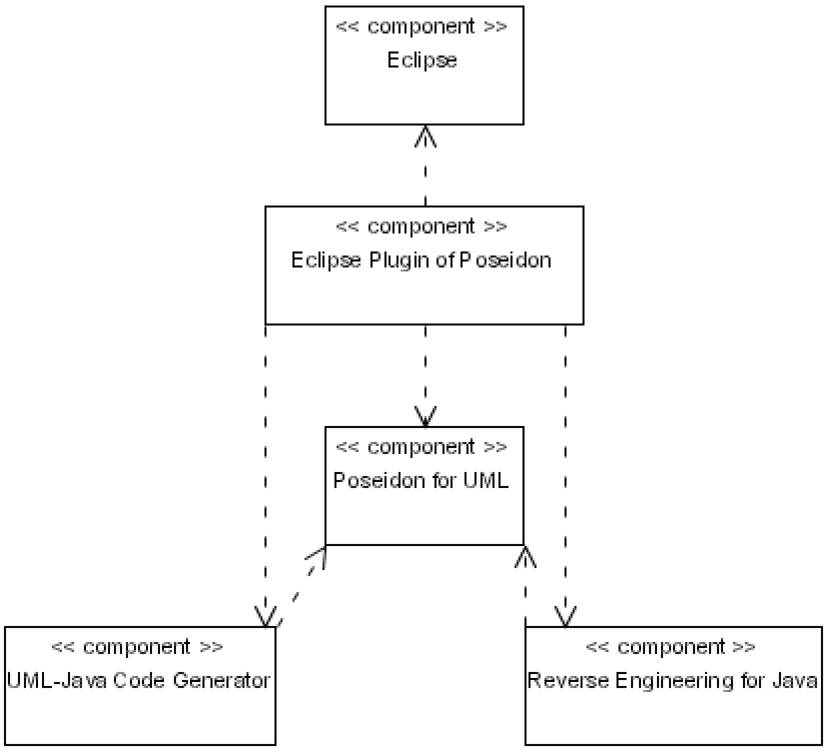


Figure 7-9.4 The role of Poseidon's own Plugins

As seen in the Figure 7-9.4 component diagram, plugins Uml-Java code generator and Reverse Engineering for Java depend on Poseidon. They use the API Poseidon provides for plugins. The UML model as well as GUI can be accessed by this open API.

The Eclipse plugin for Poseidon integration is dependent on Poseidon as well as Poseidon's own plugins mentioned for generation of code and reverse engineering. The synchronization uses UML-Java code generator for the initial code generation of model elements. As the model elements are edited by user, the source code is not re-generated; it is modified by help of the AST and Java Model API of JDT.

9.1.3.1 Project Handling

For any Java project in the workspace, to start synchronized interaction, the user needs to first have the corresponding Poseidon UML model of the project. There are two conditions considered:

- *Existing UML Model*: If the synchronized UML model of the Java project already exists, the user only needs to open the model. She can open the model by executing "*Open Poseidon-UML Model* action" in the "*UML"* menu.[34]
- *No UML Model*: If the UML model for the Java project does not exist, she can create the model by executing *"Create Poseidon UML model*" action in the "*UML"* menu of the project.

The existence of the UML model is determined as follows: For a Java project with name "***myJavaProject***" in the workspace, the UML project with the same name is looked for under

[34] Here we make an assumption that the existing UML model is in synchronized form.

the project root path, i.e. for UML project "***myJavaProject.zuml***" (.zuml is the extension for Poseidon projects). If this file exists, it contains the UML model which is the synchronized model for this Java project.

9.1.3.2 UML to Java Changes

A user can edit the open UML model in implemented views. The changes on UML model should trigger Java source code changes whenever the source content needs to be updated. The changes can be grouped in 3 main groups:

- Creation of new UML model elements
- Deletion of UML model elements
- Modification of existing UML model elements

In Poseidon, the changes on UML models happen in commands based on *Command Pattern*. The simple commands only handle UML model. They do not modify source code.

To update the source code corresponding to the modified UML model elements, the command factories in Poseidon are replaced by new ones. The new ones contain commands which handle UML model as well as Java source code. By replacing the command factories, all possible UML model change events can be captured and implemented for integration.

The *UML-Java code generator* component seen in Figure 7-9.4 is used for Java code generation whenever a Java element is created. By the UML model changes as name, visibility or modifier changes, the update of the code is done by using Java model and AST API.

9.1.3.3 Java to UML Changes

As part of the synchronization process, Java-to-UML changes are also handled. The changes on Java elements are notified via `IElementChangedListener` which was described in the chapter 5. This listener does not notify the changes very detailed. In the implementation, it is still used.

When a Java element is added, removed or modified, the changes are notified by the Eclipse platform as the class implementing `IElementChangedListener` is registered to the `JavaCore`. As the changes are taken, the respective Poseidon commands are executed by the command factory manager. The decision which UML command(s) to execute is done after analyzing the notified changes. The arguments to the UML commands are extracted by analyzing the piece of code modified.

Figure 7-9.5 shows the component dependencies between Poseidon plugin, the Java Model of Eclipse and MDR components. As the figure shows, the semantic model (sm) commands for UML are using MDR component. The sm commands also use Java Model and AST API to update or get information about Java classes, packages, methods, etc.

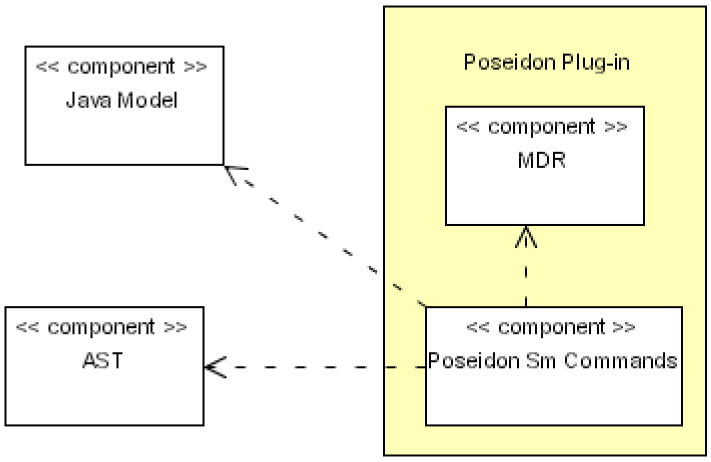


Figure 7-9.5 MDR and Java Model dependencies

The further details on system with implementation are described in the next section.

9.2 Implementation

In this section, chosen implementation details of the integration are described.

9.2.1 Views

As mentioned in the previous section, four views are implemented. They are the Diagram View, Navigation Pane View, Details Pane View and Perspective Pane view. The extension point implemented is "org.eclipse.ui.views". The name of the class implementing the extension should be specified in the plugin descriptor (the plugin.xml [35] file):

```
<extension
   point="org.eclipse.ui.views">
   <category
         name="Poseidon category"
         id="com.gentleware.poseidon.integration.ide.eclipse">
   </category>

<!— the first view NavPane -->
   <view
         name="NavPane View"
         icon="icons/sample.gif"
                                                      category="com.gentleware.poseidon.integration.ide.eclipse"
class="com.gentleware.poseidon.integration.ide.eclipse.views.NavPaneView"
id="com.gentleware.poseidon.integration.ide.eclipse.views.NavPaneView">
   </view>
<!— the second view DetailsPane -->
   <view
         name="DetailsPane View"
         icon="icons/sample.gif"
                                                      category="com.gentleware.poseidon.integration.ide.eclipse"
class="com.gentleware.poseidon.integration.ide.eclipse.views.DetailsPaneView"
id="com.gentleware.poseidon.integration.ide.eclipse.views.DetailsPaneView">
      </view>
    </extension>
```

[35] This file contains the main information for the plugin which is installed to the plugins/ directory of the Eclipse installation.

Table 7-9.1 Extension points' specification in plugin.xml

Table 7-9.1 shows a snippet of the *plugin.xml* configuration file for Poseidon plugin. The definition of two of the implemented views can be seen: The NavPane view and the DetailsPane view. The other two views are specified similarly. The attributes name, icon, category, class and id define the properties of the view.

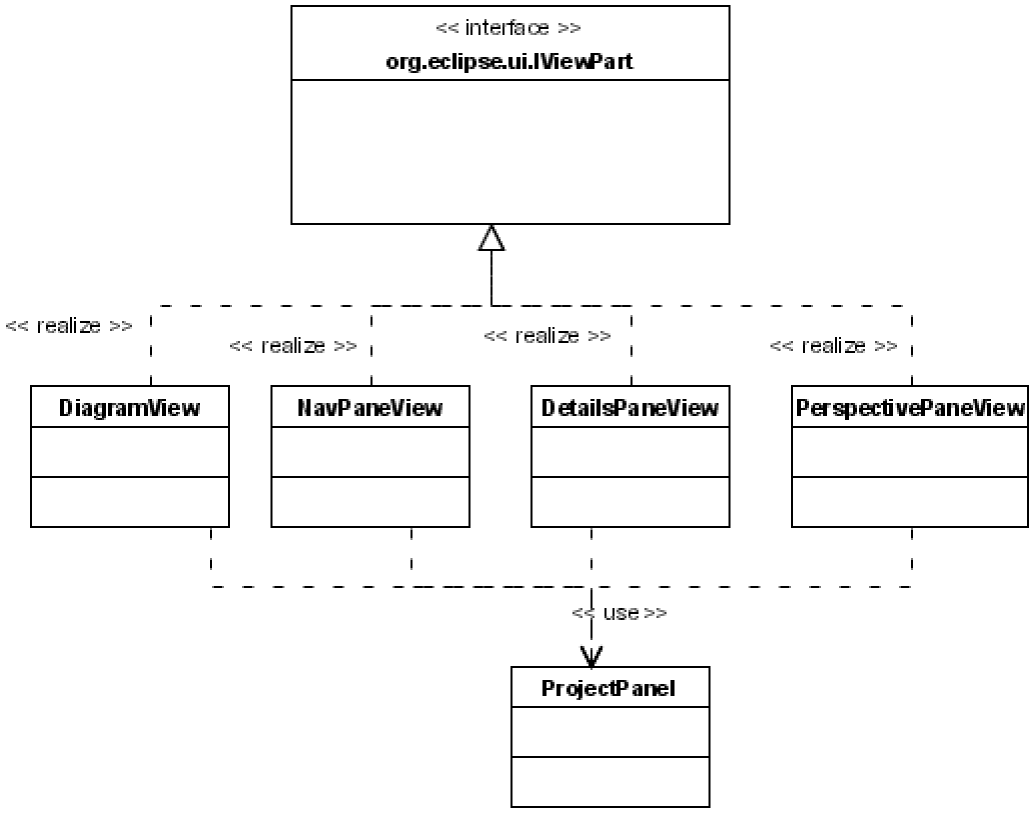


Figure 7-9.6 View extensions' class diagram

Figure 7-9.6 shows the class diagram for the views. Plugins providing new views to Eclipse should implement the interface `org.eclipse.ui.IViewPart`. All of the Poseidon views implement this interface and provide necessary information to the Eclipse Platform. The views also depend on the `ProjectPanel`[36] class because the creation of the views uses instances of Poseidon's project panels. These panels are retrieved by static methods of `ProjectPanel`.

9.2.2 Popup Menus and ActionSets

Besides views, the Poseidon plugin offers popup menus and action sets [37] for better GUI integration. These implemented extensions define new actions for project handling as well as simple Poseidon actions such as *Print Diagrams*, *Save Graphics* etc.

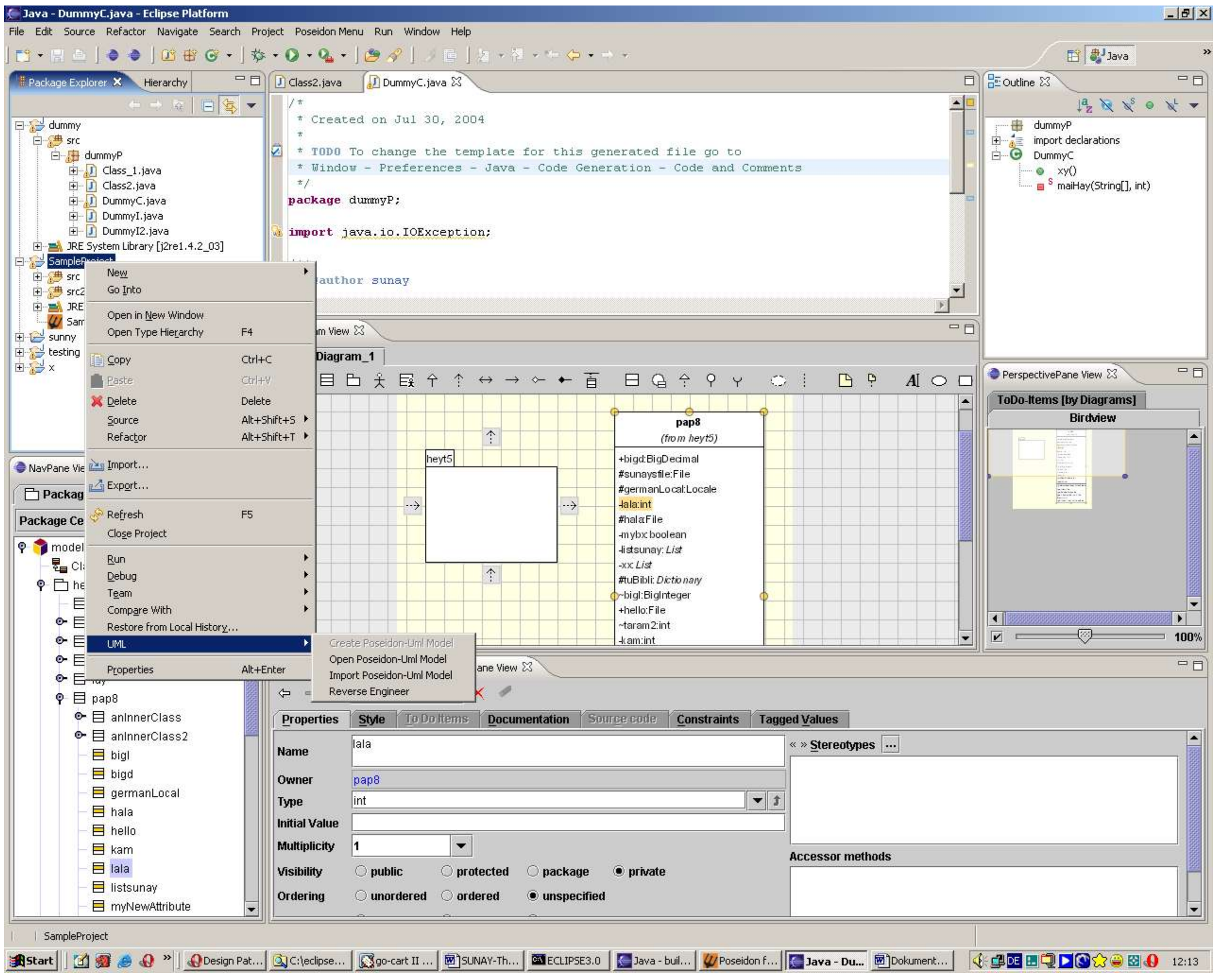
[36] org.argouml.ui.ProjectPanel class

[37] Actionsets in Eclipse are the main top menus

```
<extension
      point="org.eclipse.ui.popupMenus">
    <objectContribution
         objectClass="org.eclipse.jdt.core.IJavaProject"
         nameFilter="*"
         id="com.gentleware.poseidon.integration.ide.eclipse.contribution1">
      <menu
           label="UML"
           path="additions"
           id="com.gentleware.poseidon.integration.ide.eclipse.menu1">
         <separator
              name="group1">
         </separator>
      </menu>
      <action
           label="Reverse Engineer  "
           class="com.gentleware.poseidon.integration.ide.eclipse.popup.actions.ReverseEngineeringAction"
           menubarPath="com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
           enablesFor="1"
           id="com.gentleware.poseidon.integration.ide.eclipse.ReverseEngineeringAction">
      </action>
      <action
           label="Import Poseidon-Uml Model  "
            class="com.gentleware.poseidon.integration.ide.eclipse.popup.actions.ImportPoseidonUmlModelAction
"
           menubarPath="com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
           enablesFor="1"
           id="com.gentleware.poseidon.integration.ide.eclipse.ImportPoseidonUmlModelAction">
      </action>
      <action
           label="Open Poseidon-Uml Model  "
           class="com.gentleware.poseidon.integration.ide.eclipse.popup.actions.OpenPoseidonUmlModelAction"
           menubarPath="com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
           enablesFor="1"
           id="com.gentleware.poseidon.integration.ide.eclipse.OpenPoseidonUmlModelAction">
      </action>
      <action
           label="Create Poseidon-Uml Model  "
            class="com.gentleware.poseidon.integration.ide.eclipse.popup.actions.CreatePoseidonUmlModelAction
"
           menubarPath="com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
           enablesFor="1"
           id="com.gentleware.poseidon.integration.ide.eclipse.CreatePoseidonUmlModelAction">
      </action>
    </objectContribution>
  </extension>
```

Table 7-9.2 Popup Menus Definition in plugin.xml

Figure 7-9.7 shows the popup menu added for Java projects in the workspace (“UML”). It contains four menu items for creating, opening and importing Poseidon UML models for the selected Java project as well as reverse engineering them.



```
/*
 * Created on Jul 30, 2004
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package dummyP;

import java.io.IOException;

author sunay
```

Name: lala
Owner: pap8
Type: int
Initial Value:
Multiplicity: 1
Visibility: ( ) public ( ) protected ( ) package [✓ private]
Ordering: ( ) unordered ( ) ordered [✓ unspecified]

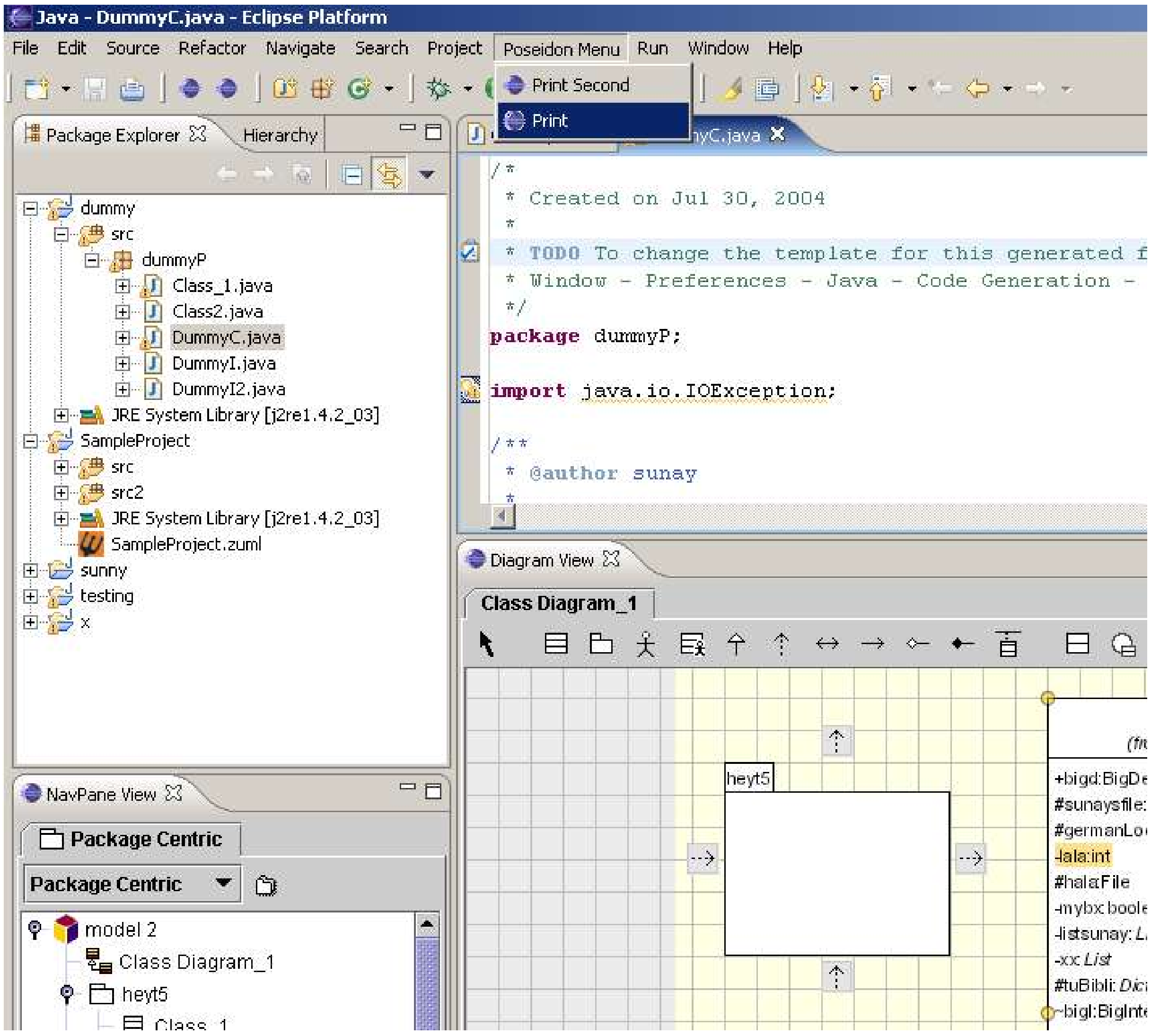
Figure 7-9.7 Added Popup Menu for Java projects

Table 7-9.2 shows the definitions for the new popup menu extension points in the plugin.xml. The object contribution element specifies that the popup menu is only available for objects of the type stated in the objectClass attribute. In our case, the value is objectClass="org.eclipse.jdt.core.IJavaProject", which defines the popup menu for the Java projects in the Eclipse platform. The element <action> defines new menu items in the popup menu for the object with type `IJavaProject`.

```
<extension
      point="org.eclipse.ui.actionSets">
    <actionSet
        label="Poseidon Action Set"
        visible="true"
        id="com.gentleware.poseidon.integration.ide.eclipse.actionSet">
      <menu
          label="Poseidon &amp;Menu"
          id="poseidonMenu">
        <separator name="poseidonGroup"/>
      </menu>
      <action
          toolbarPath="poseidonGroup"
          label="&amp;Print"
          class="com.gentleware.poseidon.integration.ide.eclipse.actions.PrintAction"
          tooltip="Print Current Poseidon Diagram"
          icon="icons/sample.gif"
          menubarPath="poseidonMenu/poseidonGroup"
          id="com.gentleware.poseidon.integration.ide.eclipse.actions.PrintAction"/>
```

Table 7-9.3 ActionSet Extension Point Definition in plugin.xml

Figure 7-9.8 shows the **Poseidon Menu** in the Eclipse main menu.



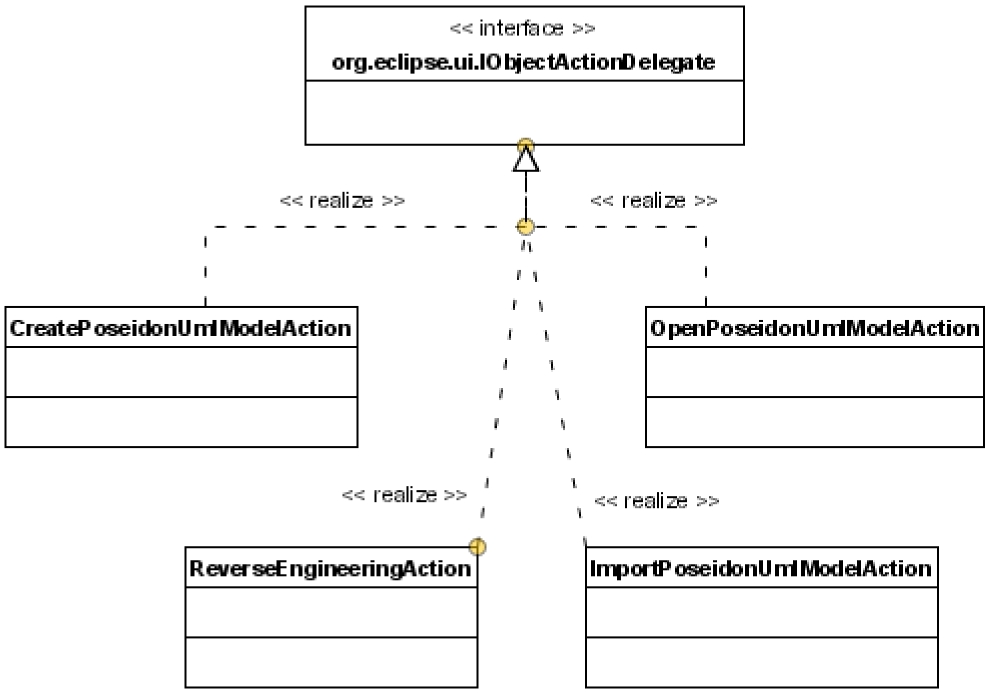
```
/*
 * Created on Jul 30, 2004
 *
 * TODO To change the template for this generated f
 * Window - Preferences - Java - Code Generation -
 */
package dummyP;

import java.io.IOException;

/**
 * @author sunay
 *
```

Figure 7-9.8 Poseidon Menu with Simple Actions

Table 7-9.3 shows the definition of the actionsets in the plugin.xml file of the Poseidon plugin. The Print action can be called by clicking the Print Menu item in the Poseidon Menu of the Eclipse Platform which is provided by the plugin.



<< interface >>
org.eclipse.ui.IObjectActionDelegate
<< realize >>
<< realize >>
CreatePoseidonUmlModelAction
OpenPoseidonUmlModelAction
<< realize >>
<< realize >>
ReverseEngineeringAction
ImportPoseidonUmlModelAction

Figure 7-9.9 Class Diagram showing Popup Menu extensions for IJavaProject instances

The actionset defined by the plugin simply calls the main Poseidon actions such as *Print* or *Save graphics*. For the time being, the actions in the menu are not directly UML model related. They are independent of the synchronization procedure.

Figure 7-9.9 shows the UML class diagram for the popup menu extensions provided by the Poseidon plugin. `ReverseEngineeringAction` and `ImportPoseidonUmlModel-Action` will be described soon. They are simple actions where the first one only reverse engineers the selected Java project, and the second one imports a Poseidon UML project into the workspace without providing synchronization.

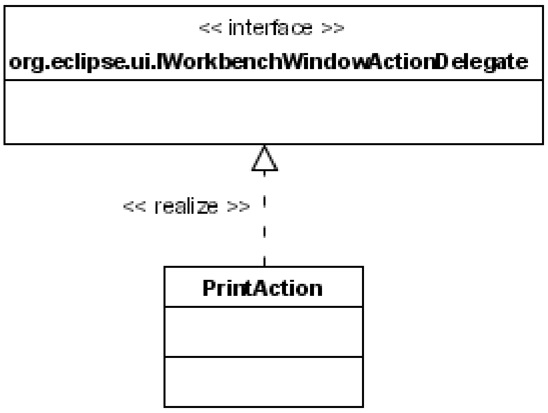


Figure 7-9.10 Class diagram showing ActionSet extensions

Finally, Figure 7-9.10 shows the class diagram for actionset extensions. Only one exemplary actionset extension is provided. The print action which has no direct relation to the synchronization and roundtrip engineering is provided by this extension.

org.eclipse.ui.IWorkbenchWindowActionDelegate is implemented by PrintAction. Executing this extension simply calls the print action from the Poseidon standalone version.

9.2.3 Project Handling

Project handling is achieved through the actions defined in the popup menu for the Java projects in the workspace. There are two main actions implemented for this, the "Create Poseidon UML Model" and the "Open Poseidon UML Model" actions.

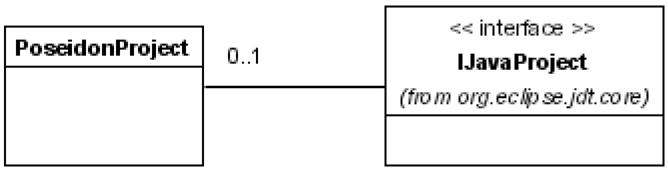


Figure 7-9.11 Poseidon and Java Project relationships

The execution of actions conforms to the class diagram seen in class diagram of Figure 7-9.11.[38] The class diagram shows that any Poseidon project opened (as part of synchronization) in Eclipse has an association with a project in Eclipse with Java nature. If the UML model for a Java project has not yet been created, the corresponding Poseidon project does not exist. Therefore the multiplicity for the PoseidonProject end of the association is not *1*, but *0..1*.

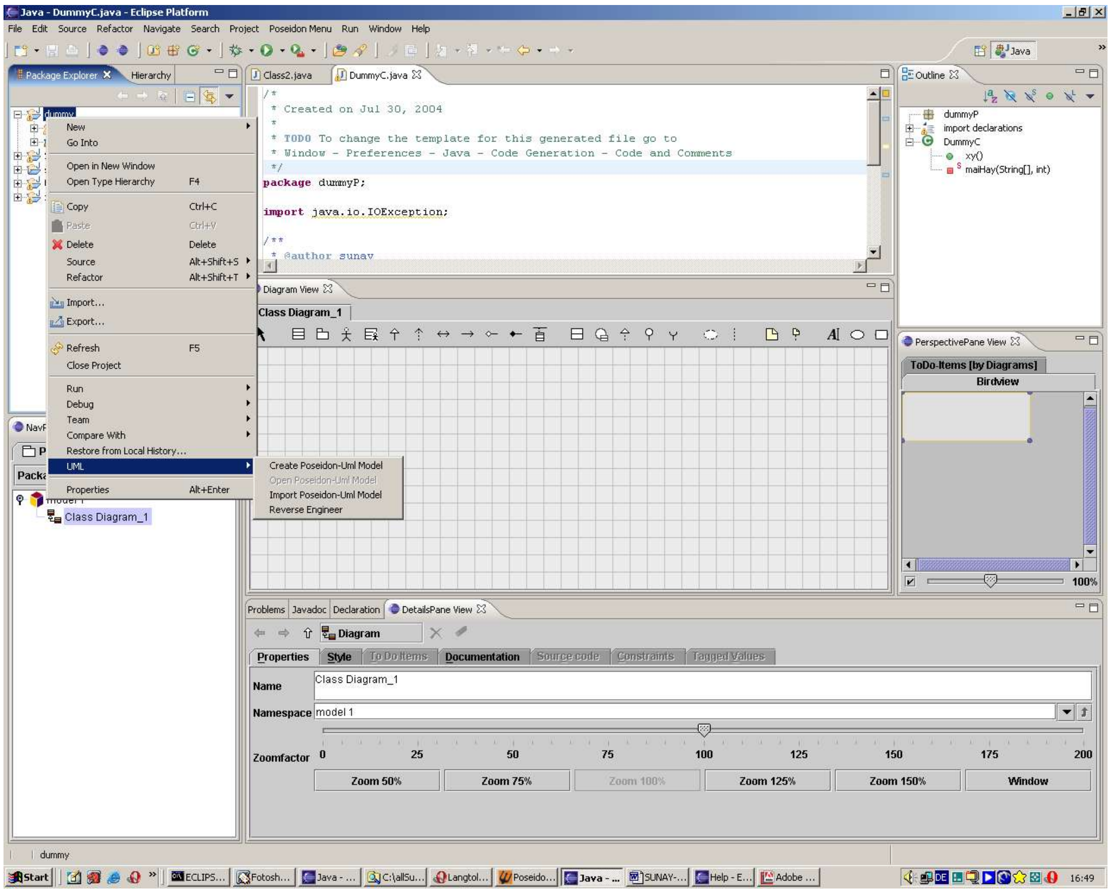
9.2.3.1 Create Poseidon UML Model action

This action is implemented by the class CreatePoseidonUmlModelAction. If for the selected Java project the UML model does not exist (see 9.1.3.1 for more details), this action is enabled at the startup of the Eclipse. The popup menu for Java projects can be seen in Figure 7-9.12. org.eclipse.ui.IObjectActionDelegate interface is implemented by these actions. When this action is executed, reverse engineering for all the project source code is done and then the UML model obtained by reverse engineering is saved with the same name as the Java Project under the Java Project root directory. This action can not be executed for a second time unless the existing UML Model (project) is manually deleted.

9.2.3.2 Open Poseidon UML Model action

This action can be executed only when the UML model for the selected project already exists. After executing the action, the UML model is opened and sets the current project in Poseidon views. An example execution of the action can be seen in Figure 7-9.13. The created UML model elements can be seen in the *NavPane* view and the respective details and overview in the views *Diagram*, *DetailsPane* and *PerspectivePane*.

[38] The multiplicity for association ends which are not shown in the Figure is 1.



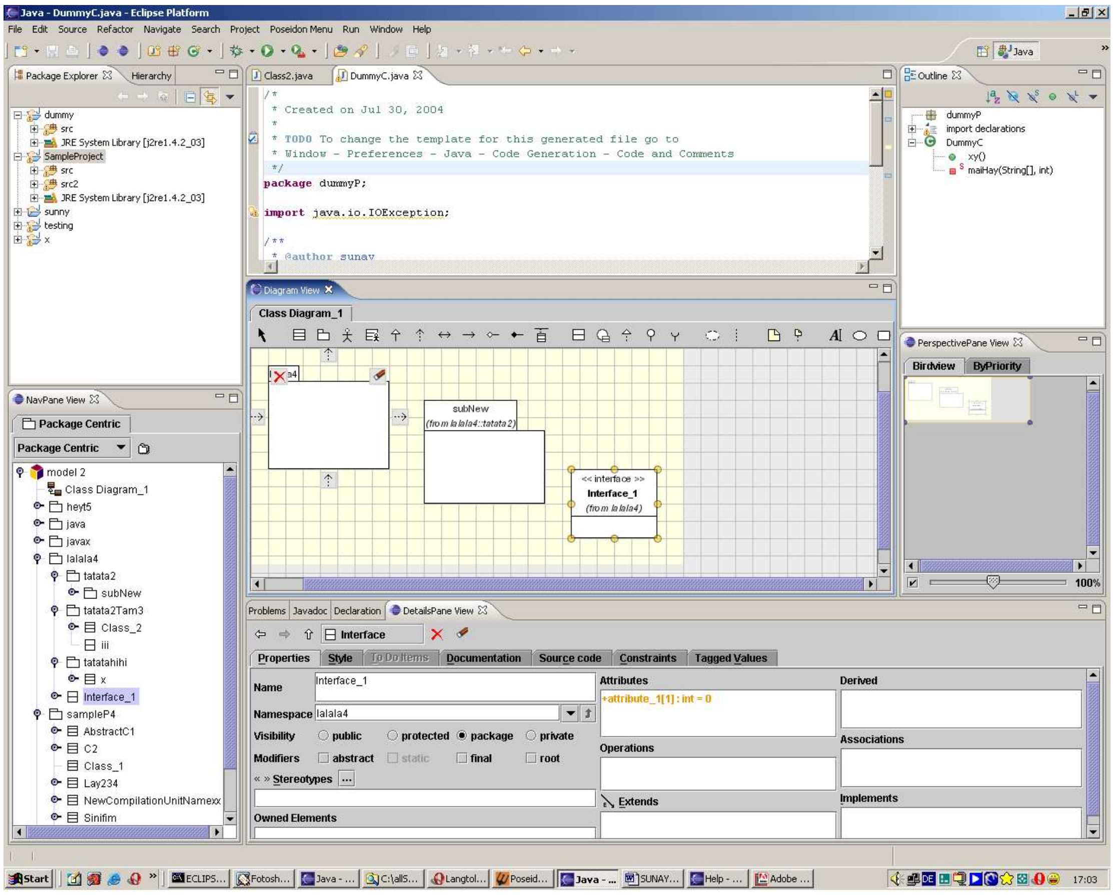
```
/*
 * Created on Jul 30, 2004
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package dummyP;

import java.io.IOException;

/**
 * @author sunay
```

Name: Class Diagram_1
Namespace: model 1

Figure 7-9.12 Popup menu for Java Projects in workspace



```
/*
 * Created on Jul 30, 2004
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package dummyP;

import java.io.IOException;

/**
 * @author sunay
```

Name: Interface_1
Namespace: lalala4
Visibility: public protected [✓ package] private
Modifiers: abstract static final root
« » Stereotypes
Owned Elements
Attributes: +attribute_1[1] : int = 0
Operations
Extends
Derived
Associations
Implements

Figure 7-9.13 Opened UML Model for Selected Java Project

9.2.4 Design of the Map used in Synchronization Implementation

To achieve synchronization of the UML model and Java source code, a map holding UML model element and respective Java element[39] is used. The map contains as key the ids for UML model elements (which are generated by Poseidon for easy accessing). An entry in the map has the form: (*Id for UML Model element, Java element reference*).

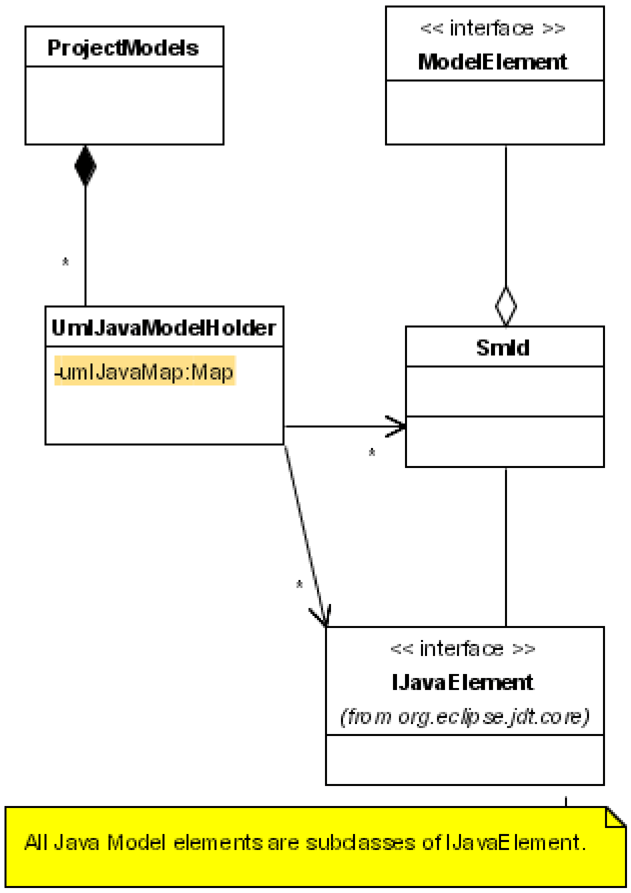


Figure 7-9.14 UML-Java Map Design

Figure 7-9.14 above shows the high level design of UML-Java map used in the implementation. `SmId` class represents the ids used by Poseidon semantic model elements.[40] `ProjectModels` contains the mapping for the opened UML and Java projects. Whenever UML model for a Java project is created or opened, the map is filled with its entries. `UMLJavaModelHolder` (for simplicity only relevant attribute is shown in the diagram.) contains the mapping for one UML project (respectively Java project). The map `umlJavaMap` contains the entries described above. The associations shown represent the map entries.

Individual Entries in the Map

- `UMLPackage` (from the package `org.omg.uml.modelmanagement`) and `IPackageFragment`

[39] Any reference to Java element implies Java elements in context of Java Model in JDT API.
[40] Diagram Interchange model elements' id has type DiId

UMLPackage instances map to IPackageFragment instances in the map.

- UMLClass or Interface (from org.omg.uml.foundation.core) and IType

UMLClass instances or Interface instances map to IType instances in the map.

- Operation (from org.omg.uml.foundation.core) and IMethod

Operation instances map to IMethod instances in the map.

- Attribute (from org.omg.uml.foundation.core) and IField

Attribute instances map to IField instances in the map.

Realization (implementation) and generalization relationships are not inserted to the map as separate entries although they are independent model elements in UML models, but they are represented and taken care of in the synchronization. The "implements" and "extends" keywords in the source code represents the existence of these relationships which can be checked programmatically via Java Model and AST API of Eclipse. Visibilities as well as modifiers are part of the definition of interfaces, classes and packages in UML and Java. So, they are not represented as separate entries in the map although they are taken care of during synchronization.

As associations, aggregations and compositions are not considered during synchronization, currently they are not inserted into the map.

9.2.5 Synchronization

In this section the portion of the implemented synchronization process is summarized. The synchronization events can be classified in two main groups:

- Poseidon notifications triggering updates in Eclipse.
- Eclipse notifications triggering updates in Poseidon.

The first group of changes implies the changes of UML model which have to update the Java source code in the Eclipse platform. An example change is *Creation of a new UML class*. Whenever this happens, to keep the opened UML model and the Java source synchronized, the corresponding Java source file should be created.

Figures Figure 7-9.15, Figure 7-9.16 and Figure 7-9.17 represent the main parts of the UML metamodel used for creating UML models.[41] Figure 7-9.15 is a simplified class diagram representing the core part. The metamodel classes for representing classes, namespaces, operations, methods, attributes, etc., can be seen in this diagram. The inheritance hierarchy as well as relationships between them can be seen easily.

Figure 7-9.16 summarizes the portion of the UML metamodel representing the relationships between model element instances as generalization, association, etc. The parts not used in the implementation for managing UML models are not shown for simplicity.

Finally Figure 7-9.17 shows a simplified class diagram representing the part of the UML metamodel for classifier hierarchy. Class, Interface and DataType are classifiers and are subtype of Classifier as seen in the figure.

[41] Basically UML models contained in class diagrams

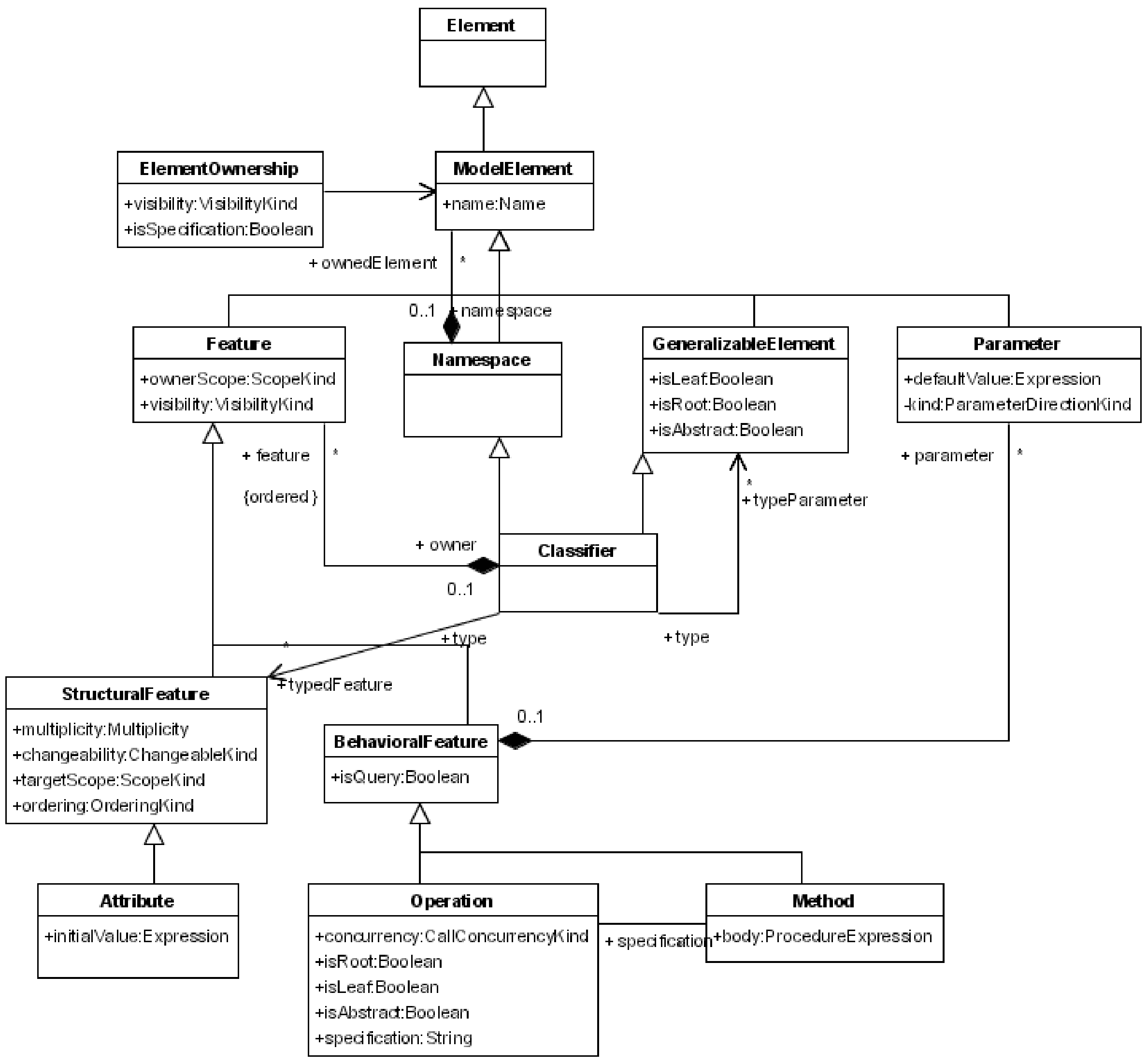


Figure 7-9.15 UML Metamodel- Backbone of the Core package

9.2.5.1 Poseidon to Eclipse Changes

9.2.5.1.1 Creation of new UML model elements

Creation of the following model elements results in the source code being updated as well as in performing other related changes in the Eclipse Platform JDT.

- Create Class
- Create Interface
- Create Attribute
- Create Operation

Creation of other model elements such as associations, dependencies etc. in the class diagrams does not yet result in Eclipse updates.

9.2.5.1.1.1 Create UML Class

A new UML class is created. The namespace for the created class contains the package where the class resides. The corresponding Java package in the Java Model is found and a new Java class is created belonging to this package. The code generated for the UML model is the source code for the created class. This generated code is taken from the *Code Generation* component of Poseidon.

The UML models are instances of UML metamodels. To create a new UML class, an instance of Class seen in Figure 7-9.17 is created. The namespace where this class resides can be retrieved by querying the `getNamespace`() on the class instance created. The namespace and class information are later on used at the creation of source code of the respective Java class. Properties of the class such as visibility and some of the modifiers that exist in the UML metamodel are retrieved directly. But for modifiers such as "`transient`" and "`volatile`" that are not represented in the UML metamodel, tagged values for the newly created class instance are inserted [42] instead to the UML model. The `transient` and `volatile` modifiers to create the Java class are retrieved from these tagged values.

The descriptions for UML model handling for the next subsections are not given. It is assumed that the figures showing the UML metamodel give an idea of the handling.

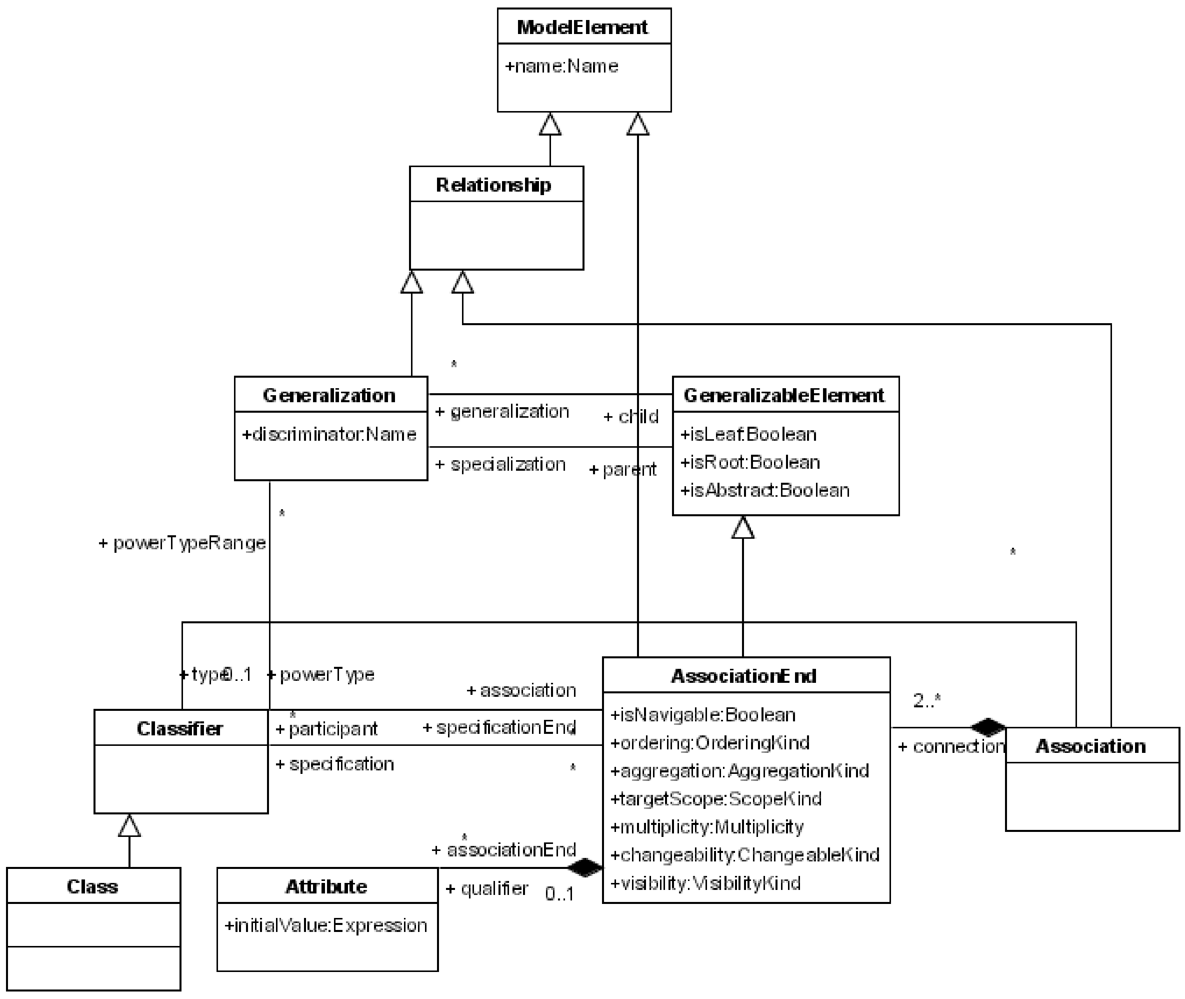


Figure 7-9.16 Core Package- Relationships

[42] These tagged values for representing the modifiers are part of Java profile in Poseidon.

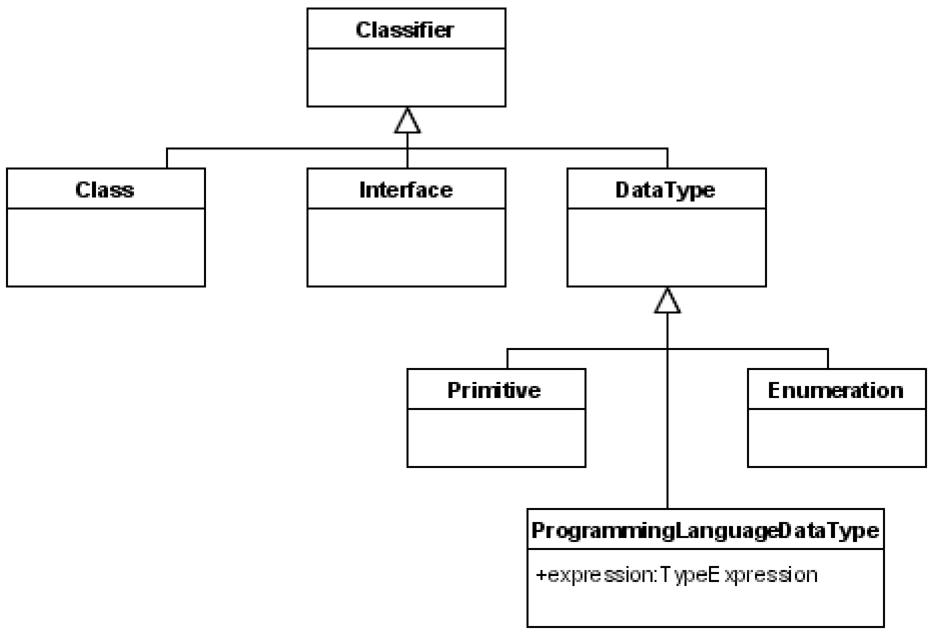


Figure 7-9.17 UML Metamodel- Classifiers in Core Package

9.2.5.1.1.2 Create UML Interface

A new UML interface is created. The namespace for the created interface contains the package where the class resides. The corresponding Java Package in the Java Model is found and a new Java interface is created belonging to this package. The generation of code is similar to creating a new UML class.

9.2.5.1.1.3 Create UML Attribute

A new UML attribute is created. The class or interface this attribute belongs to is found in the Java model. The corresponding class or interface is updated by adding the generated code for the new UML attribute.

9.2.5.1.1.4 Create UML Operation

A new UML operation is created. The class or interface this operation belongs to is found in the Java model and the owner class's source code is updated by adding the generated code for the created operation.

9.2.5.1.2 Deletion of UML model elements

Deletion of any UML model element triggers changes in the Eclipse Platform which are individually represented in the UML-Java map implemented, and deletes the corresponding piece of code from the respective files.

9.2.5.1.3 Modification of existing UML model elements

Implementation for some basic modification of existing UML model elements is done. They can be grouped as following:

- Renaming model elements
- Changing the visibilities
- Changing the leaf property for the model element (**final** or not)
- Changing the concurrency property of the model element (**synchronized** or not for Java)
- Changing the "**abstract**" property
- Changing the "**transient**" property
- Changing the "**volatile**" property
- Changing the owner scope for the model element (**static** or instance)

During implementation of the various kinds of modifications on UML model elements, Abstract Syntax Trees and Java Model API of JDT are often used. Especially for changing visibilities, `abstract`, `transient`, `volatile`, `final`, `synchronized` keywords easily in the Java source code, AST is used. The AST representation of the specified source files is obtained and the specified piece of source code is found. Java Model in Eclipse provides very useful methods such as getting the range of source code for Java methods, fields, types etc. For the search to be faster only the corresponding range of source code needs to be searched and the keywords are changed according to the change done on the UML model element. The text buffer for the class is updated automatically by Eclipse.

During implementation, the main problem was at the "*rename*" of the Java elements. Whenever a rename on a UML model element executes, the corresponding Java element needs to be renamed, too. The mapping of a Java element for a UML model element or vice versa is kept in a map. This map is created after the user executes "Create Poseidon UML Model" action. The map has entries like "UML model element reference"→ "Java element reference". For a UML class, an entry having the mapping Java class is added to the map as mentioned. Respectively, for UML attribute and operations, Java fields and methods are added. Interfaces and packages are also added similarly. The difficulty is at this point:

Whenever the name of a Java element from the Eclipse Java model changes, Eclipse creates a new reference for the Java element whose name gets changed because the Java elements are only adapters representing the real resources. The previous reference which was already in the map gets invalid, i.e., "exists()==false". This brings the need that the map entry containing this invalid Java element should be found and it should be deleted and the new entry with the new reference for the Java element and the same UML model element should be added. The situation is even worse if the Java element that got invalid also has children. In this case also the children become invalid. So the algorithm for updating the map for invalid Java element references should be applied again for the children.

9.2.5.2 Eclipse to Poseidon Changes

In this section, the source code changes that are also propogating to the UML model are presented. To keep the state of the UML model and the source code in sync, changes done in the source code should trigger and execute the changes on the UML model.

- Create attribute
- Create Method
- Create Interface
- Create Class
- Create Method Parameter

- Create Generalization
- Create Realization
- Delete UML Model Elements

The implemented modifications are only a few representative ones. The changes on the source code are retrieved after the analysis of Java element listener. A very major problem encountered during implementation of the modifications is that the Java element changes that are notified by Eclipse are not fine-grained enough. For example, if a method body changes, the element listener only notifies that the owner class content is changed without mentioning the method whose body is changed. Even if the return type for a method changes, again the notified change is "CONTENT CHANGE" in the owner class, which is not detailed enough.

The implemented modifications (such as creation of new Java elements, generalization, realization and deletion of Java elements) are notified in details that are accurate enough by the Eclipse platform.

A solution for this problem of inaccurate level of details of changes' notification in the source code consists in to analyzing the buffer with changes that Eclipse notifies. But this analysis would be very problematic, and is out of the scope of this work. The analysis would need to get any character change in the Java editor buffer. During getting new characters or deletion of characters (or group of characters), for any newly notified event (change on the buffer), there should be a very detailed analysis about what these accumulated buffer changes could mean. For example, the user could be changing the visibility. For deletion of characters in "public", there will be at least 5 notifications done (deletion of "ublic" in case "p" does not get deleted). Then for addition of "protected" keyword, there will be at least 8 notifications (addition of characters "rotected" in case "p" is not deleted). All those changes done are only for changing a simple visibility. For analyzing all changes that can occur in Java editors, a comprehensive algorithm would be necessary.

9.2.5.3 Associations

Associations are, as reviewed in many papers, difficult to map to Java code and distinguish in the Java code from the ordinary UML attributes. Especially obtaining associations in reverse engineering process is difficult. Most of the reverse engineering tools do not create any association relationships. Some commercial products that provide real-time synchronization save in the source code details about the UML model as comments. Considering the limited time in this thesis work, synchronization of associations is omitted. They are planned for future work.

9.2.6 CommandFactories in Poseidon

In Poseidon, every interaction of the user happens in *commands*. The command factories for semantic model of UML as well as diagram interchange contain all the interactions a user can do. *Command factories are singletons in Poseidon. Every Poseidon application started owns a single instance of command factories*. The command factories and command behavior in Poseidon is implemented as a *Command Pattern [66]* [43].

The command framework eases the implementation of *UNDO* operations.

[43] Command Pattern Definition: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

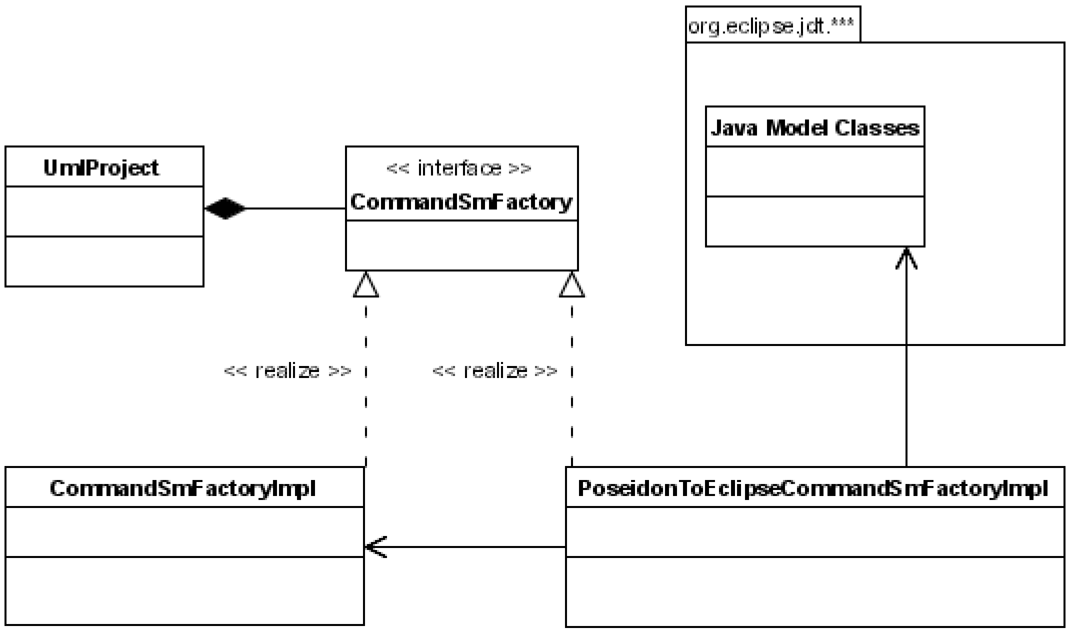


Figure 7-9.18 Command Factories Class Diagram

For the standalone Poseidon, `CommandSmFactory` interface contains commands that modify semantic model of the UML model, ignoring diagram interchange.[44] The commands for diagram interchange do not change the UML semantic model and they are listed in `CommandDiFactory` interface.

When Poseidon is integrated to Eclipse, the changes on the UML model should also update the corresponding Java source code in Eclipse JDT. Because Poseidon core commands handles only UML models, the existing command factories need to be replaced.[45] `PoseidonToEclipseCommandSmFactoryImpl` implements `CommandSmFactory`.This factory replaces the core command factory, `CommandSmFactoryImpl`. It manages UML models as well as Java source code changes that should be executed. For example, the command "*Create New Class*" called in NavPane view, first creates a new UML class, and then creates the corresponding Java class in the Eclipse workspace by using Eclipse JDT API. `PoseidonToEclipseCommandSmFactoryImpl`'s implementa-tion conforms to the ***Decorator pattern*** [66][46]. Figure 7-9.19 shows the roles of the classes wrt the Decorator pattern.

[44] Poseidon for UML uses Diagram Interchange for describing UML diagrams. Actually, Diagram Interchange is part of UML2.0 specifications.

[45] Another solution is listening for changes done on UML model, but analysis and change notification is too complex in MDR component which is used in Gentleware.

[46] Decorator pattern definition: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

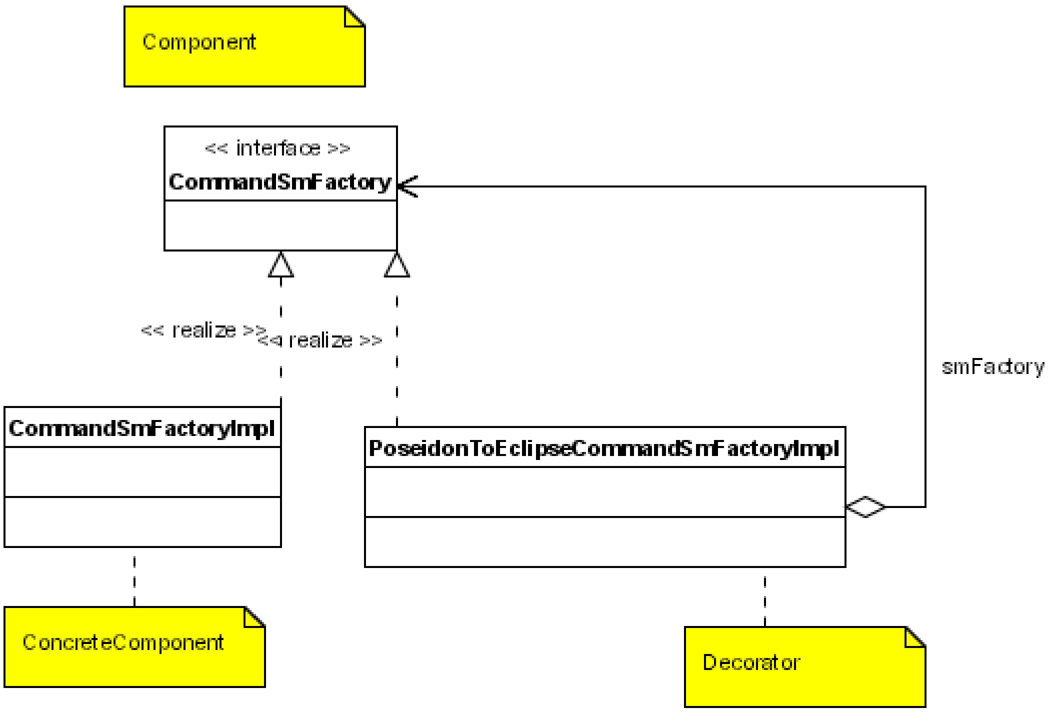


Figure 7-9.19 Decorator Pattern roles for command implementation

The synchronization events implemented (see 9.2.5.1) currently only changes the UML semantic model. It does not touch the diagram interchange. So, command factories for the diagram interchange handling are not replaced.

9.2.7 UML Model Management by MDR

The modifications on the UML models in Poseidon are done by using the component MDR [67] which was mentioned before.

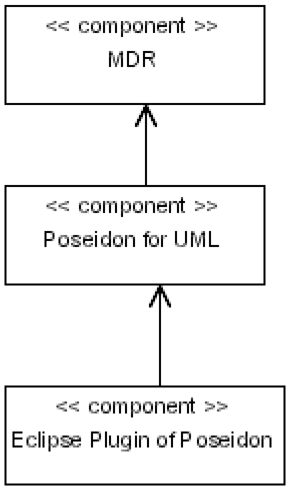


Figure 7-9.20 MDR Dependency of Poseidon for UML

The implementation of commands in Poseidon is done via extensive use of MDR API. The learning curve is a little steep because as a previous UML user, I did not have much information on UML metamodel and MDR implementation.

The commands found in `PoseidonToEclipseCommandSmFactoryImpl` are MDR dependent because they work on UML models. The dependency for the command "*Create Uml Class*" is given as an example. The dependency for other implemented commands is similar.

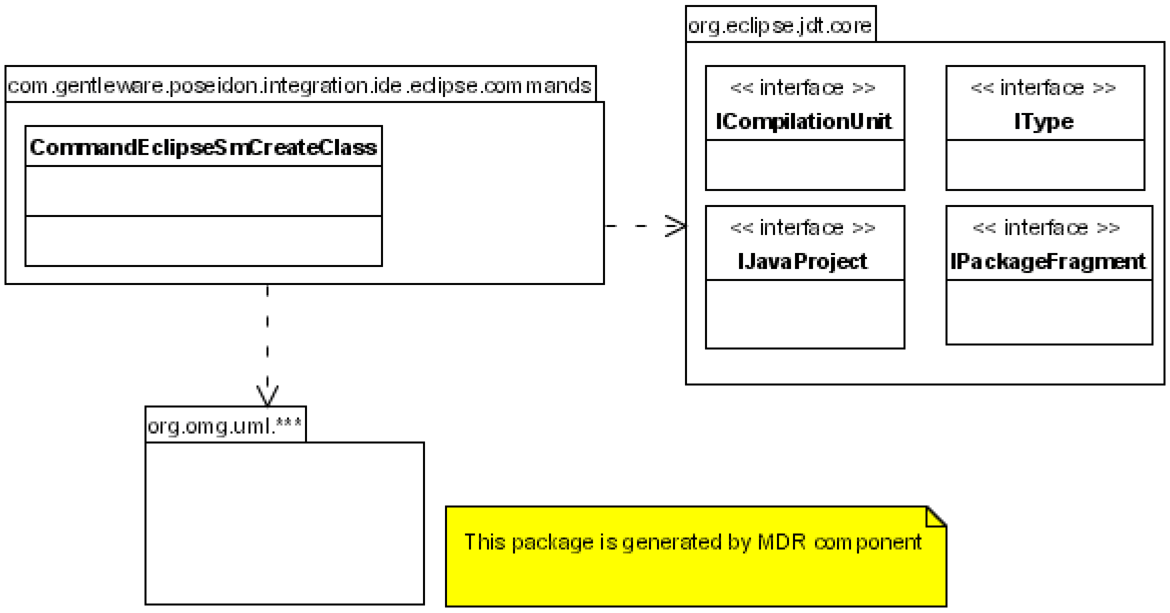


Figure 7-9.21 MDR and JDT Dependency of Commands

Here, in Figure 7-9.21, the class diagram for dependencies of commands that manage UML model and Java source code can be seen.

9.2.8 Main Problems Encountered During Implementation

Although some of the problems encountered are already mentioned during describing implementation details, this section tries to give an overall list of the main problems encountered. The problems (those that are not forgotten) are listed as follows:

- ***Threading problems which are caused by Swing-SWT Integration***

One of the motivations for this thesis work was Swing and SWT incompatibility. It was not possible to integrate plugins with their GUI implemented with Swing library. I found out that Eclipse 3.0 release plan is considering enabling Swing integration to Eclipse GUI. Eclipse 3.0 was released on June 25th 2004 and till then I worked with the milestones versions of Eclipse. The milestones versions are normally buggy versions which are not ready for releasing. I suffered from these bugs.

As the Eclipse 3.0 is released, I could finally work on Eclipse without worrying for migrating to new versions. The implementation for integrating Swing panels views of Poseidon into Eclipse as views already started then. The integration for Swing GUI is not as compatible and supported as SWT GUI in plugins as expected.

The Swing and SWT integration poses problems whose solutions are not documented in Eclipse 3.0 documentation. The event dispatchers of Swing and SWT are different and they are handled in separate GUI threads. Any update to Swing components should either be done in Swing event threads and Swing components or they should be called as in the following code snippet:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      Services.logInfo("Saving the project" );
      PoseidonProjectConnector.saveProject(name + ".zuml");
    }
});
```

This code snippet is called from an Eclipse action implemented for Poseidon plugin. As the Eclipse action implements a SWT-based interface, the threads that update Swing GUI should be called in `SwingUtilities.invokeLater` method. The same holds for SWT updates that need to be done in Swing threads. In this case, one should call the specific functionality as follows:

```
pluginDisp.asyncExec (new Runnable () {
    public void run () {
          if(!PoseidonStarter.isPoseidonStarted() )
               PoseidonStarter.init(true);
                    //update the views...
               Util.showPoseidonViews();

        }
});
```

`asynchExec` method of `Display` class in Eclipse API should be used to implement the necessary functionality. If the methods updating GUI with the other GUI library is not used as shown above, the whole Eclipse platform gets ***deadlocked***. Finding out how to solve these deadlocks was time consuming and there were no documentation as Eclipse 3.0 was relatively new.

- ***Work on Eclipse 3.0 Milestones whose API changed often until release. Eclipse versions before 3.0 do not allow integration of Swing applications in SWT.***

Eclipse 3.0 is, as mentioned, released after 3 months of this thesis work started. Up to the final release, I needed to work with milestones *3.0 M8 and 3.0 M9*. The API of even 3.0 M8 and 3.0 M9 is not compatible with each other. Any implementation done in 3.0 M8 was not even compiling on 3.0 M9. This too often change of Eclipse API is really annoying although it was for milestones[47].

- ***AST Handling and Documentation***

Documentation of AST of Eclipse Java Model API is very brief. The developers need to explore the Eclipse internal implementation and the usage of AST to understand. AST modification of the Java compilation units was only done whenever it was really necessary

[47] Unfortunately I had to live with milestones as Eclipse 2.X versions did not allow integration of Swing components in Eclipse Workbench.

and not available through Eclipse Java Model API.[48] For example, for changing the visibilities, modifiers etc., the Java model provides no methods. Either one has to manually update the buffer content matching to visibilities or build the AST tree of the Java source file and update the visibilities accordingly. So, such functionality is implemented with AST API[49] of JDT.

- ***Java element listener for Java elements in the Java Model***

The changes on the source code done in Java editors of Eclipse are notified through `IElement-ChangedListener` and `IBufferChangedListener`. As described in Section 7.4.1, the element listener notifies the changes done on the Java elements. Using and analyzing the changes through `IBufferChangedListener` in text buffers is too complicated for analyzing. `IElementChangedListener` notifies changes again real-time. Although the assumption was that the changes are notified correctly and detailed enough, the analysis made on this listener showed that it does not provide sufficient data. A change on the return type for a method is notified as *"change in the content of the type owning the method*" and a change on the element name is notified as "*a new Java element is added*" which is not correct.

Change on the inheritance hierarchy or implementation hierarchy is also notified without enough details necessary for real-time roundtripping. As a result, using this listener which we were dependent on is very problem-posing and unfortunately not enough for updating UML model based on source code changes. The only reasonable solution to this problem can be updating the UML model elements manually after source code changes.

- ***MDR classloader***

At the beginning of the thesis, the first goal was to integrate the Swing based GUI of Poseidon into SWT based Workbench of Eclipse. With the views extensions, this goal was achieved. Another goal was as expected to have Poseidon run as it runs in standalone version. [50] As previously stated Poseidon depends on and uses MDR for UML data management. By default without doing changes, MDR did not work in Eclipse workspace. MDR is highly class loader dependent to manage metamodels and models. Finding classes is done by searching in the class loaders. MDR was not able to find the classes necessary for UML data management in Eclipse environment. A deep search in the newsgroups and the Internet led to the conclusion that MDR was not searching the classes in the Eclipse class loader which is the main class loader in the Eclipse environment. As the Eclipse class loader was responsible for loading any plugin libraries, MDR has to look up in this class loader. A solution was found[51] and adapted to Poseidon plugin.

- ***Refactor "Rename" or simple "Rename"***

A decision to be made was whether to use refactoring operations or simple changes on Java source code. When the user makes changes on the UML model, the synchronization triggers corresponding changes on the source code. For instance, if the user changes the name of a class in the UML model, the desired behavior of the user is to rename the corresponding Java class so that all the references also get updated. This indirectly means that a refactoring is

[48] Classes and interfaces in package `org.eclipse.jdt.core`
[49] Classes and interfaces in package `org.eclipse.jdt.core.dom`
[50] Without any synchronization
[51] Setting the context loader of the current thread to the Eclipse class loader by `Thread.currentThread().setContextClassLoader (`*`theEclipseClassLoader`*`);`

necessary. Refactoring in Eclipse workspace is too heavy and it can not be undone. Especially considering the heavy burden on the synchronization task, the simple modification approach is adopted. Therefore, the rename operation of a UML class triggers only a change on the name of the Java class without touching the references of the updated class. As it is clear to see, this was a general question to answer. It is not only to decide whether simple rename or refactor rename. In some cases like visibility changes, the refactoring would be posing extra problems other than performance. If the visibility of an attribute is changed to private, getters and setters should have to be generated and direct accesses to the attribute should be changed with getters and the setters of the attribute.

- ***Testing is very much time consuming, Poseidon start and Eclipse start take altogether long time***

Running Eclipse Platform even without extra plugins needs around 100 MegaBytes of memory. It is a huge application. Poseidon is also using a big space of the memory, by average around 60 Megabytes. For testing the operations implemented and finding out bugs, restart of both applications was necessary in many cases. This is a very time-consuming process for implementation.

- ***Roundtripping between Java source code and UML models is still not well and optimally***

Conceptually roundtripping is still not well defined and clarified for UML models and Java source code. Representation of associations, composition and aggregation for many UML tools at roundtrip-ping is very different. Together saves necessary information in the source code, Omondo writes all of the UML model information in the source code as Xdoclet tags etc. To support roundtripping, a way of keeping all details of UML models and its corresponding code is necessary and the best place to save these details is the source code itself.

- ***Support by many tools as the mapping is very problematic for some UML constructs***

Mapping association, aggregation and composition is still not same for the tools and the mappings are mostly introducing loss of information or the UML constructs conceptual meaning is not represented to the source code.

- ***Eclipse Plugin development and its API as well as Eclipse configuration problems had to be solved by me, team members in Gentleware and my supervisor uses IntelliJ Idea and I had no help on Eclipse environment.***

This fact is an unpleasant one, but it holds. It led to loss of time in some cases, especially at technical problems or configuration problems of Eclipse or MDR problem I mentioned.

10

11 Concluding Remarks

In this chapter, I will provide an overview of the work done in this thesis. The implementation was only a selected portion of the functionality possible. At the end of the chapter, the points where future work can be done and the work can be extended are mentioned.

11.1 Summary

The main motivation behind this thesis work was to integrate Poseidon with the Eclipse Platform. Poseidon's GUI is implemented with Swing library and Eclipse is implemented completely independent of Swing library. The Eclipse Workbench is implemented with the SWT library IBM designed and created as Eclipse was in its initial days. These days AWT of Java was so slow and buggy. Therefore IBM introduced a completely new GUI library, SWT, to implement their platform Eclipse.

Up to Eclipse 3.0 release, the plugins could not seamlessly integrate with Eclipse Platform unless they used SWT as GUI. The very important and main problem is that there are so many big applications, projects whose GUI was implemented with Swing such as Poseidon. To enable them to integrate with Eclipse, a very time-intensive programming would be necessary to implement these systems with SWT library. Many companies simply avoid this, or had to avoid this because of the cost and extra work posed. Since Eclipse 3.0, plugins that use Swing GUI can integrate seamlessly with Eclipse Platform although the integration is not as easy and as good supported as SWT itself. Luckily, Eclipse 3.0 enables it. The Swing/SWT incompatibility of previous years was the first explored problem in context of this thesis. After a review of the Eclipse Platform and its documentation, the good news about the Eclipse 3.0 plan of Swing/SWT integration is found out and used for integrating the GUI.

A big amount of time during this thesis was devoted to finding out whether EMF is a must for UML tools or plugins that aim to integrate to the Eclipse Platform. Eclipse makes promotes EMF as the framework that all UML-based plugins should base on. Believing this publicity caused loss of time investigating the EMF framework. After the analysis done and sources read, it was realized that there was no need to use EMF. EMF is very useful if the UML plugins are implemented from scratch and their model management is done then with EMF: But Poseidon already has its own mechanisms and components (MDR) as well as a lot of implemented base code for UML model management as well as code generation.

As part of this work, a deep analysis of the approaches adopted by different tools for code generation, reverse engineering and roundtrip engineering was done. The focus on the reviews has been *code generation from implementation level UML diagrams*. An implementation level UML model is another way of representing the source code in a visual way. There is no difference in the level of representation between the source code and such UML models. The mapping is assumed to be easy. But, the UML does not suggest any implementation languages such as Java or C++. UML concepts or constructs such as associations, aggregations or compositions are not defined in terms of OO-language constructs. The ways that are suggested in the literature are presented in section 8.3.

After the analysis and survey of the literature on code generation and reverse engineering, the design and implementation phase took place. Poseidon is integrated to the Eclipse Platform as a plugin. This plugin interacts with the JDT environment of Eclipse so as to synchronize Java source code and UML models. The Java development environment is at the phase of implementation thought to be the most suitable perspective in Eclipse Workbench.

The plugin which integrates Poseidon with Eclipse Platform uses the UML-Java code generator plugin of Poseidon to generate code for the newly created UML model elements. As the UML model changes, the changes on the source code are not done by UML-Java code generator, but by directly manipulating and modifying the corresponding source code with the help of the Eclipse Java Model and JDT's AST API.

In the implementation, the reverse engineering plugin of Poseidon is also used to obtain full UML models from Java sources contained in the Java projects in the Eclipse workspace. The reverse engineering plugin, in its current state, is not suitable to use for continuous synchronization (i.e. roun-tripping). It is only used in the beginning to create the UML models corresponding to the Java projects. This plugin can reverse engineer as the smallest unit a Java source file with the class path provided in its input. But during synchronization any small change on the Java source should trigger changes on the UML model and if this is done with the reverse engineering plugin, the performance would be very bad as this plugin can reverse a file as a smallest unit. For source code changes after creation of the initial UML model, the update on the UML models is directly provided considering typical code manipulations, omitting the use of this reverse engineering component. `IElementChangedListener` interface is used for analysis of source code change as much as it allows and event-based information is retrieved to update the UML model(e.g the new visibility or the new name after a rename operation).

The main problems encountered during implementation phase are explained in detail in section 9.2.8. AST handling and documentation or MDR class loader problem as well as the threading problems between Swing and SWT are only a few of the encountered problems.

11.2 Future Work

There is still much functionality that needs to be implemented to finish the Poseidon plugin for Eclipse. The ones that are most important are listed as follows:

1. Commands implementation for unimplemented commands

   The synchronization implemented during the thesis consists of a subset of the possible changes on the UML model and Java sources. The implemented functionality is added by replacing commands in the command factories of Poseidon. The unimplemented ones from the replaced command factory for semantic model operations should be completed to achieve a full synchronization.

2. Diagram Interchange commands

   The synchronization process does not consider the layout of the UML model elements, which is done by using Diagram Interchange component. The visual-related UML diagram commands are listed in Poseidon in `CommandDiFactory` interface. To help the user to really avail from the synchronization, layouting should be done automatically, which means, DI commands should be implemented

3. Update the UML model manually as Java element listener does not provide enough details for achieving a real-time synchronization.

   The problem with the listener interface `IElementChangedListener` of Java Model has been described in the previous chapter. An algorithm to synchronize the UML model on source code change should be developed which will be executed manually by the user. For example, after editing the Java source class `X` in the Java editor, it should be possible that the user selects the UML class whose source code is changed, and executes the updating of the UML model of the class manually.

4. Synchronization of associations, aggregations, and compositions

   The subset of the synchronization task implementation has not considered associations, aggregations and composition. These UML constructs should be included in the synchronization procedure.

5. The GUI of the plugin needs to be improved for a complete integration

   Currently, the GUI of the implemented plugin has views showing the Poseidon panels, some popup menu actions for project handling and a Poseidon menu item which executes simple actions of Poseidon standalone that indirectly work on the UML model such as *Save diagram as Graphics* or *Print Diagram*. These extensions can be seen in the Java perspective of the Eclipse workbench. Possible GUI advances should be considered and implemented.

Appendix A: Plugin Descriptor of the Implemented Plugin

Here you can find the full plugin.xml file for the Eclipse plugin implemented.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
     id="com.gentleware.poseidon.integration.ide.eclipse"
     name="Poseidon Plugin"
     version="1.0.0"
     provider-name="Sunay YALDIZ"
     class="com.gentleware.poseidon.integration.ide.eclipse.PoseidonviewPlugin">

  <runtime>
     <library name="../services/lib_build/services.jar">
        <export name="*"/>
     </library>
     <library name="../services/lib/trove.jar">
        <export name="*"/>
     </library>

<!-- List of all libraries Poseidon and the plugin needs : In total 59 libraries listed -->

     <library name="../mdr_service/lib/org-openide-execution.jar">
        <export name="*"/>
     </library>
     <library name="../mdr_service/lib/org-openide-io.jar">
        <export name="*"/>
     </library>
  </runtime>

<!—The required plugins listed with their id. These are mostly Eclipse Platform plugins -->

  <requires>
     <import plugin="org.eclipse.ui"/>
     <import plugin="org.eclipse.core.runtime.compatibility"/>
     <import plugin="org.eclipse.core.resources"/>
     <import plugin="org.eclipse.jdt"/>
     <import plugin="org.eclipse.jdt.core"/>
     <import plugin="org.eclipse.jface.text"/>
     <import plugin="org.eclipse.ui.editors"/>
     <import plugin="org.eclipse.ui.ide"/>
     <import plugin="org.eclipse.ui.workbench.texteditor"/>
     <import plugin="org.junit"/>
  </requires>

<!-- Finally  defining the extension points which are extended by extensions-->
<!—The extensions extending the view extension point-->

    <extension
        point="org.eclipse.ui.views">
     <category
           name="Poseidon category"
```

```
            id="com.gentleware.poseidon.integration.ide.eclipse">
      </category>
      <view
            name="NavPane View"
            icon="icons/sample.gif"
            category="com.gentleware.poseidon.integration.ide.eclipse"
            class="com.gentleware.poseidon.integration.ide.eclipse.views.NavPaneView"
            id="com.gentleware.poseidon.integration.ide.eclipse.views.NavPaneView">
      </view>
      <view
            name="Diagram View"
            icon="icons/sample.gif"
            category="com.gentleware.poseidon.integration.ide.eclipse"
            class="com.gentleware.poseidon.integration.ide.eclipse.views.DiagramView"
            id="com.gentleware.poseidon.integration.ide.eclipse.views.DiagramView">
      </view>
      <view
            name="PerspectivePane View"
            icon="icons/sample.gif"
            category="com.gentleware.poseidon.integration.ide.eclipse"
            class="com.gentleware.poseidon.integration.ide.eclipse.views.PerspectivePaneView"
            id="com.gentleware.poseidon.integration.ide.eclipse.views.PerspectivePaneView">
      </view>
      <view
            name="DetailsPane View"
            icon="icons/sample.gif"
            category="com.gentleware.poseidon.integration.ide.eclipse"
            class="com.gentleware.poseidon.integration.ide.eclipse.views.DetailsPaneView"
            id="com.gentleware.poseidon.integration.ide.eclipse.views.DetailsPaneView">
      </view>
   </extension>

<!—The extensions extending the view perspectiveExtension extension point: Java Perspective is
extended-->

   <extension
         point="org.eclipse.ui.perspectiveExtensions">
      <perspectiveExtension
            targetID="org.eclipse.jdt.ui.JavaPerspective">
         <view
               ratio="0.4"
               relative="org.eclipse.ui.views.ResourceNavigator"
               id="com.gentleware.poseidon.integration.ide.eclipse.views.NavPaneView"
               relationship="bottom">
         </view>
      </perspectiveExtension>
      <perspectiveExtension
            targetID="org.eclipse.jdt.ui.JavaPerspective">
<!--
               ratio="0.5"
               relative="org.eclipse.ui.views.TaskList"
               relationship="right">
              -->
         <view
               relative="org.eclipse.ui.views.Properties"
               id="com.gentleware.poseidon.integration.ide.eclipse.views.DiagramView"
            relationship="stack">
        </view>
      </perspectiveExtension>
      <perspectiveExtension
            targetID="org.eclipse.jdt.ui.JavaPerspective">
        <view
              ratio="0.5"
              relative="org.eclipse.ui.views.ContentOutline"
              relationship="bottom"
              id="com.gentleware.poseidon.integration.ide.eclipse.views.PerspectivePaneView">
        </view>
      </perspectiveExtension>
      <perspectiveExtension
            targetID="org.eclipse.jdt.ui.JavaPerspective">
        <view
              relative="org.eclipse.ui.views.Properties"
              id="com.gentleware.poseidon.integration.ide.eclipse.views.DetailsPaneView"
              relationship="stack">
        </view>
      </perspectiveExtension>
   </extension>

<!—The extensions extending the objectContribution extension point: The popup menu for
IJavaProject instances is extended -->

   <extension
         point="org.eclipse.ui.popupMenus">
      <objectContribution
            objectClass="org.eclipse.jdt.core.IJavaProject"
            nameFilter="*"
            id="com.gentleware.poseidon.integration.ide.eclipse.contribution1">
         <menu
               label="UML"
               path="additions"
               id="com.gentleware.poseidon.integration.ide.eclipse.menu1">
            <separator
                  name="group1">
            </separator>
         </menu>
         <action
               label="Reverse Engineer  "
               class="com.gentleware.poseidon.integration.ide.eclipse.popup.actions.ReverseEngineering
Action"
               menubarPath="com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
               enablesFor="1"
               id="com.gentleware.poseidon.integration.ide.eclipse.ReverseEngineeringAction">
         </action>
         <action
               label="Import Poseidon-Uml Model  "
               class="com.gentleware.poseidon.integration.ide.eclipse.popup.actions.ImportPoseidonUml
ModelAction"
               menubarPath="com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
               enablesFor="1"
               id="com.gentleware.poseidon.integration.ide.eclipse.ImportPoseidonUmlModelAction">
         </action>
         <action
```

```
            label="Open Poseidon-Uml Model  "
            class="com.gentleware.poseidon.integration.ide.eclipse.popup.actions.OpenPoseidonUmlM
odelAction"
            menubarPath="com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
            enablesFor="1"
            id="com.gentleware.poseidon.integration.ide.eclipse.OpenPoseidonUmlModelAction">
        </action>
        <action
            label="Create Poseidon-Uml Model  "
            class=
 "com.gentleware.poseidon.integration.ide.eclipse.popup.actions.
 CreatePoseidonUmlModelAction"
            menubarPath=
       "com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
            enablesFor="1"
            id=
       "com.gentleware.poseidon.integration.ide.eclipse.CreatePoseidonUmlModelAction">
        </action>
      </objectContribution>
   </extension>
   <extension
        point="org.eclipse.ui.popupMenus">
      <objectContribution
          objectClass="org.eclipse.jdt.core.ICompilationUnit"
          nameFilter="*"
          id="com.gentleware.poseidon.integration.ide.eclipse.contribution2">
        <menu
            label="Update UML Model"
            path="additions"
            id="com.gentleware.poseidon.integration.ide.eclipse.menu1">
          <separator
              name="group1">
          </separator>
        </menu>
        <action
            label="Reverse Engineering Action-File"
            class=
       "com.gentleware.poseidon.integration.ide.eclipse.popup.actions.
        FileReverseEngineeringAction"
            menubarPath=
         "com.gentleware.poseidon.integration.ide.eclipse.menu1/group1"
            enablesFor="1"
            id=
     "com.gentleware.poseidon.integration.ide.eclipse.FileReverseEngineeringAction">
        </action>
      </objectContribution>
   </extension>
```

<!—The extensions extending actionSets extension point: Eclipse Workbench Menu as well as Toolbar is extended -->

```
   <extension
        point="org.eclipse.ui.actionSets">
      <actionSet
          label="Poseidon Action Set"
          visible="true"
          id="com.gentleware.poseidon.integration.ide.eclipse.actionSet">
        <menu
              label="Poseidon &amp;Menu"
              id="poseidonMenu">
          <separator name="poseidonGroup"/>
        </menu>
        <action
              toolbarPath="poseidonGroup"
              label="&amp;Print"
              class=
         "com.gentleware.poseidon.integration.ide.eclipse.actions.PrintAction"
              tooltip="Print Current Poseidon Diagram"
              icon="icons/sample.gif"
              menubarPath="poseidonMenu/poseidonGroup"
              id=
       "com.gentleware.poseidon.integration.ide.eclipse.actions.PrintAction"/>
        <action
              toolbarPath="poseidonGroup"
              label="&amp;Print Second"
              class=
         "com.gentleware.poseidon.integration.ide.eclipse.actions.PrintAction"
              tooltip="Print Current Poseidon Diagram action 2"
              icon="icons/sample.gif"
              menubarPath="poseidonMenu/poseidonGroup"
              id=
        "com.gentleware.poseidon.integration.ide.eclipse.actions.PrintAction"/>
      </actionSet>
   </extension>
 </plugin>
```

REFERENCES

1 [] Grady Booch. *Object-Oriented Design with Applications*. Number 0-805-30091-0. The Benjamin/Cummings Publishing Company, Inc., 390 Bridge Parkway – Redwood City, California 94005 USA, 1991

2 [] Anders Henriksson, Henrik Larsson (2003*). A definition of roundtrip engineering*, Technical report (not published)

3 [] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Conquering complex and changing systems*, Prentice-Hall, 2000.

4 [] U. Nickel, J. Niere, J. Wadsack, and A. Zündorf: *Roundtrip Engineering with FUJABA*. In Proc. of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany, Technical Report 8/2000, Fachberichte Informatik, Universität Koblenz-Landau, 2000.

5 [] T. Fischer, J. Niere, L. Torunski, A. Zündorf. *Story Diagrams: A New Graph Grammer Language based on the Unified Modelling Language and Java*, in Proc. of TAGT '98 (Theory and Application of Graph Transformations), LNCS 1764, pp. 296-309, ISBN 3-540-67203-6, Springer 1999

6 [] H.J. Köhler, U. Nickel, J. Niere, A. Zündorf, *Integrating UML Diagrams for Production Control System*, to appear in Proc. of the 22nd Intl. Conf. on Software Engineering, Limerick, Ireland, June 2000.

7[] Omondo EclipseUML Modeling tool, http://www.omondo.com/

8[] Generic Libraries for Java (JGL®), Version 3.1, http://www.recursionsw.com/jgl.htm. September 2004

9[] Chikofsky E. and Cross J., *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software

10 [] Yann-GaÄel Gueheneuc, *Abstract and precise recovery of UML class diagram constituents*, Proceedings of ICSM, 2004. Poster.

11 [] J. Korn, Y.-F. Chen, and E. Koutso¯os. *Chava: Reverse engineering and tracking of Java applets*. In proceedings WCRE, pages 314{325. IEEE CS Press, Nov. 1999.

12 [] D. Jackson and M. C. Rinard. *Software analysis: A roadmap*. In proceedings of ICSE, pages 133{145. ACM Press, Jun. 2000.

13[] Yann-Gaël Guéhéneuc. *A Reverse Engineering Tool for Precise Class Diagrams*. Proceedings of CASCON, 2004.

14 [] G. C. Gannod and B. H. C. Cheng. A framework for classifying and comparing software reverse engineering and design recovery techniques. In *proceedings of WCRE*, pages 77{88. IEEE CS Press, Oct. 1999.

15 [] Yann-Gaël Guéhéneuc, Hervé Albin-Amiot. *Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects*. 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), August 2001

16 [] Carmen Zannier. Master Thesis: *Tool Support for Complex refactoring to design patterns,* at University of Calgary, August 2003

17 [] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, Narendra Jussien, École des Mines de Nantes. *Instantiating and Detecting Design Patterns: Putting Bits and Piece.s* Together 16th IEEE International Conference on Automated Software Engineering (ASE'01), November 2001

18 [] R. Kollman, P. Selonen, E. Stroulia, T. Systä, A. Zundorf .*A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering*, Ninth Working Conference on Reverse Engineering (WCRE'02) , November 2002

19[] University of Paderborn. Fujaba, 2002. `http://www.fujaba.de`

20 [] Ralf Kollmann and Martin Gogolla. Application of UML Associations and Their Adornments in Design Recovery. In Peter Aiken and Elizabeth Burd, editors, *Proc. 8th Working Conference on Reverse Engineering (WCRE)*, pages 81–90. IEEE, Los Alamitos,2001

21[] Presentation on Roundtrip Engineering, George Mason University www.isse.gmu.edu/~duminda/classes/ spring04/RoundTrip-Engineering_v3.ppt

22[ ] UML Distilled, Third Edition, Martin Fowler, Addison-Wesley Publishing, 2004, ISBN 0-321-19368-7

23[] OMG UML 1.4 Specification, 2001

24[] UML Toolkit, Hans-Erik Erikkson, Magnus Penker, Wiley Computer Publishing, 1998, ISBN:0-471-19161-2

25[] Metaobject Facility, MOF Specification, Version 1.4, http://www.omg.org/mof/, September 2004

26[] Poseidon for UML Users Guide version 2.5, September 2004

27[] Poseidon for UML Documentation, Hints . http://www.gentleware.com/support/dev/hints.php4, September 2004

28[] Fransson, Jens; Müller, Stefan; UML Diagramme – Austausch und Interaktion, Diplomarbeit at Universität Hamburg in Fachbereich Informatik, July 2003

29[] Diagram Interchange Specification, UML 2 Diagram Interchange

30[] XMI Metadata Interchange specification, XMI version 2.0, http://www.omg.org/technology/documents/formal/xmi.htm, September 2004

31[] Eclipse Platform Technical Overview, http://www.eclipse.org/whitepapers/eclipse-overview.pdf

32[] Eclipse Platform 3.0 Help, JDT Plugin Developer Guide

33 [] Contributing to Eclipse, Principles, Patterns and Plugins, Erich Gamma, Kent Beck, Addison-Wesley Publishing, October 2003, ISBN: 0321205758

34 [] William Harrison, Charles Barton, Mukund Raghavchari. Mapping UML Designs to Java. *Conference Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 178-187 (October 2000).

35[] AspectJ, http://www.eclipse.org/aspectj/

36 [] G. Kiczales. E.Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. An overview of AspectJ. In Proc. Of 15th. ECOOP, LNCS 2072, p. 327-353, Springer-Verlag, 2001

37[] S.Herrmann. Composable Designs with UFA. Submission to AOSD 2002.

38[] No Magic, Magicdraw, http://www.nomagic.com/magicdrawuml/features.htm.

39[] Object International Software, Together/J, http://www.togethersoft.com/together/togetherJ.html.

40[] Rational Software, Rational Rose,
http://www.rational.com/products/rose/index.jtmpl

41[] Softera, SoftModeler,
http://www.softera.com/manual/UserGuide.htm.

42[] OMG Model Driven Architecture, MDA, http://www.omg.org/mda/

43 [] K. O. Chow, Weijia Jia, Vito C. P. Chan, Jiannong Cao: *Model-Based Generation of Java Code*. Parallel and Distributed Processing Techniques and Applications (PDPTA) 2000, Las Vegas, Nevada, USA

44[] S. Shlaer, S.J. Mellor, The Shlaer-Mellor Method, Project Technology White paper, 1996.

45 [] I. Jacobson, G. Booch and J. Rumbaugh.The Unified Software Development Process, Addison-Wesley, 1999.

46 [] Ossher, H., P. Tarr, W. Harrison, and S. Sutton, *N Degrees of Separation: Multi-Dimension Separation of Concerns*. Proceedings of 1999 International Conference on Software Engineering (May 1999)

47 [] Clarke, S., W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code," *Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Languages, and Applications,* Denver (November, 1999).

48 [] Joern Bettin, Equinox Ltd. Software Architects, *A language to describe software texture in abstract design models and implementation*, White Paper, September 2001

49 [] Iris Groher, Stefan Schulze, *Generating Aspect Code from UML models,* The 4th AOSD Modeling With UML Workshop, October 2003

50 [] S. Herrmann, M. Mezini. Aspect-Oriented Software Development with Aspectual Collaborations. Submission to ECOOP 2002.

51 [] William H. Harrison. *Composition and Multiple-Inheritance in OO Design(Where in the Madness is the Method?)*, OOPSLA 2001 Workshop on Advanced Separation of Concerns, October 2001.

52[] HyperJ, http://www.alphaworks.ibm.com/tech/hyperj

53[] Tengger, http://www.alphaworks.ibm.com/tech/tengger

54 [] Yann-Gaël Guéhéneuc, Hervé Albin-Amiot, Rémi Douence, and Pierre Cointe. *Bridging the Gap*
Between Modeling and Programming Languages. Technical report 02/09/INFO, École des Mines de Nantes, May 2002.

55 [] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

56 [] Gonzalo Génova, Carlos Ruiz del Castillo, Juan Lloréns: *Mapping UML Associations into Java Code*. Journal of Object Technology 2(5): 135-162 (2003)

57 [] James Rumbaugh. "A Search for Values: Attributes and Associations". *Journal of Object Oriented Programming*, 9(3):6-8, June 1996.

58 [] James Rumbaugh. "Relations as Semantic Constructs in an Object-Oriented Language", In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pp. 466-481, Orlando, Florida, 1987.

59 [] Universität Wien. Aggregation and composite objects., 2000 www.ifs.univie.ac.at/ISOO/OOCONCEPT/aggregation.html.

60[] Eclipse Modeling Framework Web Page, http://www.eclipse.org/emf/,

61[] IBM EMF Redbook, *last visited August 2004* http://www.redbooks.ibm.com/redbooks/SG246302/wwhelp/wwhimpl/js/html/wwhelp.htm,

62 [] Eclipse Modeling Framework, Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose, Eclipse Series, October 2003, ISBN 0-13-142542-0

63 [] MDA Journal, A BPT Column, *Domain-Specific Modeling and Model Driven Architecture*, January 2004

64[] JetBrains Company, http://www.jetbrains.com/

65[] Fabrique tool, http://www.jetbrains.com/fabrique/index.html

66 [] Design Patterns, Erich Gamma, Richard Helm,Ralph Johnson, John Vlissides ; Addison Wesley Proffessional Computing Series, 1995

67[ ] Metadata Repository, MDR, http://mdr.netbeans.org/; September 2004