Diploma Thesis

April 26, 2006

SEAL platform

Towards an Integrated Tool Platform for Software Architecture and Evolution Analysis

Christian Hanimann
of Moerschwil, Switzerland (99-916-389)

supervised by
Prof. Dr. Harald Gall
Dr. techn. Martin Pinzger

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

SEAL platform

Towards an Integrated Tool Platform for Software Architecture and Evolution Analysis

Christian Hanimann

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Author: Christian Hanimann, christian.hanimann@access.unizh.ch

Project period: 1 November 2005 - 1 May 2006

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank my supervising assistant Martin Pinzger for his great assistance. My thanks also to Prof. Harald Gall for giving me the opportunity to write this thesis.

Many thanks to Beat Fluri for his input concerning LaTeX and his LaTeX-style on which this document is based.

Thanks also to Nick Bell for proofreading my English. And many thanks to Bettina Vetterli for her moral support and patience with me during this work.

Abstract

Software maintenance and evolution are important tasks in the software lifecycle. To make software maintenance and evolution easier, procedures exist to represent the software as a model and to measure the software. There are several graphical approaches to represent this generated data.

This thesis concerns part of this work. To save a generated FAMIX model of a software durable, the data of this model is saved with Hibernate in a relational database. The metrics of this software, computed with the Metrics plug-in, are mapedp with the corresponding entities of the FAMIX model and are also stored in the database.

The metrics are visualised with the Kiviat Visualizer. On the basis of these graphs, several questions, concerning architecture, design and evolution, will be answered.

Zusammenfassung

Software Wartung und Evolution ist in der heutigen Zeit eine wichtige Aufgabe im Lebenszyklus einer Software. Um die Wartung und Evolution zu vereinfachen, wird versucht, die Software in Modelle abzubilden und sie zu vermessen. Um die Resultate anschaulich darzustellen existieren verschiedene graphische Anstze.

Diese Arbeit widmet sich einem Teilgebiet dieser Aufgabe. Um ein generiertes FAMIX Modell einer Software dauerhaft verfügbar zu machen, wird dieses Modell mit Hibernate in einer relationalen Datenbank gespeichert. Zudem werden die mit dem Metrics plug-in erzeugten Metriken den richtigen Entitten zugeordnet und ebenfalls mit Hibernate in der Datenbank gespeichert.

Mit dem Kiviat Visualizer werden diese Metriken dann graphisch dargestellt. Anhand dieser Graphen werden dann verschiedene Fragen zur Architektur, zum Design und zur Evolution beantwortet.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

The software we use evolves over the time. It becomes more and more complex, the size increases and the quality declines [Leh97]. Bugs are fixed, new functions are added, a number of functions removed and so on. We can see that during a life cycle of a software system we have a large number of changes. We can differentiate between four categories of software changes:

- **Adaptive:** Changes to adapt the system to new environments such as new operating systems, compilers and other tools and components
- **Corrective:** Changes to repair defects in the software system
- **Perfective:** Changes to improve the product, such as adding new requirements, or to enhance the performance
- **Preventive:** Bug fixing

After a large number of changes a software system is increasingly hard to maintain. So the moment comes to reengineer the software system. Seacord et al. [RCS03] describe software reengineering as follows:

> Software reengineering is a form of modernization that improves capabilities and/or maintainability of a legacy system by introducing modern technologies and practices.

To carry out reengineering it is necessary to understand the software system we need to reengineer. For this purpose we can study the documentation of the system and look at the code. But the problem is that the documentation is often not adequate or not available. And such a system has often hundreds and thousands of lines of code. It is not easy to gain an overview of the functions of this code.

A way to get a better understanding of a software system is to use Reverse Engineering. The definition of Reverse Engineering is as follows, according to Chikofsky et al. [EJC90]:

> Reverse engineering is the process of analysing a subject system to
> - identify the systems components and their interrelationships and
> - create representations of the system in another form or at a higher level of abstraction.

A representation of a system is an abstract model of the code. A model represents all components, depending on the source language. Possible components are: directories, files, packages, classes, methods, different types of variables, attributes. In addition it describes how these different components depend on each other and how they interact.

To represent the data we use an extended version of the FAMIX meta model [SD99a] in this diploma thesis. This is a language-independent model to represent object-oriented source code data. We give a short introduction to FAMIX in chapter 4.

1.2 The goal of the diploma thesis

In this diploma thesis we use the ArchView Approach [Pin05] in a case study with a Java software project. We aim to answer a set of research questions concerning the design, architecture and evolution of this software.

To carry out this case study, the following steps are necessary:

1. Extending the FAMIX exporter tool.
2. Storing the extracted FAMIX model in a relational database.
3. Implementing a tool to export software metrics, generated with the Java metrics tool.
4. Storing the exported metrics in a relational database.
5. Defining a set of research questions concerning the design, architecture and evolution of a Java software project.
6. Creating various higher-level views of the source code model.
7. Answering the research questions.

1.3 Structure of the diploma thesis

- Chapter 2 gives an introduction to the related work of software architecture and evolution analysis and the visualisation of extracted source code information
- Chapter 3 introduces the catalogue of research questions concerning the design, architecture and evolution of a piece of software.
- Chapter 4 provides the background information about the meta model used.
- Chapter 5 describes the implementation of the two plug-ins used to obtain and store the meta model and the software metrics.
- Chapter 6 describes the case study.
- Chapter 7 presents our conclusion to this thesis.

Chapter 2

Related Work

This diploma thesis applies the ArchView approach represented in [Pin05]. This approach is an architecture recovery and analysis approach with higher-level views of a software system. To gain an short overview of existing techniques, this chapter shows several other approaches.

2.1 Software Architecture Analysis

The scope of architecture analysis is very large. It is possible to find many different appendages to analyse software. We can find methods which are supported with tools to do the different tasks automatically or we can also find methods with many (or only) manual tasks.

In 2000, Gannod et. al. [GCG00] have described an approach in which a method of manual and automated architecture analysis is used. First they generated a model by hand, using documentation, source code and communication with developers. To build a model for the automated analysis, they had to design a new model with the ACME ADL[1], based on this manually extracted information. ACME allows the architecture in different ADL's to be described so that the model is extensible.

A more automated way is the approach of Rötschke et al. [TR02]. They realized a tool-set to do this work and used (still incomplete) UML[2] models and source code to extract information. With the CDMA tool they extracted relevant information from the source code and stored this information in a database. With another tool they also extracted information from the UML diagram and tried to map this information using the information extracted from the source code. Detected violations are also stored in the database. To examine the generated data they used PHP scripts to generate tables or diagrams.

Riva et al. [CR04] used UML as a model to represent the architecture of a software system. To create the model they had to map the source code files by hand with the logical components. To gather the relevant information from the implementation they used a Python script. These extracted data are needed to calculate high-level dependencies between the packages. The results can be visualized as hierarchical graph. In this graph, the nodes represent packages and components. The arcs represent logical dependencies.

Another approach to analysing architecture is described in [FdB]. De Boer et al. used XML for static and dynamic analysis of architectures and to express the signature of an architecture. It is possible to use this model with other tools and data can be easily shared.

[1] ACME homepage: http://www.cs.cmu.edu/ acme/
[2] UML homepage: http://www.uml.org

2.2 Software Evolution Analysis

As described in [Leh97] software maintenance is becoming increasingly complex. For this reason it is necessary to have one or more processes in order to analyse software evolution adequately. In recent years, more and more research groups have developed new techniques to do this work.

Burd et al. [EB00] analysed the calling structure of each version of a software system and compared the generated structures with all versions. They recorded all modifications, such as addition or deletion of procedural units or calls within a specific procedural unit. Afterwards they analysed the data usage in all software versions in the same way.

Lanza [Lan02] introduced the Evolution Matrix to trace software evolution. He used different metrics to generate a matrix, known as the Evolution Matrix. To visualize a class of a certain program version, he needed two metrics, one for the width, one for the height. It is possible to visualize the life cycle of a class with this information.

CVSscan is a tool from Voinea et al. [LV05]. They used the data of CVS system to generate a data model. With this model, they computed the difference between several versions of a software system. To show this differences between all versions, they used a visualization tool. With this method it is also possible to show time influence, which means it is possible to show when an entity appears or disappears.

2.3 Visualization

To understand all generated information it is useful to visualize this data in a suitable way. A number of software evolution or architecture analysis methods use their own visualization, but often we obtain only the data of the analysis. Because reverse engineering and reengineering have become more important in the last few years, various new techniques to visualize data have been developed. We can differentiate between 2D and 3D methods of carrying out this task.

Lanza et al. [ML03] described a method to visualize data, named Polymetric Views. They used the visualization tool CodeCrawler[3] to analyse object-oriented software. The ArchView approach which we apply in this thesis, uses an extended Polymetric View technique. Polymetric Views use two-dimensional displays. Nodes represent software entities and edges represent relationships between those entities.

- **Node Size:** The height and the weight can each render a metric.
- **Node Colour:** Another metric be visualised with the colour of a node. The higher the metric value is, the darker the node is.
- **Node Position:** It is possible to visualise two metrics with the X and Y coordinates of the node position.

Another way to visualise information is ”Butterflies”, see figure 2.1, described in [SD05]. In this case Butterflies were used to characterize packages and class dependencies. Butterflies are based on a radar visualization, which is in turn based on dividing a circular area with a certain number of axes. Radar visualizations are complex and so Ducasse et al. defined a distribution of the metrics to generate a butterfly shape. They defined the butterfly in the following way:

- **The left wing** represents what a package provides to other packages.
- **The right wing** represents what a package uses from other packages.

[3]CodeCrawler: http://www.iam.unibe.ch/ scg/Research/CodeCrawler/

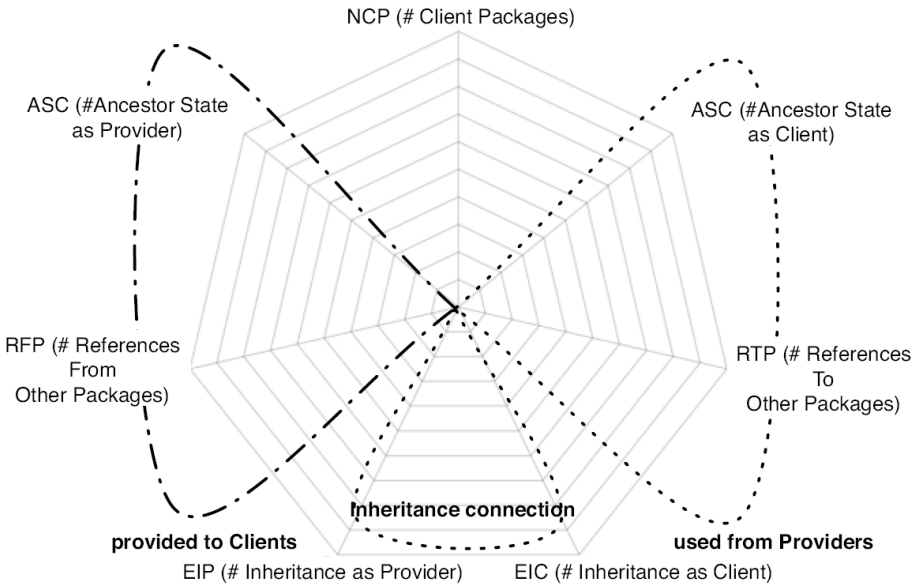

Figure 2.1: A butterfly. Figure from [SD05].

- **The bottom part** shows the inheritance (e.g. a package has classes which have subclasses in other packages).

A similar method was also used by Chuah et al. [MCC97]. They used 3D glyphs to represent data. These glyphs can show many different types of data (e.g. releases, engineers, features, etc.). Each variable is encoded as an equal slice of a circle. The radius of a slice encodes the size of the representing variable and each variable has its own colour.

Chapter 3

Design Quality Catalogue

In this chapter, we define a number of research questions which we will answer later in chapter 6.

3.1 Introduction

In the evaluation we use our FAMIX and Metrics export plug-ins. With the Kiviat Visualizer we generate different graphs and we use these graphs to answer the defined research questions.

With these research questions, we examine the design, architecture and evolution of a piece of software. We also want to check if these tools are adequate for this problems or if the tools have to be adapted or extended.

3.2 Basic conditions

To do this work, we have a number of technical limitations. These limitations are:

In this version, the Kiviat Visualizer:

- offers only the possibility to visualize one release of a software system.
- is not able to visualise dependencies between entities (for example between classes).

3.3 The questions

We define the questions which we answer later in this diploma thesis.

1. **What can a package say about software quality?** A package contains several classes. There is interaction between classes in the same package and between these classes and classes in other packages. What can we learn about a software system at this level?
2. **Is it possible to find large classes?** Large classes can have a large number of lines of code, but we can also define this as a class with many methods and/or attributes. Can we detect these classes with our tools?
3. **Is it possible to detect complex classes?** If we have large classes then they are often also complex classes. A class which has many methods and/or attributes is often in an interaction with many other classes.

4. **What can we say about inheritance?** Used in the right way, inheritance has a number of advantages. Can we say something about the usage of inheritance?
5. **What are the large methods?** Large methods are harder to maintain and complexity increases with size. Can we identify the large methods?
6. **Can we identify the complex methods?** Are we able to identify the complex methods? Are the complex methods also the large methods which we tried to identify in the previous step?
7. **Can we say something about evolution?** Do these tools allow us to make a prediction about evolution?

Chapter 4

Famix-Background

This chapter provides the background information about the FAMIX [SD99a] meta model. This model is used to store the extracted source code data of a Java project.

4.1 Purpose

The FAMIX meta model (FAMOOS Information Exchange Model) was developed by the University of Berne in 1999. It is a language-independent model to represent object-oriented source code data. It defines the exchange model which was used within the FAMOOS reengineering[1] project.

The idea behind FAMIX was to introduce an alternative to the current standard in object-oriented modelling languages, UML[2] from the Object Management Group. The group (Demeyer et al. [SD99b]) which developed the FAMIX model, argues that UML is not sufficient to serve as a tool-interoperability standard for integrating round-trip engineering tools. This is because UML defines different concepts that do not appear in the implementation model. And otherwise there are concepts in the implementation model which don't have any equivalents in UML. Demeyer et al. use this new FAMIX meta model as the tool interoperability standard within the FAMOOS project.

The FAMIX model is extensible: the model allows extensions with language specific entities and properties if needed. It contains only parsed and not interpreted data.

[1]The FAMOOS Project: http://www.iam.unibe.ch/ scg/Archive/famoos/
[2]UML: http://www.uml.org/

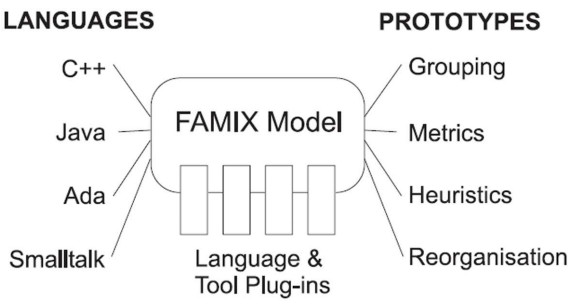

Figure 4.1: Conception of the FAMIX model. Figure from [SD99a].

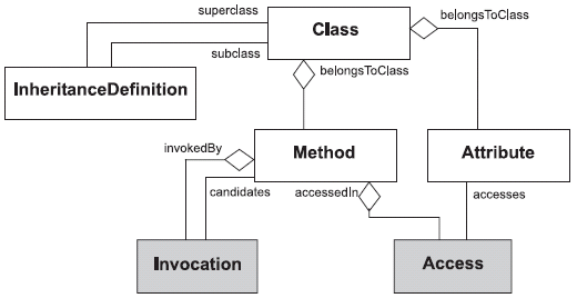

Figure 4.2: The FAMIX core model. Figure from [SD99a].

Figure 4.1 shows the conception of the FAMIX model. For each language a tool which implements the language dependent parsing technology is needed. These tools get the source code of a program written in this programming language and parse the information into the FAMIX model. The information is now language independent and can be used by any other program.

Each different language needs a specific plug-in because each language has its own entities and properties. And it is also possible that a tool needs to define a number of specific properties.

4.2 Description

4.2.1 Overview

As described in section 4.1 the FAMIX model is a language-independent model to represent object-oriented source code data. To enable the transfer of the generated data between different tools, FAMIX is based on CDIF [Par92]. This is an industrial standard for transferring models. This model could be generated with different tools. The advantages of CDIF are:

1. It is an industrial standard
2. It uses standard plain text encoding
3. CDIF supports extensibility

4.2.2 The FAMIX core model

Figure 4.2 shows the core model of FAMIX. The entities and relations of the FAMIX core model can extracted immediately from source code. This core model consists of the main object-oriented entities:

- Class
- Method
- Attribute
- InheritanceDefinition

To compute metrics or analyse dependencies we need more information. For this reason there are two other entities in the FAMIX core model.

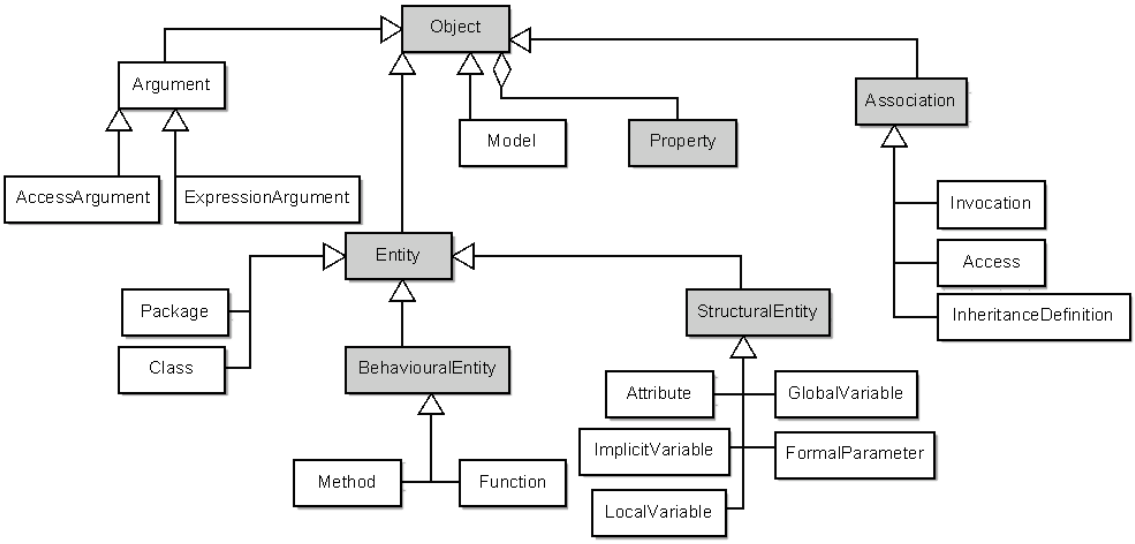

Figure 4.3: The complete original FAMIX model. Figure according to [SD99a].

1. Invocation
2. Access

With this additional information it is possible to answer different questions concerning the cooperation between the entities.

4.2.3 The abstract part of the FAMIX model

In the abstract part of the model all elements are children of the type object. Figure 4.3 shows the completely original FAMIX model with the abstract model (grey boxes). For the purpose of specifying language plug-ins, it is allowed to define new language specific objects and to add new attributes to existing objects.

Object, Property, Entity and Association are made available to manage the extensibility requirements, defined in [SD99a].

4.2.4 The extended FAMIX model

The Eclipse plug-in for the FAMIX export plug-in is based on the extended FAMIX model of Coogle [Sag06], an Eclipse plug-in for searching similar classes in Java projects. The FAMIX version of Coogle contains several additional attributes to represent extracted Java source code data and is itself based on the FAMIX Java language plug-in 1.0 [Tic99].

For our requirements we have adapted the FAMIX model in an other way than adding only new attributes or language specific objects. We have inserted a new class named *Generalization*. Furthermore we have inserted a class *Inheritance* and a class *Subtyping*.

- *Generalization* This class inherits from Association. It provides almost the same information as the class InheritanceDefinition. This information is: accessControlQualifier, subclass, superclass and index.

- *Inheritance* This class inherits from Generalization. It is used for the inheritance of a Java class. The class Inheritance knows the super class of a class.
- *Subtyping* This class also inherits from Generalization. The class *Subtyping* knows the Interface of a class.

This adaptation has a number of advantages compared to the official implementation with the class InheritanceDefinition. The InheritanceDefinition provides information about inheritance and subtyping. This is implemented with the constructor from listing 4.1. When the superclass is an interface, the boolean *extending* is true: otherwise it is false.

```
public InheritanceDefinition(Class subclass, Class superclass,
                             boolean extending)
```

Listing 4.1: The class InheritanceDefinition.

To get all interface classes, a tool has to check each instance of the InheritanceDefinition. With our extended model we are able to get this information directly. This fact also helps us to store the generated FAMIX model with Hibernate[3]. We are thus able to store this model information in two different tables in a database, one table for *Subtyping*, the other table for *Inheritance*.

[3]Hibernate: http://www.hibernate.org

Chapter 5

Implementation

Eclipse and Hibernate are two tools which we used to realise our FAMIX export plug-in and Metrics export plug-in. This chapter gives a short introduction to the technology of the Eclipse platform and Hibernate. It also describes the two plug-ins.

5.1 Eclipse

To store the generated FAMIX model and the metrics of a Java project, we developed two plug-ins for Eclipse[1]. For a better understanding of the Eclipse platform, we describe the most important points of the Eclipse SDK in this section.

5.1.1 The Eclipse platform

Eclipse is a modular platform. To realise this concept the developers of Eclipse chose plug-in technology. With a plug-in it is possible to extend the platform and to add new functions or even documentation. Each subsystem of Eclipse is also another plug-in or even a set of plug-ins. Figure 5.1 shows the Eclipse platform and the plug-in technology.

The most important subsystems of Eclipse are:

[1] Eclipse: http://www.eclipse.org

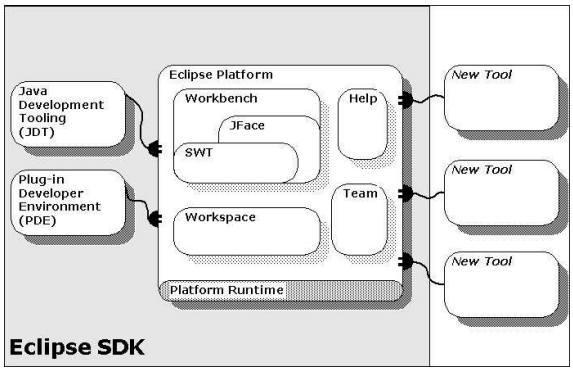

Figure 5.1: The Eclipse platform. Figure from [Fou].

- **Runtime core:** The most important part in general is the runtime core. This core implements the runtime engine. It starts the platform. And it is also responsible to discover and run plug-ins. This action is normally a dynamic activity, which means that a plug-in is known by Eclipse but will only be loaded and activated when the user needs the action provided by this plug-in. This is also known as the *Lazy Loading Rule* [EG04].

- **Workbench UI:** This part implements the workbench user interface. It defines different extension points to enable other plug-ins to enlarge the user interface with new menus, toolbar actions, new wizards etcetera. To generate this user interface the workbench user interface can use SWT (Standard Widget Toolkit). With this SWT it is possible to create the look and feel of the underlying operating system.

To develop Java software, Eclipse needs the JDT plug-in.

- **Java Development Tools (JDT):** The JDT plug-in extends the platform with different features. This plug-in provides functions for editing, viewing, compiling, debugging and last but not least running Java code. This plug-in is already integrated in the SDK version of Eclipse.

The FAMIX and Metrics export tools which we developed are also plug-ins. To be able to develop plug-ins we need another set of plug-ins, the PDE.

- **Plug-in Development Environment (PDE):** The PDE extends the JDT with tools to create, manipulate, debug and deploy new plug-ins. This set of plug-ins is also included in the SDK version of Eclipse.

5.1.2 An Eclipse plug-in

A plug-in extends the Eclipse platform with new functions or even documents. A plug-in contains numerous files. The central file is the *plugin.xml*. This file contains all the information about the plug-in, such as extension points, dependencies or extensions. This file contains all information and settings needed by the plug-in.

- **Dependencies:** In this point all additional plug-ins needed by the new plug-in must be declared . By default these are the three plug-ins org.eclipse.ui, org.eclipse.core.runtime, org.eclipse.jdt.core. If a new plug-in needs any additional plug-ins, they must be added. (Example: the Metrics export plug-in needs the additional Metrics 1.3.6[2] plug-in to get the computed metrics of a Java project.)

- **Extension Points:** To extend an existing plug-in, it is necessary to have a point to get access to this plug-in. The extension points exist for this purpose. An extension point defines how this plug-in can be extended without changing the plug-in.

- **Runtime:** If another plug-in wants to use one or more extension points of a plug-in, it is necessary to allow, this plug-in to see all needed packages and classes in this plug-in. Otherwise it is not possible to extend this plug-in because the new one needs class **A** from package **C** but it cannot access reach this class.

- **Extensions:** A new plug-in provides new functionality. To use this new functionality it is also necessary to provide a possibility to start or view it. For this purpose new extensions can be added to the plug-in. An extension can be a popup menu, a view or something else.

[2]The Metrics 1.3.6 plug-in: http://metrics.sourceforge.net

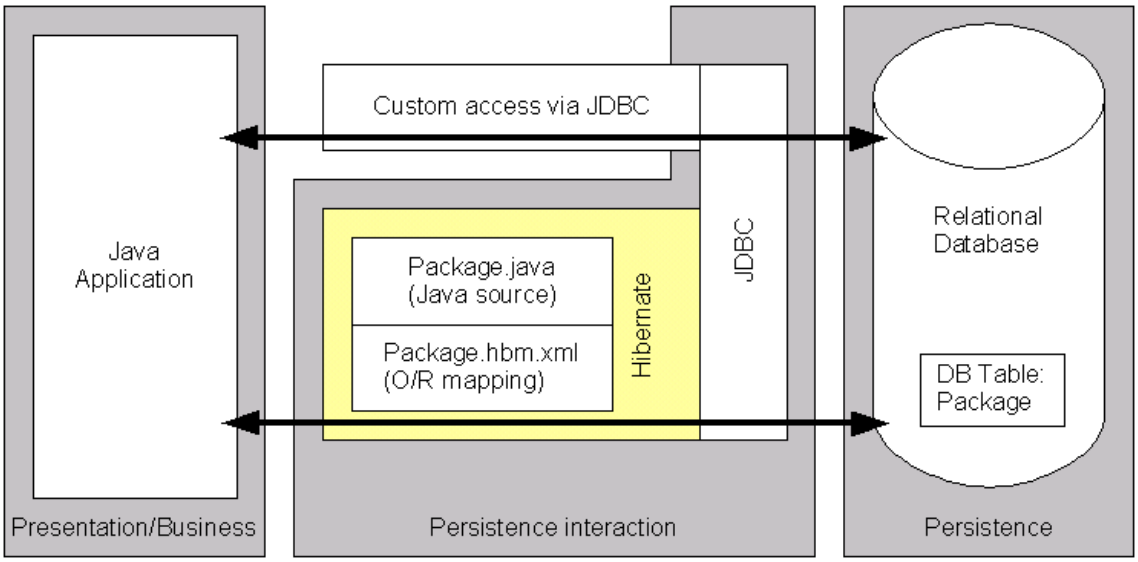

Figure 5.2: Two ways to store and read data in/from a database. Figure according to [Ive05].

In this *plugin.xml* file there is also information on how Eclipse has to build this plug-in for the contribution. This is necessary because a plug-in can use also external jar files, for example, and when Eclipse builds this plug-in for the contribution, it is also necessary to add these jar files in the plug-in and the classpath has to be set correctly.

This *plugin.xml* file will be created automatically when we start the wizard to create a new plug-in project. The wizard also generates Java classes. One of these classes is the *plugin.java*. It contains all code needed to start or stop the plug-in. It is not necessary to change anything in this Java class.

For each extension we defined, Eclipse needs a Java class. This class will be called when we use this extension. For example: We have defined a popup menu and we are using it. The plug-in will call the class which is associated with this action. In this class we have to define the action for this extension.

5.2 Hibernate

To store our generated FAMIX model and the metrics of a Java project, we use Hibernate. Hibernate is an object/relational persistence and query service. This tool helps to relieve the developer of a large number of common data persistence related programming tasks. Using XML descriptors, Hibernate provides an object-oriented view of a relational database [Ive05].

In order to work, Hibernate needs different components:

1. Common Java classes.
2. A Mapping file for each Java class we need to store/read data.
3. A configuration file.
4. A relational database such as MySQL or PostgreSQL.

5.2.1 Java classes

Java classes provide he basis for mapping with Hibernate. A class does not need any specific attributes or methods to save a class instance with Hibernate. The class has to implement only the getter and setter methods for each attribute which we need to store. With these setter and getter methods, Hibernate is able to read and write all the necessary information.

Hibernate is not bounded to simple attributes such as Integer or String. It is also possible to map information stored for example in a HashSet. For this purpose Hibernate needs the interface of the class HashSet.

```
public class Package {

   // The attribute and collection to store.
   private String packageName = "";
   private Set<Class> classes = new HashSet<Class>();

   public void Package() {
      super()
   }

   public Set<Class> getClasses() {
      return classes;
   }

   public void setClasses(Set<Class> classes) {
      this.classes = classes;
   }

   public String getName(){
      return name;
   }

   public void setName(String name){
      this.packageName = name;
   }

   public void addClass(Class cl) {
      this.classes.add(cl);
   }
}
```

Listing 5.1: Example of a Java class for Hibernate.

Listing 5.1 shows an example for a Java class with information which we need to store in or load from the database. We have the simple attribute *packageName*, which is a String, and we have a HashSet *classes* which is a set of the class *Class*. To store the information of this class *Class* it is necessary to have all the getter and setter methods in *Class* so that Hibernate can access the attributes in this class. In our example these are the methods:

- getName()
- getClasses()
- setName(String name)
- setClasses(Set<Class> classes)

5.2.2 The Hibernate mapping files

But how does Hibernate know which attributes or collections we need to store and how they are labelled? For this purpose, each class we handle with Hibernate needs its own mapping file. These mapping files are labelled in the following manner: *class_name.hbm.xml*. If we follow our example from section 5.2.1, our mapping file is labelled *package.hbm.xml*.

These files describe the collaboration between the Java classes themselves and between the Java classes and Hibernate. So we can find the following information in this mapping files:

1. All properties of a Java class which are needed to store in or read from a database.
2. Inheritances between classes
3. Associations between classes

We shall to follow our example from 5.2.1 and show how a mapping file is made.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="seal.famix.model">

  <!-- Package mapping -->
  <union-subclass name="Package" extends="Context" table="package">

    <!-- Associations -->
    <many-to-one name="belongsTo" column="parent_id"
    class="Context" cascade="none" />

    <!-- 1-n package container relationship -->
    <set name="packages" inverse="true" cascade="save-update">
      <key column="parent_id" />
        <one-to-many class="Package" />
    </set>

    <!-- 1-n class container relationship -->
    <set name="classes" inverse="true" cascade="save-update">
      <key column="parent_id" />
        <one-to-many class="Class" />
    </set>
```

```
  </union-subclass>
</hibernate-mapping>
```

Listing 5.2: Example of a mapping file.

A mapping file implements a certain structure. This structure is defined in the DTD, the Document Type Definition. To know which DTD is used, the mapping file has to declare it. This is the first definition in a mapping file.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

Listing 5.3: The DTD in a mapping file.

To find a class, Hibernate needs to know the path to the class. To do this, we can set the optional attribute *package="..."* which defines the complete path.

```
<hibernate-mapping package="seal.famix.model">

<!-- place for the mapping -->

</hibernate-mapping>
```

Listing 5.4: The mapping with the path to the mapped class.

It is necessary to define how a class has to be mapped. For this case there are three different strategies. Each strategy has a number of advantages and disadvantages. To imagine this we use an example with the superclass *Animal* and the derived classes *Cat, Cow* and *Horse*:

- **Table per class hierarchy:** This strategy needs only one table in a database for *Animal* and *Cat, Cow, Horse*. The really big disadvantage is that a column declared in a subclass may not have a *NOT NULL* constraints.
- **Table per subclass:** This strategy needs many more tables than the table per class hierarchy. For the example with *Animal, Cat, Cow, Horse* we need four tables in a database. The three derived classes have a primary key association to the superclass.
- **Table per concrete class:** This strategy also needs a large number of tables in a database, for each of the derived classes *Cat, Cow, Horse* exactly one table, but no table for *Animal*. A major problem is that a property declared in the superclass needs the same column name in all subclasses.

Listing 5.5 shows the strategy with a table per concrete class. We have chosen this strategy for the FAMIX export plug-in and the Metrics export plug-in. The reason is that with this strategy we need more tables in the database but it is more flexible than with the table per class hierarchy. So we are able to load only the necessary information, using the visualization tool. We are not forced to always load more than the needed class.

```
<union-subclass name="Package" extends="Context" table="package">
  <many-to-one name="belongsTo" column="parent_id"
  class="Context" cascade="none" />
```

```
</union-subclass>
```

Listing 5.5: Defining a union-subclass relation.

Many-to-one is an ordinary association to another persistent class. The foreign key referencis the primary key of the target table.

Listing 5.6 shows a relation ship between two classes. In this example we have a *one-to-many* relation. In this case, a *Package* contains zero, one or more instances of a *Class*. The foreign key also references the primary key of the target table.

```
<!-- 1-n class container relationship -->
<set name="classes" inverse="true" cascade="save-update">
  <key column="parent_id" />
  <one-to-many class="Class" />
</set>
```

Listing 5.6: Defining a one to n relationship.

5.2.3 The Hibernate configuration file

The classes are developed and the mapping files are written. But how can Hibernate know where the mapping files are and how they are labelled? And what about the database?

For this case Hibernate has a further file, the *hibernat.cfg.xml*. In this file all the information about the mapping files and the database connection is stored. The first element in the *hibernat.cfg.xml* file is the DTD declaration. This DTD is only a little bit different from the DTD in the mapping files. It is not the *Hibernate Mapping DTD 3.0* but the *Hibernate Configuration DTD 3.0*. The next element is the tag for the session factory. This session factory will be used to create a session for a new connection to the database. Listing 5.7 shows the configuration file for our example from section 5.2.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.sourceforge.net/
       hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.username">User name</property>
    <property name="connection.password">Password</property>
    <property name="connection.url">DB URL</property>
    <property name="dialect">SQL dialect</property>
    <property name="connection.driver_class">Driver class</property>

    <!-- FAMIX mapping -->
    <mapping resource="seal/famix/model/mappings/Package.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Listing 5.7: The configuration file for Hibernate.

With the *property name="..."* tags it is possible to define all parameters for the database connection. In this case hibernate gets the following information:

- The user name to connect to the database.
- The password if a password is needed to connect to the database.
- The complete URL of the database. (Example: jdbc:mysql://localhost/famix)
- The SQL dialect of the database. (Example: org.hibernate.dialect.MySQLDialect)
- The JDBC driver to connect the to database. (Example: com.mysql.jdbc.Driver)

There are several other possible optional properties to set. For example it is possible to define if Hibernate has to show all SQL statements, if it has to use a reflection optimizer and so on. More details are described in the Hibernate manual [Sys]

The disadvantage of this method is that Hibernate can open a connection only to this database. If another database has to be connected, the *hibernate.cfg.xml* has to be adapted. To remove this disadvantage it is possible to configure the connection settings dynamically in a Java class. With this method of creating a session factory it is possible to change a database more easily than by adapting the configuration file. This method is shown in Listing 5.8.

```
Configuration cfg = new Configuration();
  cfg.setProperty("hibernate.connection.url","DB URL");
  cfg.setProperty("hibernate.connection.username", "User name");
    cfg.setProperty("hibernate.connection.password", "Password");
    cfg.configure("/hibernate.cfg.xml");

    // Creates the new session
    Session session = cfg.buildSessionFactory().openSession();
```

Listing 5.8: The alternative configuration of a session factory.

To give Hibernate the information about the mapping files, there is the *mapping resource="..."* tag. This entry defines the complete path to the mapping file, including the file name.

With all this information, Hibernate is ready to work.

5.2.4 The database schema

The Java classes are ready, the mapping and configuration files are written to map the Java classes and to connect the database. So it remains to prepare the database. This is the easiest step of all. The mapping files have the same structure as the tables in the database. For this reason Hibernate is able to create all the tables automatically. With an additional tag, see listing 5.9, in the *hibernate.cfg.xml* file, it is possible to enable Hibernate to create the tables. Table 5.1 shows the generated database schema of a table.

```
<property name="hbm2ddl.auto">create</property>
```

Listing 5.9: The auto creation of tables.

In this table of the example from 5.2.1 *parent_id* and *model_id* are two foreign keys: *parent_id* has to be another package id.

column	type	modifiers
id (primary key)	BIGINT(20)	not null
name	VARCHAR(255)	
parent_id	BIGINT(20)	

Table 5.1: Exported database schema of the Example from section 5.2.1.

5.3 The FAMIX export plug-in

To generate and export the FAMIX model we implemented an Eclipse plug-in. This plug-in uses the AST[3]-parser *patviz* to generate the FAMIX model and Hibernate to store the data in a database.

5.3.1 The plug-in features

The plug-in provides a wizard which guides users through the process. To start the wizard, a Java project must be selected. In the context menu of this project there is a new entry labelled *Parse AST to FAMIX*. The wizard needs a number of inputs and then it starts the parsing. The main tasks of the wizard art to:

- read in the information about the database (URL, username password).
- read in the information about the selected program (name and version).
- create a new database connects an existing database.
- start the FAMIX parser.

To run this plug-in, Eclipse 3.0 or later, J2SE 5.0 (Java 1.5) and the MySQL database 5.0 or later are needed.

5.3.2 Structure of the FAMIX export plug-in

To provide the service of this FAMIX export plug-in the following main components are needed:

- The AST parser
- The FAMIX model
- The Hibernate mapping
- A wizard to choose or create a database

These components are embedded in several packages. For a better understanding of the structure, a short introduction about the different packages of the plug-in will be helpful.

- **patviz.*:** This package contains the parser to parse the Eclipse AST to the FAMIX meta model. This is the heart of the plug-in.
- **seal.famix:** The FAMIXInstance in these packages contains all containers for parsed classes, methods, attributes. And this FAMIXInstane also contains a model which we will describe in the package *seal.famix.model*.

[3] AST: Abstract Syntax Tree

- **seal.famix.model:** This contains the extended FAMIX model. We have described this extension in 4.2.4. This model stores the parsed meta model, generated from the parser in the *patvix.** packages. This package also contains a class named *model*. This *model* contains information about the parsed system, the exported program (name, version) and so on.
- **seal.famix.container:** These are the containers which are used in the class FamixInstance. These containers provide methods to add parsed information and to search for parsed information. There are containers for: *Classes, Methods, Packages*. To resolve bindings these packages also provides the class *Claim*.
- **seal.famix.mappings:** Here we find the mappings for the FAMIX classes.
- **seal.famix.wizard.*** These are the packages for the wizard with all the classes needed for the wizard which guides the user through the process of creating or selecting a database. It collects the information about the program and starts the parsing of the selected Java project.
- **seal.AST2FAMIX.* and seal.famix.plugin:** Contains all classes which are needed to start the plug-in.

5.3.3 Design

The design of the our FAMIX model is very similar to the design of the FAMIX model used in the Coogle [Sag06] plug-in. The difference consists of the new classes *Inheritance, Generalization* and *Subtyping*. As described in 4.2.4, these classes help to ease the handling of the inheritance and the interfaces during mapping with Hibernate. Figure 5.3 shows the new classes, added to the FAMIX model. The complete FAMIX model is shown in figure C.1 in the Appendix.

5.3.4 Data flow

The flow of the data is easy to understand. First, we have a Java project. Eclipse loads this project and generates the AST. Then the parser gets this AST and generates the FAMIX model. Hibernate gets the data from this model and stores it in the database. Or if we want to use an existing FAMIX model, Hibernate reads the data from the database and generates the model by loading the data from the database. Figure 5.4 illustrates this simple data flow. The marked areas are our plug-ins.

The data flow between the FAMIX model and the database is a two way flow, which means a newly generated FAMIX model can be stored in a database or an existing FAMIX can be loaded from a database.

5.3.5 Saving the data

To save the generated FAMIX model, we use Hibernate. For each class instance we have to save, we need the respective mapping file. Because of the choice to use the strategy "A table per concrete class", we have more classes and mapping files than tables in the database. In order not to lose data, we use the inheritance in the mapping files. Listing 5.10 shows an inheritance in the mapping files. With this mechanism it is possible to store all information without using a table for each class.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

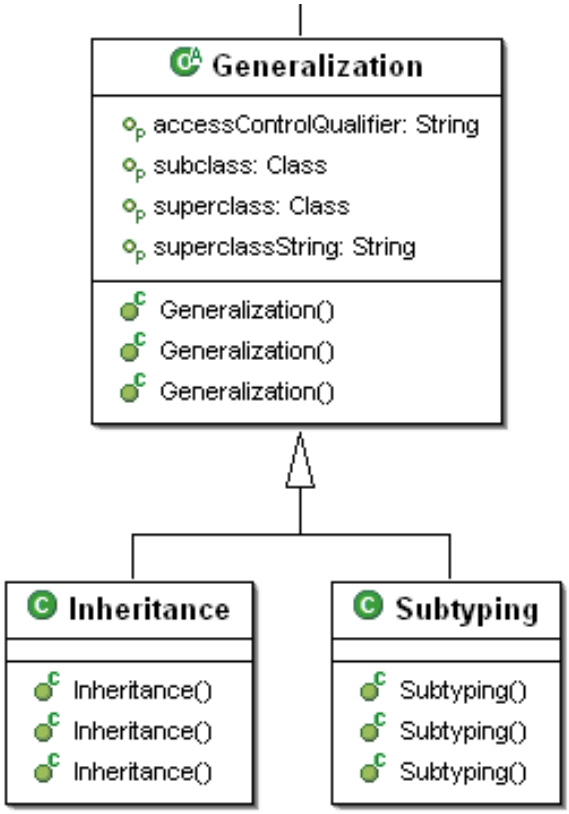

Figure 5.3: Details of the new classes.

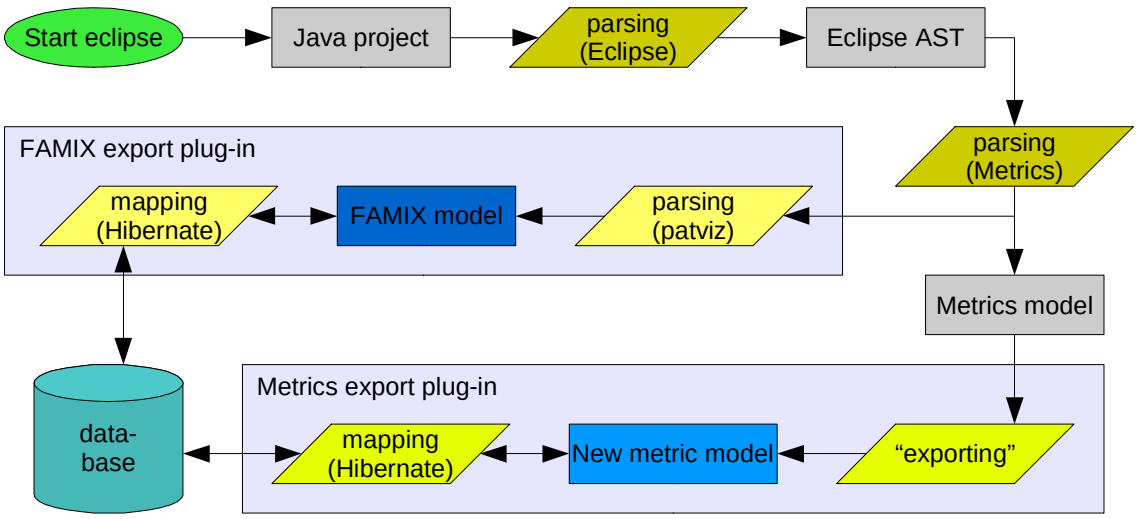

Figure 5.4: The data flow of the two plug-ins.

```
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="seal.famix.model">
  <union-subclass name="Context" extends="Entity" abstract="true">
  </union-subclass>
</hibernate-mapping>
```

Listing 5.10: Inheritance in the mapping files.

5.3.6 Problems and future work

During our work with this FAMIX parser plug-in we detected that the parser is very slow and that the FAMIX model needs a large amount of memory. To improve the speed, we assume that a solution would be to reengineer the resolving of the claims. These claims are unresolved objects like methods, classes and so on.

5.4 The Metrics export plug-in

The Metrics plug-in generates several metrics of a Java project. To store these metrics, we have implemented the Metrics export plug-in.

5.4.1 The plug-in features

The plug-in also provides a wizard which guides user through the process like the FAMIX export plug-in. To start the wizard, a Java project must be selected. In the context menu of this project there is a new entry labelled *Export Metrics*. The wizard needs a number of inputs and then it starts the export of the metrics. The main tasks of the wizard are to:

- read in the information about the database (URL, username password).
- read in the information about the selected program (name and version).
- check if the tables needed have already been created. If this is not the case, it creates the needed tables.
- start the metrics export.

Like the FAMIX export plug-in, this plug-in also requires Eclipse 3.0 or later, J2SE 5.0 (Java 1.5) and the MySQL database 5.0 or later.

5.4.2 Structure of the Metrics export plug-in

The Metrics export plug-in is a relatively small plug-in. We can see this in the structure of the plug-in. It contains fewer packages and classes than the FAMIX export plug-in. The main components of this plug-in are:

- The exporter, based on the *MetricsFirstExporter* of the Metrics plug-in.
- The metrics model.

- The wizard to choose the database.

To give an overview, the most important packages are described in the following lines.

- **metricsexporter.exporter** This package contains the class, which gets all metrics from the Metrics plug-in. The exporter fills in this data in our own simple data model.
- **metricsexporter.model** Our data model is in this package. This model is only for interim storage. We need this step in order to use Hibernate. Hibernate is not able to get data from the Metrics plug-in directly so that we are forced to engage this intermediate step.
- **metricsexporter.mappings** This package contains all the Hibernate mapping files, the mapping files for this plug-in and also the mapping files of the FAMIX export plug-in.
- **metricsexporter.wizard.*** For the sake of comfort, this plug-in also has a wizard which guides the user through the process. These packages contain all the classes needed.

5.4.3 Data flow

The data flow of the Metrics export plug-in is also simpler than the data flow described in 5.3.4 of the FAMIX export plug-in. The Metrics plug-in parses the Eclipse AST and computes the different metrics. The class *exporter* gets this data and creates our own data model. Afterwards Hibernate saves this model in the database.

5.4.4 Design

The design of the Metrics export plug-in is very simple. For this reason no pattern was used. To get the metrics from the Metrics plug-in, there is one class, the *exporter*. This class gets the metrics from the plug-in and stores them in our own data model. This data model consists of three new classes to store all information. The first class contains the descriptions of these metrics:

- the name of the metric.
- the short name of the metric.
- the type of entity which can be measured (possible entities: package, class, method).

In the preferences of the Metrics plug-in it is possible to define ”Save Ranges”. If a metric value is outside of this ”Save Range”, the Metric plug-in gets a warning. The second class in our own data model stores the values of this ”Save Range”. But this class also contains other computed values of a metric. The different values are:

- The minimum and maximum of the ”Save Range”.
- The computed total, average and standard deviation of a metric.

The third class contains all values of the metrics. In this class all the information concerning the measurement of a package, class or method is stored. This information comprises:

- The computed values of a measured entity.
- The name of the entity.
- The kind of metric (a link to the metric description).

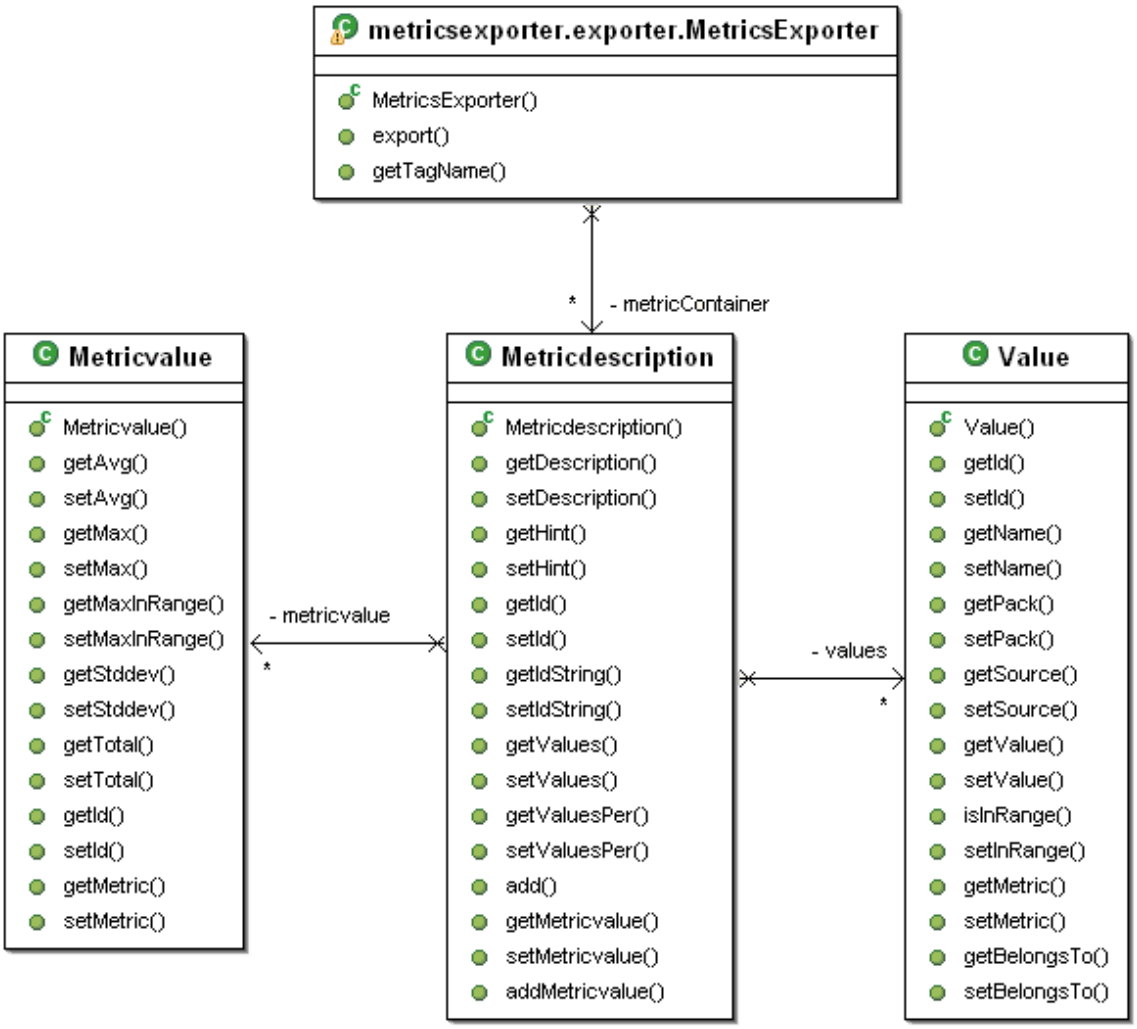

Figure 5.5: The complete model with the exporter of the Metrics export plug-in.

Figure 5.5 shows the complete model with the exporter and the data model.

For the sake of independence, we have also decided that the Metrics export plug-in should know the FAMIX model, so that it is possible to map the metrics with the correct classes, methods or packages. In this way a metric knows its owner. But we don't want to change the FAMIX export plug-in and the FAMIX model. So a class, method or a package doesn't know any metrics.

5.4.5 Saving the data

Saving this data with Hibernate is not complex. Only three mapping files are needed for this data model. No inheritance exists, so that only associations have to be handled.

To compare the classes, packages and methods from the Metrics plug-in and the FAMIX model and allocate the metrics, it is necessary to load a part of the FAMIX model. For this purpose the FAMIX mapping files are also needed.

To implement this process, the Metrics export plug-in has to load the right class, method or package for each value. Then the plug-in adds this loaded class to the value and stores the value in the database.

5.4.6 Problems and future work

During the implementation we founded several problems concerning the Metrics plug-in. The problems are: When the FAMIX export plug-in parses a project, a method afterwards has the structure: **Method_name(List of arguments)**. With this procedure it is possible to differentiate between the method and the overloaded methods. But methods exported from the Metrics plug-in have only the structure **Method_name**. So it is not possible to differentiate between a method and the overloaded methods. At the moment it is not always possible to map the exported metrics with the correct method. Figure 5.6 illustrates this problem.

The second problem is also a mapping problem. The Metrics plug-in exports the methods of an inner class incorrectly. This means a method which belongs to an inner class has the class which contains the inner class and not the inner class itself as its parent. But the FAMIX export plug-in parses an inner class correctly. So it is not possible to map these two methods.

To illustrate this circumstance, listing 5.11 shows an example of a class with an inner class. As a comparison, listing 5.12 shows a part of the output from the Metrics plug-in. We can see that the packages are correct. But the source (parent) of the method *myMethod* should be *InnerClass* and not *MyClass*. This is incorrect.

```
package myPackage;

public class MyClass {

  public MyClass(){}

  public class InnerClass{

  public InnerClass(){}
    private int number = 0;

       public void myMethod(int number){
          this.number = number;
```

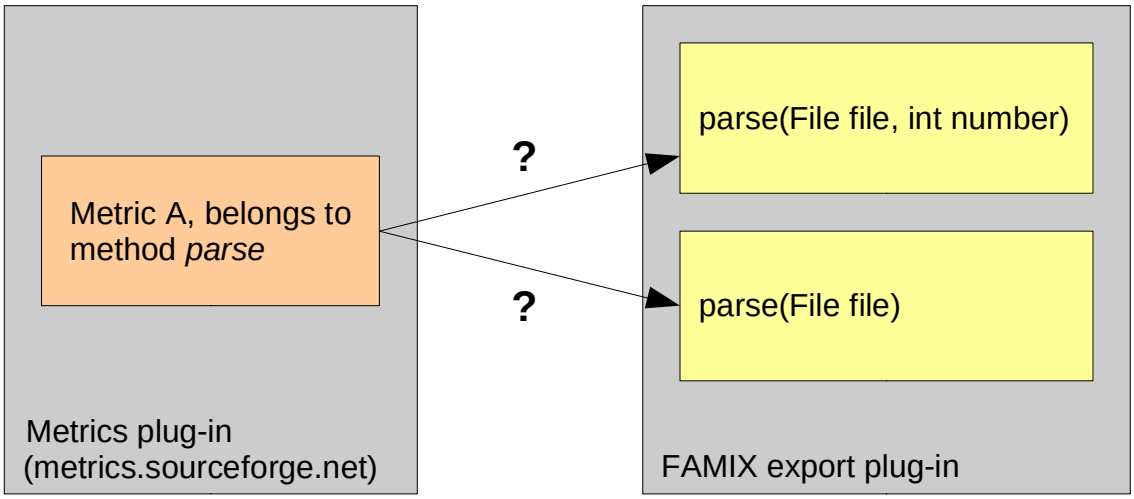

Figure 5.6: The correct mapping of method values from the Metrics plug-in is a problem.

```
    }
  }
}
```

Listing 5.11: Example of a class for the export with the Metrics plug-in.

```
<Metric id="MLOC" description="Method Lines of Code">
  <Values per="method" total="1" avg="0.333" stddev="0.471" max="1">
    <Value name="myMethod" source="MyClass.java"
         package="myPackage" value="1" />
    <Value name="InnerClass" source="MyClass.java"
         package="myPackage" value="0" />
    <Value name="MyClass" source="MyClass.java"
         package="myPackage" value="0" />
  </Values>
</Metric>
```

Listing 5.12: Example of the output generated width the Metrics plug-in.

The easiest way to solve these problems is probably to fix the Metrics plug-in. For example it is necessay to change the parsing of the methods, so that a method signature also contains the list of arguments and not only the method name.

Chapter 6

Evaluation

In this chapter we describe the evaluation of our implemented FAMIX and Metrics export plug-ins with a Java application.

6.1 Procedure

We evaluate our implemented plug-ins with the FAMIX export plug-in itself, especially with the parser patviz and the FAMIX model. We had planned to use the open source software ArgoUML [1] for this task, but we were not successful in parsing this project. The problem was that the parser (patviz) is very slow when it parses a large Java project. So we tried to parse the ArgoUML but after twenty four hours the parser was very, very slow. It parsed a class and after the next twelve hours it was parsing still the same class. Other tries with different memory settings of Eclipse were no better. So we have decided to use the FAMIX export plug-in for this evaluation instead of ArgoUML.

In this evaluation we execute the following steps:

1. We parse the FAMIX export plug-in to the FAMIX model and export it to the database.
2. We measure the plug-in with the Metrics plug-in.
3. We map the generated metrics with the FAMIX model and export it to the database too.
4. We use the Kiviat Visualizer to generate different graphs.
5. We answer the research questions, defined in chapter 3.

6.2 Preparing the data for the evaluation

The basis for this evaluation is a database with all the data needed. To create this data, we use our FAMIX export and Metrics export plug-ins.

In a first step we parse the FAMIX export plug-in itself. To do this, we have to start the plug-in and then we chose the complete Java project and start the parsing with the menu "Parse AST to Famix" in the context menu of the project. After this process we generate the different metrics with the Metrics plug-in. Afterwards we select the Java project again and start the Metrics export plug-in with the menu "Export Metrics" in the context menu. The plug-in exports the metrics

[1] The ArgoUML project: http://argouml.tigris.org/

generated with the Metrics plug-in, maps the metrics with the FAMIX model and stores the values in the database.

We are ready to start the visualization with the Kiviat Visualizer.

6.3 Answering the research questions

6.3.1 A package view

What can a package say about software quality?

A Java project is divided into several packages. A package is a construct like a folder and it contains classes or other packages. We start with our analysis at this level. This level provides a good opportunity to get a first impression of a software system.

For our analysis we have chosen all the packages from the parser and the FAMIX model. During our evaluation we analyse these packages and a part of the classes in these packages. We have chosen these packages because they enable us to compare it with an older version.

To answer this question, we have selected the following packages:

- container
- famix
- javarsf
- model
- util

The packages *seal* and *patviz* are root packages and contain no classes, so we don't have to analyse these two packages.

If we have a look at figure 6.1 we can see that we have four packages with relatively high metrics. This are the packages *container, famix, javarsf* and *model*. The packages *famix* and *model* have a high Afferent Coupling (CA). To have a high CA is undesirable because with a high CA there are many classes outside this package which depend on the classes in these packages. All four packages on the left side also have a high Efferent Coupling (CE); classes from these packages depend on the classes in other packages.

These two metrics are a hint, that it could be essential to move a number of classes from a package to another package to reduce the CA and the CE.

With these diagrams we are also able to identify other information. We see that in the package *famix* there are numerous interfaces (NOI) and as consequence the abstractness (RMA) increases. To calculate the Normalized Distance (RMD), the RMA and the Instability (RMI) are used. So we can see that we have a high RMD in the package *container*. This is not desirable, because this is a hint of a bad package design.

The definition of RMD is [Mar94]:

$$RMD = |RMA + RMI - 1|$$

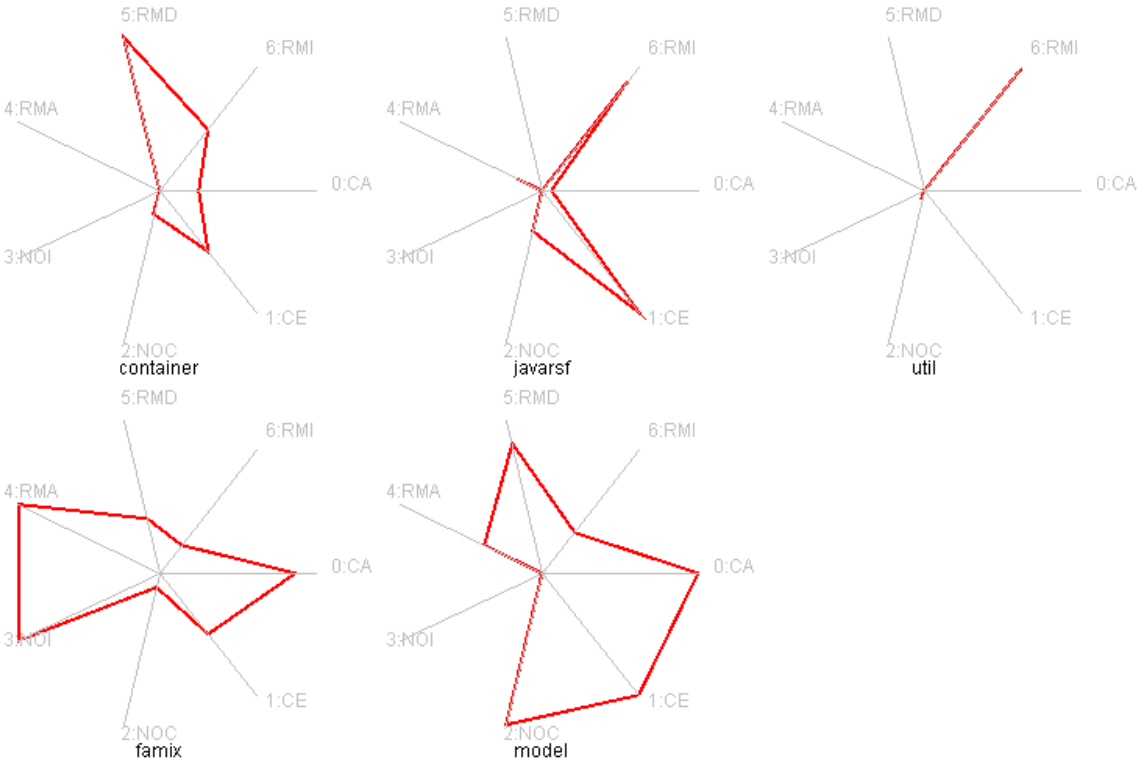

Figure 6.1: The metrics of the chosen packages.

Package	Number of classes
container	5
famix	5
javarsf	9
model	33
util	2

Table 6.1: Packages with the number of classes.

Problem

This graph gives us a good overview of the package level but a number of questions remain unanswered. When we look at the CE and NOC, we can see that we have the package *famix* with a middle CE and low NOC and the package *model* with a high CE and also a high NOC. Table 6.1 shows the packages and the number of the classes in them. With this information we can assume, but we are not sure, that almost all classes in the package *famix* must have a relatively high number of these dependencies. But we are not able to make a prediction as to how this is in the package *model*. We don't know if we have only few classes which have a high dependency or if we have many classes with a high dependency. We only know that the classes in this package depend on a high number of classes in other packages.

We have the same problem with the CA. For example we don't know on which classes in the package *famix* the classes from other packages depend. Is it only one class, or more? We only know that we have such a dependency.

Solution

It would be interesting to have an approximate number of the classes which are affected by the CA or CE. The FAMIX export plug-in and the Metrics plug-in have the same data as base and so it is possible to generate this number. The easiest way is probably to extend the Metrics plug-in. For that purpose it is possible to introduce new metrics, one which counts the classes which are the reason for the CA and one which counts the classes which are the reason for the CE.

With this new information it would be easier to decide if a package needs refactoring or not. With these numbers it would be possible to see, which of the thirty - three classes in the package *model* are liable for the CA and/or CE in this package.

It would be also interesting to visualize the dependencies between the packages with lines. The larger the line, the larger the dependencies between the packages. This would also help to identify candidates for possible refactoring.

6.3.2 Identifying large classes

Is it possible to find the large classes?

We go into the packages and analyse the classes. To get an overview of the size of these classes we first create a simple overview using rectangles. With this method we are able to identify the largest classes. To visualise these classes we have selected the number of methods (NOM) as width and the number of attributes (NOF) as height. Figure 6.2 shows the result of this view.

With this method we can identify the largest classes which are: *Class, BehaviouralEntity* and *Model*. Table 6.2 shows the number of methods and attributes of these tree classes. These classes have more methods and attributes than all other classes. This is a hint that a class could have many interactions with other classes. The large number of attributes is also a sign that a class has

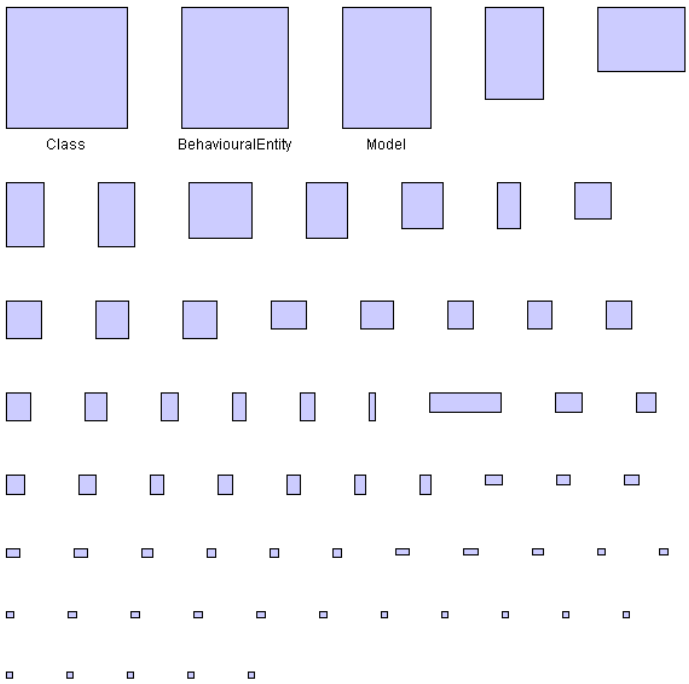

Figure 6.2: The view with NOM as width and NOF as height.

Class	Number of methods	Number of attributes
Class	44	13
BehaviouralEntity	39	18
Model	33	13

Table 6.2: Classes with the number of methods and attributes.

many interactions with the rest of the system. If a system is designed properly, these attributes can only be accessed with this class methods and not in another way.

Problems

The problem of a class with a high number of methods and attributes is that a class like this is harder to maintain. This is because of the (probably) high interaction with the rest of the system. If we change a method in this class, we have often to change other methods in other classes as well.

With this view we can identify problematical classes and this view provides us with a useful hint to look at these classes when we perform refactoring. Probably it is possible to reduce the number of methods and attributes if we are able to split a class with so many methods and attributes. But if the class has a certain function it is often not possible to split the class.

This view helps us to identify large classes, which means classes with a high number of methods and attributes. But we cannot see if such a class also has a large number of interactions with the rest of the system.

Solution To solve this problem we recommend choosing another view. A polymetric view with drawn dependencies helps to identify the large classes and it also helps to detect the interaction between the different classes. In this version of the Kiviat Visualizer it is unfortunately not possible to draw these dependencies. So we think that these enhancements will be a usful future project.

6.3.3 Identifying complex classes

Is it possible to detect complex classes?

To get a better overview of these tree classes, we use polymetric views. We again use the number of methods and the number of attributes. And this time we also use the Lack of Cohesion of Methods (LCOM). Figure 6.3 shows the view with these three metrics. To allow us to compare we have chosen the classes *BehaviouralEntity*, *Class*, *Model* and as the reference the class *Claim*.

We can see that the LCOM in our three classes is high. The compared class *Claim* has a smaller LCOM. We know that is it better to have a low LCOM, because a high LCOM indicates low cohesion. And if we have low cohesion, the complexity of a class increases. The goal of an implementation is to have classes with good cohesion.

A better way to find out something about the complexity is the metric Weighted Method per Class (WMC). This metric indicates the complexity of a class. A higher value indicates a higher complexity. If we also look at figure 6.3 we can see that when we have a high LCOM the WMC is often higher than zero. The high LCOM can be also a good sign for the existence of complexity in a class.

Problems

One Problem is that the LCOM indicates the complexity again, because it evaluates the cohesion. A further problem is, that a big LCOM is not automatically bad and an LCOM with zero is not automatically good. Why is this? This is a problem of this metric.

The definition of LCOM is [SRC94]:

$$LCOM = |P| - |Q| \; if \; |P| > |Q|$$

$$= 0 \; otherwise$$

P = number of pairs of methods which do not have a collective attribute.
Q = number of pairs of methods which have at least one collective attribute.

To illustrate this problem we create an example. Listing 6.1 shows a class with two attributes and four methods. To get the different pairs and the P and Q we create table 6.3. If we count the different methods we get $P = 3$ and $Q = 3$. When we calculate the LCOM the result is $LCOM = 0$! But we can see that this is not really true.

```
public class C {
  private int a;
  private int b;

  public int d() {
    return a;
```

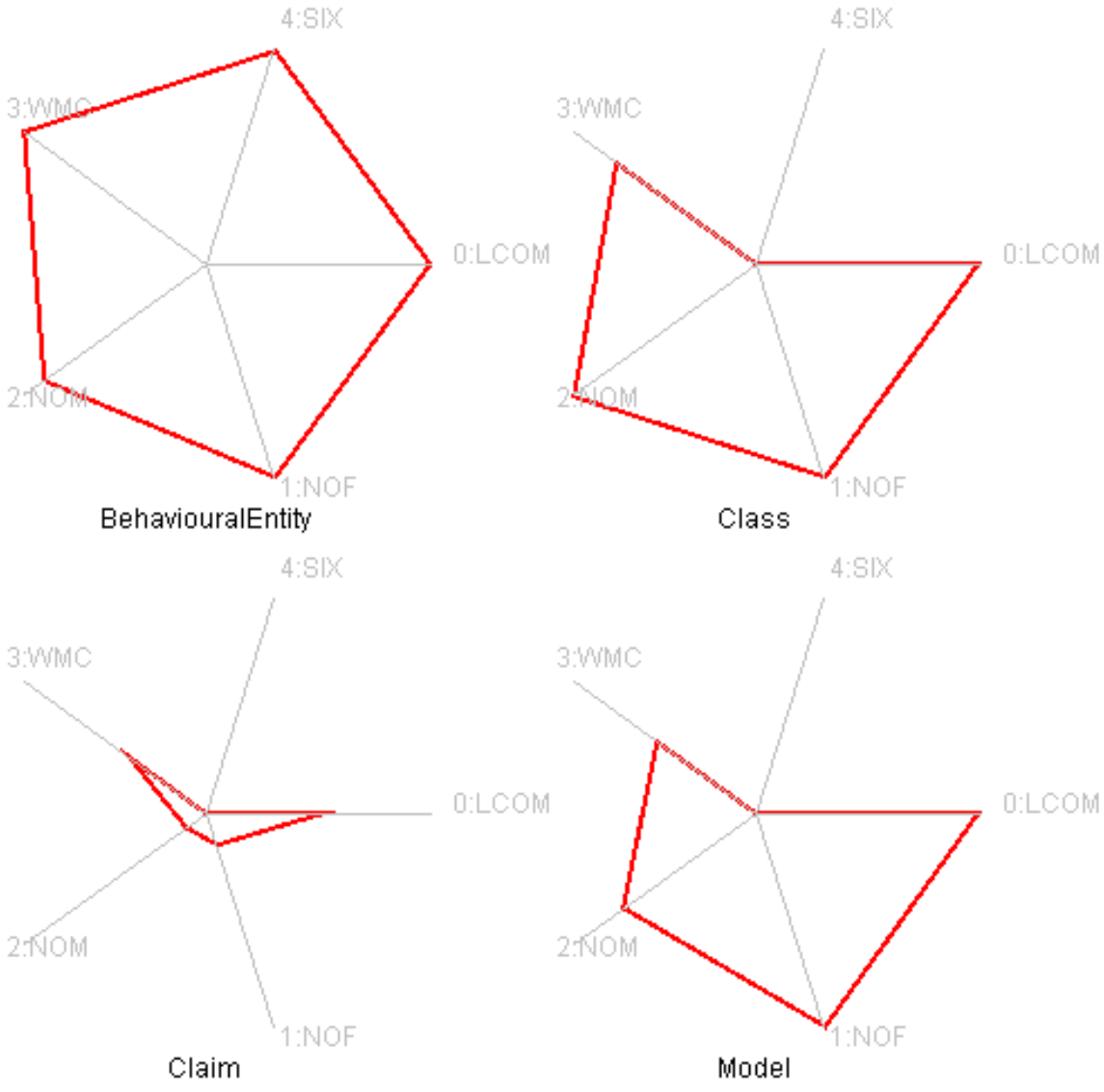

Figure 6.3: LCOM of the classes *Claim* and *Class*.

Pairs of methods	Collective attributes
d,e	none
d,f	none
d,g	none
e,f	b
e,g	b
f,g	b

Table 6.3: Methods and collective attributes.

```
  }

  public int e() {
    return b;
  }

  public int f() {
    return b * b;
  }

  public int g() {
    return b * b * b;
  }
}
```

Listing 6.1: Example class for the calculation of LCOM.

The other example is that an $LCOM > 0$ is not automatically bad. This may be the case when we have a class which serves as a data object. Then we have a getter and a setter method and they use the same attributes. But in this case this is not a problem.

A problem of the WMC is that this metrics are often used in different versions. A number of versions do not count constructors or overridden methods.

Solution

The problem with the use of the LCOM is, that we don't can see if a class with a high LCOM is a data object or a common class. It is necessary that we have to perform the following (manual) steps:

1. Identifying all classes with a high LCOM (and probably with a LCOM of zero).
2. Reading the documentation to identify data objects.
3. Removing the data objects from our list.
4. Having a short look at the rest of our list.

In this way we are able to identify more or less the classes with a really bad LCOM. During refactoring we are able to reduce the LCOM of this classes.

But we can avoid these steps when we use the WMC. This is the better indicator for a complex class. And to avoid the problems with this metric, it is enough to always use the same metrics tool to build the metrics.

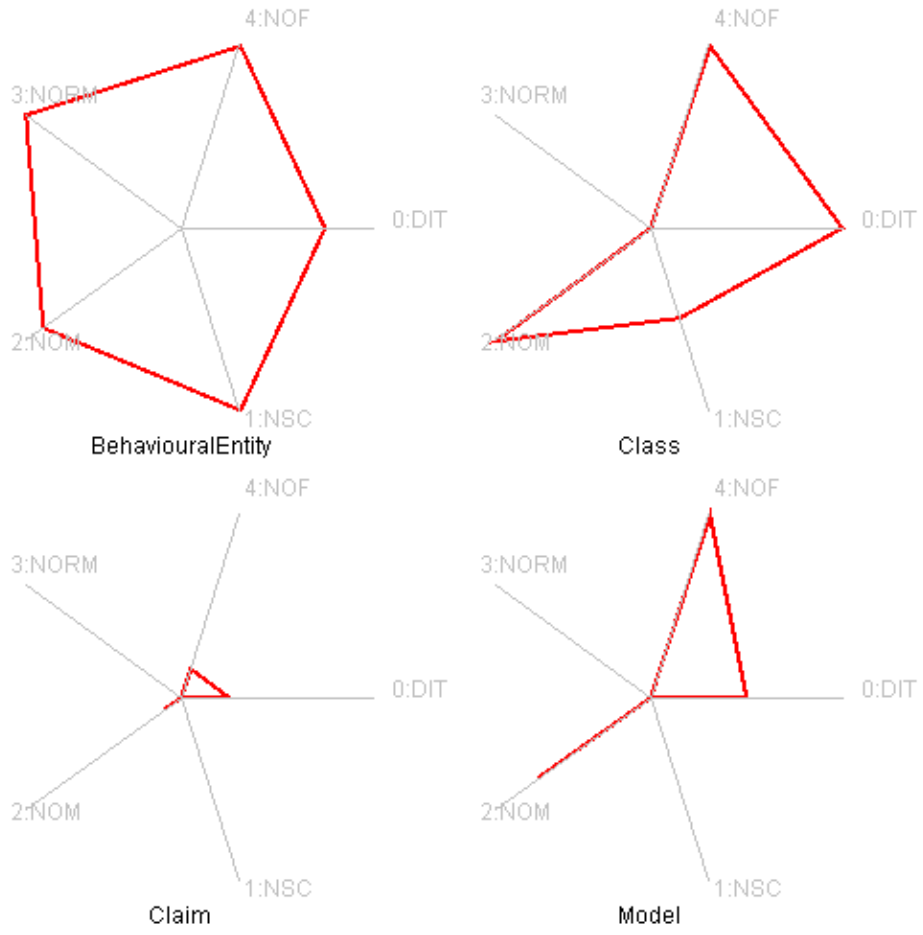

Figure 6.4: Metrics of the inheritance.

What can we say about inheritance?

With our tools we can visualise the metrics Depth of Inheritance Tree (DIT) and the number of Children (NSC). Figure 6.4 shows our four classes with these metrics and the metrics NOM, NORM and NOF as additional information to generate the graph.

We can see, that these classes have different DIT's. We know that these classes have the following inheritance trees; see figure 6.4. And we can also see, which class has how many children. To get an overview we can use table 6.4. This table shows us the values of these two metrics.

This two metrics can indicate a problem in the architecture of a software system. If the DIT is very long, it is possible that architecture of this software system could be bad and the system should be refactored. And for this reason is better to have a DIT which is not too long.

Problem

The problem with this view is that we don't know the exact value of the DIT. This means we are not able to say how many classes we have above an analysed class. We can only differentiate if a class is in a inheritance tree or not, but not the depth of this tree. To estimate if an architecture has problems or not, it is necessary to know the depth of a tree. The class with the maximum tree

Class	DIT (with java.Object)	NSC
BehaviouralEntity	3	2
Claim	1	none
Class	4	1
Model	2	none

Table 6.4: Classes with the DIT and NSC.

needs the complete line, but we don't know if this maximum has a value four, ten or even twenty.

The same is true about the NSC. We can only see if a class has many children or not. But we do not know the values again. The question is also if this class has four, ten or even more children. But we do not know this fact and we are not able to visualise these circumstances.

To expand our knowledge, we have to count the DIT or NSC manually.

And another problem with NSC is the metric itself. What is the meaning of a high NSC? A high NSC indicates that the class has good reusability. But it can also indicate that this could be an abuse of the inheritance. We do not which is true.

Solution

To solve this problem a possibility is to expand the function of the Kiviat Visualizer. It has to count the DIT and NSC and to display these values. Then it is easy to identify the classes with high values of DIT and/or NSC and to decide if it would be necessary to adapt the architecture of a software system or not.

In our example it was relatively easy to estimate that the maximum could be four. But when a software system has more classes, then it is not so easy to estimate the maximum number.

There is only one solution to the problem with the NSC. We have to examine the classes by hand to find out if a high NSC is caused by good reusability or by the abuse of the inheritance. Our tools can't do it for us.

But with this tool we can immediately identify the classes, which need an accurate examination.

6.3.4 A method view

What are the large methods?

We go into the classes and look at the methods in general. To identify the largest methods, we use the view showing all methods. We again use a view with rectangles. As metrics we have selected the Method Lines of Code (MLOC) as width and the Number of Parameters (PAR) as height. Figure 6.5 shows the result of this selection.

We can identify three different patterns:

1. We have a lot of methods with a low number of MLOC and PAR.
2. We have a lot of methods with a low number of MLOC but with a high number of PAR.
3. We also have methods with a relatively high number of MLOC and a high number of PAR.

If we look at the first type of this pattern, we can see that these methods are often getter and setter methods, constructors or static methods. The second type of this pattern also often consists of setter methods and of methods to add something (example: add Invocation). The third type of

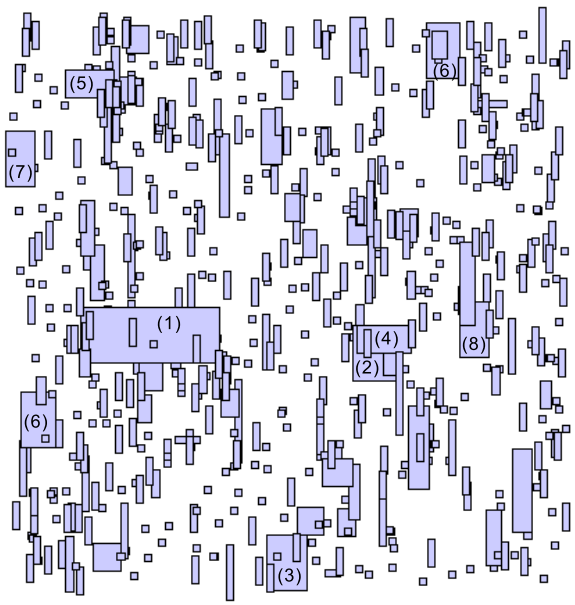

Figure 6.5: The different sizes of the methods.

this pattern consists of methods which compute something. But we can also find (a few) getter and setter methods.

With this view we can identify that the largest methods are:

- parse (1)
- parseFieldDecl (2)
- addAccess (3)
- setContent (4)
- print (5)
- fulfilClaim (6)
- getMethod (7)
- parse (8)

Problem With this view, we can identify the largest methods. And we can see that we have in this case three different types of patterns which classify the methods. The problem is that we also want to identify all important and not only the largest methods. That normally means that the setter and getter methods are not very interesting because they often look like the example in listing 6.2.

```
public void setName(String name){
  this.name = name;
}
```

```
public String getName(){
  return name;
}
```

Listing 6.2: Common setter and getter methods.

Often setter and getter methods don't do anything other than set and return a variable. So these methods are not very interesting to examine. It would be practical not to see these ordinary setter and getter methods.

A second problem is the fact that we can have different methods with the same name. If we look at our list with the largest methods, we can see that we have found two large methods with the same name: *parse*. But we don't know which method belongs to which class. So we have to check all methods by hand if we need to find out which methods could belong to which class.

Solution To avoid these problems, a possible solution is to extend the Kiviat Visualizer. To solve the second problem, a selection of classes is needed. We would then be able to select a class and we would get only a method list of the selected class or classes. Then the problem of methods with the same name can be reduced. But this approach helps only if we don't have overloaded methods in the same class. If we have overloaded methods, we are not able to differentiate between this methods. For this case it is probably better to display the entire method including the signature.

Example: we have a class with two methods with the name: show. At the moment, we have the following list in the selector:

- show
- show

With the entire signature we have this list in the selector:

- show()
- show(int number)

With this version it is easier to identify a method than with the first version.
To solve the problem with the setter and getter methods two approaches are available:

- The Kiviat Visualizer has to provide the possibility to suppress getter and setter methods. The user thus decides if these methods are important for him or not.
- Another is to suppress methods with a low number of MLOC. For example, the user can say that he wants to display only methods with an MLOC greater than three. A common getter or setter method has not more than three lines of code. See also our example in listing 6.2.

Can we identify the complex methods?

For the task "Identifying complex classes" we select all four available metrics on the method level. With the generated graph we choose to display the methods with high values; these are also the two largest methods, identified in the last task. Figure 6.6 shows this method with all four metrics.

We can see that we have the metric McCabe Cyclomatic Complexity (VG). This metric is a good sign for the complexity of a method. We know that a value over ten is not desirable [HS96] and if this is the case, refactoring of this method is recommendable.

The other metric which can indicate a probably complex method, is the Nested Block Depth (NBD).

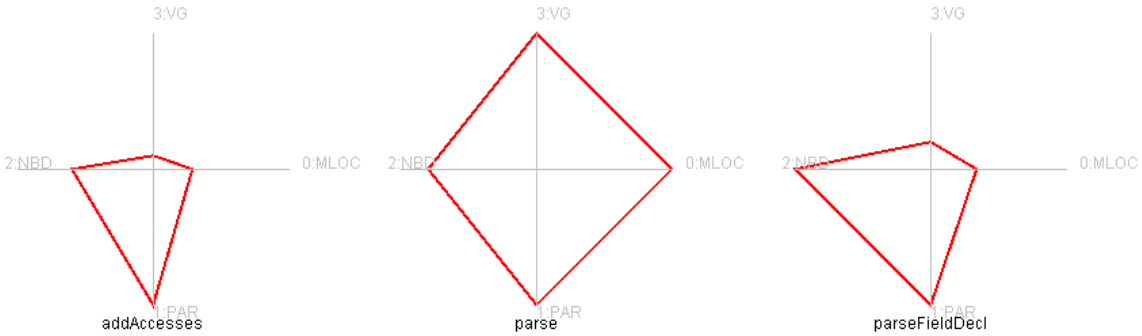

Figure 6.6: Searching for complex methods.

Problem The problem in this case is that we can see that we have a high VG, but as in the other views, we don't know what is the number of the VG. Is the number of VG four, ten, or more? We need a better basis to make a decision as to whether refactoring is needed or not. Otherwise the VG is probably relatively low but we think that refactoring is needed because the view "shows" us a high VG.

Solution To get more information it is desirable to show the values in the graph. With this values, the decision to do a refactoring or not, is easer to do. This values helps us to estimate if we have the critical fronter already passed or not. It is also possible to watch a number of endangered methods. So it is possible to react before the effort to do a refactoring rise extreme.

6.3.5 An evolution view

Can we say something about evolution?

Yes, it is possible to make a statement about the evolution. **But:** In this version of the Kiviat Visualizer it is not possible to display more than one release at the same time. So we are forced to create an individual graph for each release.

To display a metric, the Kiviat Visualizer normalises the value. For this reason it is not possible to make an exact statement. But when we view figure 6.7 we can see that something has changed. We don't know which of these three metrics has changed, but we can assume that at least the NOM doesn't have the same values in both releases.

That is all what we can say about the evolution. We can say that something has changed, but we are not able to say what has changed.

Solution To get more information it is necessary to expand the Kiviat Visualizer. It is essential that it can display more than one release at the same time. Because this involves a great deal of work, we didn't expand the Kiviat Visualizer for this thesis.

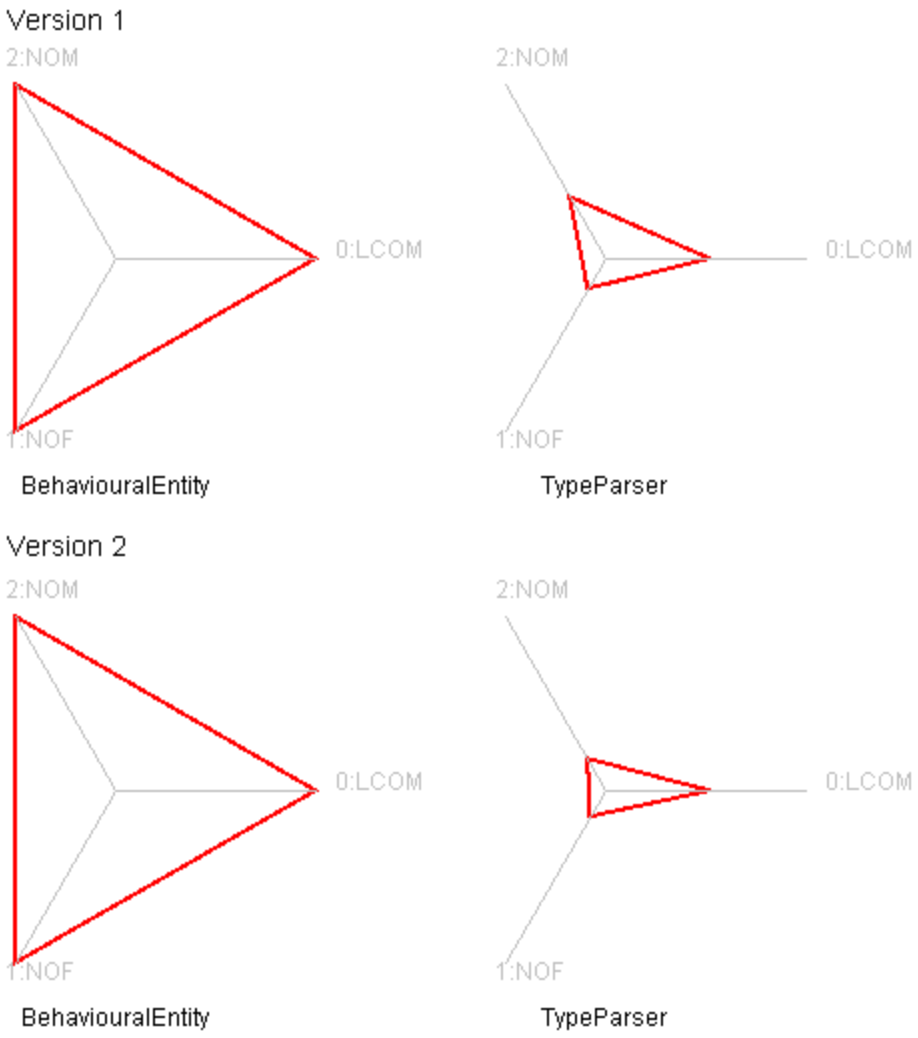

Figure 6.7: An attempt to visualise the evolution.

Chapter 7

Conclusion and future work

7.1 Conclusion

Software maintenance is an important task in the software cycle. To assist this task, many tools and methods have been developed. These tools are often bound to a certain language. Each language needs its own tools to do this work. An approach was to create a language-independent meta-model to share the data between different tools written in different languages.

For this thesis we used the FAMIX meta model which was designed as an object-oriented and language-independent model. We were able to use the FAMIX implementation of Coogle. This meant that we were not forced to implement the entire model, but only a number of adjustments to our requirements. To parse a Java Project into the FAMIX model we used the parser *patvitz* which was also used in Coogle.

This thesis describes the implementation of our two plug-ins and the subsequent evaluation. The first plug-in generates and stores the FAMIX model in a relational database. The second exports the generated metrics of the Metrics plug-in, maps the metrics with the FAMIX model and also stores it in the relational database. To store all the generated data we used Hibernate, an object/relational persistence and query service.

We learned that Hibernate is a powerful tool. But it requires considerable effort to learn the correct handling of this tool. Furthermore we found out obliged to realise that the parser is very slow and we therefore conclude that at the moment it is not possible to parse a really large Java project with this parser.

Our second tool, the Metrics export plug-in, works fine. But we also have a number of problems with it: not so much with the tool itself, but with the Metrics plug-in. This plug-in maps the methods of an inner class incorrectly and if a class has overloaded methods it is not possible to differentiate between these methods. Our Metrics export plug-in obtains data which is not absolutely correct or complete and cannot always map these metrics correctly.

After the implementation we created a complete FAMIX model of our FAMIX export plug-in and used it for the evaluation. To visualise the data we used the Kiviat Visualizer. With this tool we are able to draw different graphs, based on the generated metrics. In this evaluation we have also answered a catalogue of research questions.

Because of a number of technical limitations of the Kiviat Visualizer, not all questions have been answered satisfactorily. And unfortunately there was not enough time to adapt the Kiviat Visualizer. But we can say that the ArchView approach, on which this visualisation is based, is a promising approach. We were also able to learn a lot about the FAMIX export plug-in with a premature visualisation tool.

We recommend strongly to completing this approach and accomplishing all the tasks neces-

sary to ameliorate these tools.

7.2 Future work

To ameliorate these tools we recommend carrying out the following steps:

1. Refactoring the FAMIX parser to enhance the speed.
2. Correction of the wrong parsings and incomplete method generation in the Metrics plug-in (open source).
3. (Partially) amelioration of the Kiviat Visualizer as suggested in chapter 6, to obtain more significant graphs.

We are sure that with these corrections and adaptations the ArchView approach will become a powerful method to analyse software.

Appendix A

Abbrevitations

All abbrevitations of the metrics generated with the Metrics plug-in.

Metric	Description
CA	Afferent Coupling
CE	Efferent Coupling
NOC	Number of Classes
NOI	Number of Interfaces
RMA	Abstractness
RMD	Normalized Distance
RMI	Instability
DIT	Depth of Inheritance Tree
LCOM	Lack of Cohesion of Methods
NOF	Number of Attributes
NOM	Number of Methods
NORM	Number of Overridden Methods
NSC	Number of Children
NSF	Number of Static Attributes
NSM	Number of Static Methods
SIX	pecialization Index
WMC	Weighted methods per Class
MLOC	Method Lines of Code
NBD	Nested Block Depth
PAR	Number of Parameters
VG	McCabe Cyclomatic Complexity

Appendix B

Content of the CD

File	Description
FamixExport.jar	This archiv contains the source code of the FAMIX Export plug-in.
MetricsExport.jar	This archiv contains the source code of the Metrics export plug-in.
KiviatVisualizer	This archiv contains the source code of the premature Kiviat Visualizer
FamixExport.sql	This file contains the generated data of the FAMIX Export plug-in.
Famix.sql	This file contains the generated data of the FAMIX parser and FAMIX model (The version of Coogle).
SealPlatform.pdf	This document in Adobe Portable Document Format.
Abstract.pdf	The abstract of the thesis in English.
Zusfsg.pdf	The abstract of the thesis in German.

Appendix C

The complete FAMIX model

This chapter contains the complete extended FAMIX model. In this model are also listed not used classes. This is for the reason of completeness. The three classes are: *InterfaceRealization*, *InheritanceDefinition* and *TypeGeneralization*. Instead of these classes we have created the three classes: *Inheritance*, *Subtyping* and *Generalization*. How described in 4.2.4 the reason is to simplify the handling with hibernate. The three unused classes can be declared as deprecated.

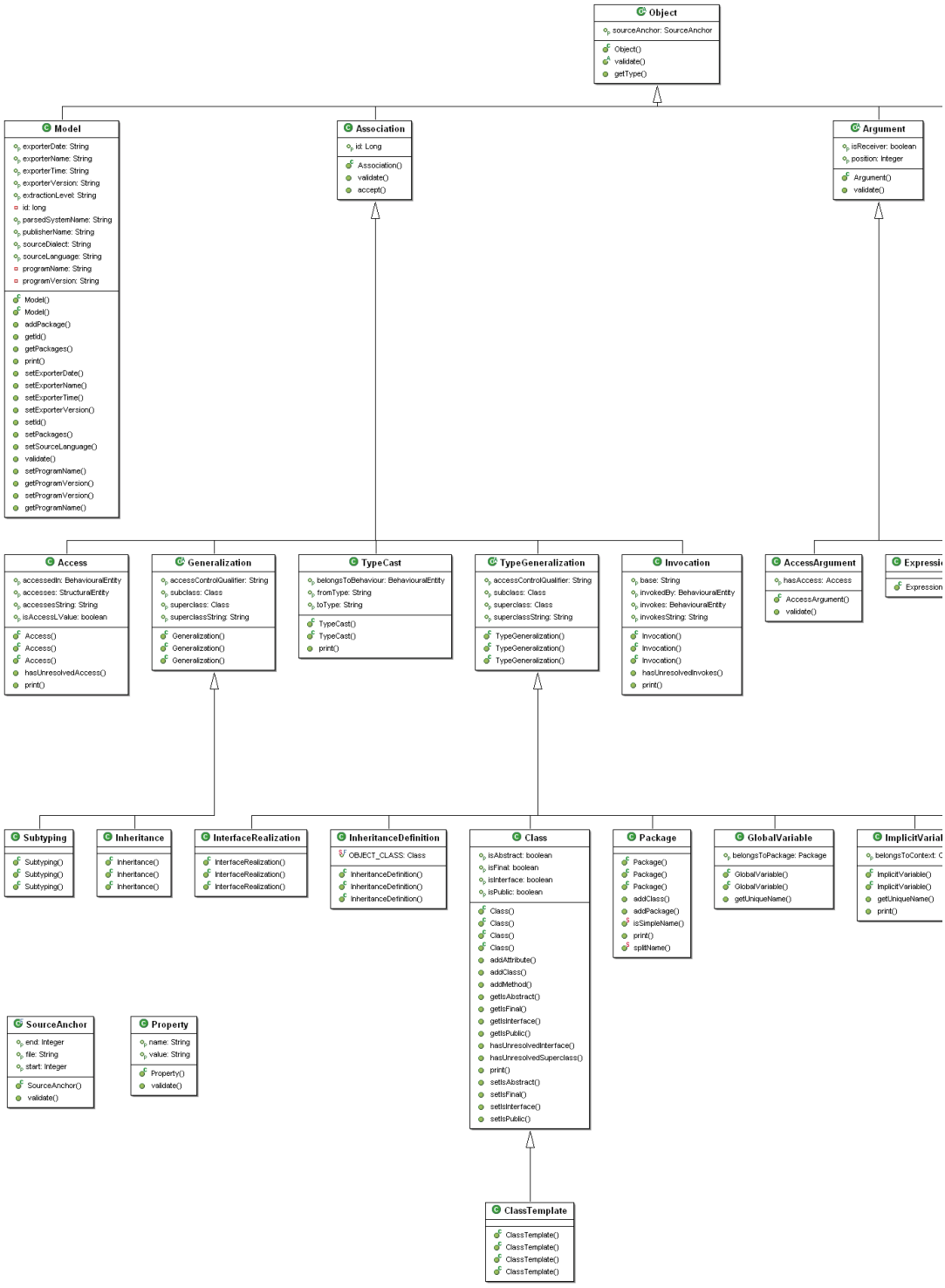

Figure C.1: The complete extended FAMIX model (Part 1).

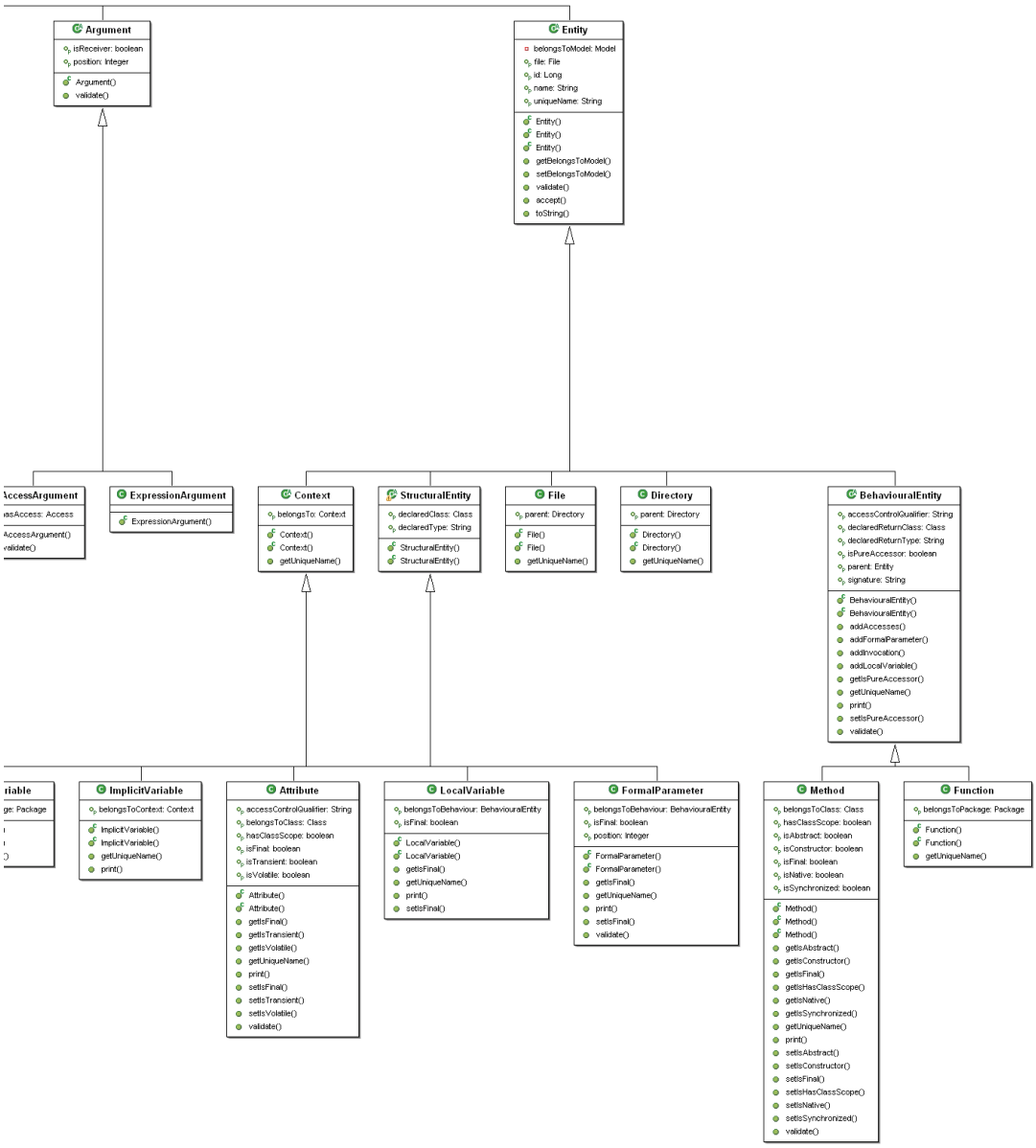

Figure C.2: The complete extended FAMIX model (Part 2).

References

[CR04] Tarja Systä, Jianli Xu, Claudio Riva, Petri Selonen. UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance. IEEE International Conference on Software Maintenance (ICSM'04). IEEE 1063-6773/04, 2004.

[EB00] John Davey, Elizabeth Burd, Steven Bradley. Studying the Process of Software Change: an analysis of software. evolution. Seventh Working Conference on Reverse Engineering 2000, pages 232 - 238, IEEE 1095-1350/00, 2000.

[EG04] Kent Beck, Erich Gamma. *Eclipse erweitern*. pages 35 - 36, Addison-Wesley, 2004.

[EJC90] James H. Cross II, Elliot J. Chikofsky. Reverse Engineering and Design Recovery: A Taxonomy. pages 13 - 17, IEEE Software 0740-7459/90/0100/0013, 1990.

[FdB] J. Jacob, A. Stam, L. van der Torre, F.S. de Boer, M.M. Bonsangue. Enterprise Architecture Analysis with XML. Proceedings of the 38th Hawaii International Conference on System Sciences - 2005, IEEE 0-7695-2268-8/05.

[Fou] Eclipse Foundation. *Welcome to Eclipse - Workbench User Guide*.

[GCG00] Robyn R. Lutz, Gerald C. Gannod. An Approach to Architectural Analysis of Product Lines. pages 548 - 557, Association for Computing Machinery, Inc. 1-58113-206-9/00/6, 2000.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics*. pages 92 - 99, Prentice Hall, 1996.

[Ive05] Will Iverson. *HIBERNATE - A J2EE Developer's Guide*. pages 1 - 15, Addison-Wesley, 2005.

[Lan02] Michele Lanza. The evolution matrix: Recovering Software Evolution using Software Visualization Techniques. Association for Computing Machinery, Inc., 1-58113-508-4/02/006, 2002.

[Leh97] M. M. Lehman. Laws of Software Evolution Revisited. Department of Computing, Imperial Collage, London, 1997.

[LV05] Jarke. J. van Wijk, Lucian Voinea, Alex Telea. CVSScan: Visualization of Code Evolution. pages 47 - 56, Association for Computing Machinery, Inc. 1.59593-073-6/05/0005, 2005.

[Mar94] Robert C. Martin. Oo Design Quality Metrics - An Analysis of Dependencies. Object Mentor Inc., 1994.

[MCC97] Stephen G. Eick, Mei C. Chuah. Glyphs for Software Visualization. Fifth Iternational Workshop on Program Comprehension 1997, pages 183 -191, IEEE 1092-8138/97, 1997.

[ML03] Stéphane Ducasse, Michele Lanza. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering. IEEE Transactions on Software Engineering, pages 782 - 795, IEEE 0098-5589/03, September 2003.

[Par92] B. Parker. Introducing EIA-CDIF: The CASE Data Interchange Format Standard. Proceedings of the Second Symposium on Assessment of Quality Software Development Tools, pages 74 -82, 1992. IEEE 0-8186-2620-8/92, 1992.

[Pin05] Martin Pinzger. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems.* PhD thesis, Vienna University of Technology, 2005.

[RCS03] Grace A. Lewis, Robert C. Seacord, Daniel Plakosh. *Modernizing Legacy Systems Software Technologies, Engineering Processes, and Business Practices*. page 10, Addison-Wesley, 2003.

[Sag06] Tobias Sager. Coogle - A Code Google Eclipse Plug-in for Detecting Similar Java Classes. Master's thesis, Department of Informatics, University of Zurich, 2006.

[SD99a] Patrick Steyaert, Serge Demeyer, Sander Tichelaar. FAMIX 2.0 - The FAMOOS Information Exchange Model. Technical report, Software Composition Group, University of Berne Neubrückstrasse 10, CH- 3012 BERNE, September 07, 1999.

[SD99b] Sander Tichelaar, Serge Demeyer, Stphane Ducasse. Why FAMIX and not UML? UML Shortcomings for Coping with Round-trip Engineering. Technical report, Software Composition Group, University of Berne Neubrückstrasse 10, CH- 3012 BERNE, 1999.

[SD05] Laura Ponisio, Stéphane Ducasse, Michele Lanza. Butterflies: A Visual Approach to Characterize Packages. 11th IEEE International Software Metrics Symposium (METRICS 2005), IEEE 1530-1435/05, 2005.

[SRC94] Chris F. Kemerer, Shyam R. Chidamber. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, pages 476 - 493, IEEE 0098-5589/94, 1994.

[Sys] JBoss Enterprise Middleware System. Hibernate reference documentation 3.0.5.

[Tic99] Sander Tichelaar. FAMIX Java Language plug-in 1.0. Technical report, Software Composition Group, University of Berne Neubrückstrasse 10, CH- 3012 BERNE, August 31, 1999.

[TR02] René Krikhaar, Tobias Rötschke. Architecture Analysis Tools to Support Evolution of Large Industrial Systems, 2002.