

TECHNISCHE UNIVERSITEIT EINDHOVEN  
Department of Mathematics and Computer Science

MASTER'S THESIS

**Automated Techniques for  
Reconstructing and Assessing  
Correspondence between  
UML Designs and Implementations**

by  
Dennis J.A. van Opzeeland

Supervisor:

Dr. M.R.V. Chaudron (TUE)

*Eindhoven, August 2005*



# Abstract

Over the years software systems have grown larger and larger. This trend will continue in the future. For many of these systems, the development does not stop once the system passed the acceptance test. Often, new features are added to existing systems at a later stage.

The developers that are responsible for the implementation of new features should understand the system before they are able to do their job. System understanding starts at a high level. It is very convenient if not inevitable to use architecture and design models for this. The implementation is far too detailed and voluminous for this purpose. Design models are only useful for understanding however, if the implementation corresponds to these artifacts.

It is important for an implementation to correspond to its design. In practice many software systems deviate from their design. Not all deviations are problematic but many systems deviate too much. In this thesis we will present techniques to assess the correspondence between a design and its implementation. We will present a classification of the differences between design and implementation which we found in several case studies. We will give an indication of the severity of these differences.

An object oriented software design consists of model elements. Model elements are for instance classes, operations or relations. The implementation also consists of model elements. These are small code fragments that represent the declaration of a class, the definition of an interface or the implementation of a method. Before we can determine how big the differences between design and implementation are, we need to recognize the design model elements in the implementation.

To express the similarity between two model elements, we use similarity measures. Similarity values are real values between 0 and 1. Similarity values can be defined for different aspects of a model element. The similarity of a class can thus be measured in different ways. The similarity values that we will discuss are based on the names of the classifiers, the packages in which the classifiers are defined and other structural aspects. These can be the set of operations, the set of attributes and relations with other classes.

We will discuss some techniques to establish a matching between design elements and implementation elements. We will find a matching at the level of classifiers. Using similarity values we can use a weighted bipartite matching algorithm for matching.

It is also possible to use software metrics for matching. In this approach we try to find a combination of metrics such that each class in a model is distinctive from other classes in that model. This combination is called a metric profile. It can be seen as a fingerprint for a class. Metrics can be calculated for both design and implementation classes. Metrics in the profiles must be correlating. Given the metric profile of a design class it is then possible to predict the metric profile of the implementation class that would match best to the design class. This predicted profile is compared with the metric profiles of real implementation classes. The difference between the real profile of a class and the predicted profile, can be expressed by a

similarity value again.

Once the matching is established, it is possible to identify differences. There are many different kinds of differences. Some differences are only small details. These differences are no threat to system understanding. It is therefore acceptable that these differences exist. We focus on differences that do compromise system understanding.

We expect the crucial differences to be found at outliers. Outliers are classes for which the metrics deviate very much from the correlation trend. We draw scatter plots of design and implementation metrics in order to find the outliers.

The techniques we present in this thesis have been implemented in some tools. We implemented a matcher tool that can be used to find the matching between design classes and their implementation. Furthermore we developed a scatter plotter tool that draws many different scatter plots in a single window. This tool helps us to identify the extreme outliers. There are also some other tools to import the required data into a database.

We applied our techniques on various industrial case studies. We performed an experiment where each of our proposed matching techniques was used to find a matching in various cases. This experiment clearly showed that matching on the names of classifiers was the best strategy. Metric profiles turned out to be unusable for matching.

# Preface

I started my study of Computer Science in 2000 at the Technische Universiteit Eindhoven. During the final project of the bachelor phase I became interested in Software Architectures. I was typically interested in how to create good architectures. Unfortunately there were not many courses about Software Architecture in the curriculum of my study.

For me it was clear that my ideal master project would be on the topic of software architectures. While browsing the internet I learned about the Empanada project. The Empanada project aims at the development of techniques to predict quality attributes of software architecture and design models. There were many open master projects in the scope of the Empanada project.

I made an appointment with Dr. Chaudron. We talked about possible directions for a master project. We decided to search for techniques to assess the correspondence between a UML design and its implementation.

I am grateful to my supervisor Dr. Michel Chaudron for providing me a very interesting master project and his useful feedback. Special thanks to ir. Christian Lange and Martijn Wijns for their useful assistance during the project. Furthermore, I would like to thank Dr. Igor Radovanovic and Dr. Erik Luit for taking place in my examination board.

I would like to thank the following companies – in random order – for providing case studies to validate my techniques:

- Mark Coopmans from Philips Optical Systems in Best;
- Lou Somers, Eric Dortmans and Rob Kersemakers from Océ R&D in Venlo;
- Els Heemeijer and Edwin Roetman from Cap Gemini in Nieuwegein;
- Goce Naumoski and David van der Vliet from ASML in Veldhoven;
- Evgeny Eskenazi and Peter van der Meer from Philips TASS in Eindhoven;
- Michel Senden and Lambert Mühlenberg from Logica CMG in Maastricht.

The master project was sometimes difficult to combine with the obligations I have for my company, Questo Software. I would like to thank the clients of Questo Software for their understanding.

Dennis van Opzeeland

Department of Mathematics and Computer Science  
Technische Universiteit Eindhoven



# Contents

<b>Preface</b>	<b>iii</b>
<b>Contents</b>	<b>vii</b>
<b>List of figures</b>	<b>x</b>
<b>List of tables</b>	<b>xi</b>
<b>Acronyms</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Research question . . . . .	2
1.3 Outline . . . . .	2
<b>2 State of the art</b>	<b>5</b>
2.1 Existing work . . . . .	5
2.1.1 Relation with reverse engineering . . . . .	5
2.1.2 Methods for assessing correspondence . . . . .	6
2.1.3 Methods for highlighting differences . . . . .	7
2.1.4 Methods for preserving correspondence . . . . .	7
2.1.5 Relation Partition Algebra . . . . .	8
2.2 Existing tools . . . . .	9
2.2.1 CASE tools . . . . .	9
2.2.2 Code tools . . . . .	10
2.2.3 Metric tools . . . . .	11
2.2.4 Correspondence checking . . . . .	11
2.2.5 Visualization tools . . . . .	12
2.3 Conclusions . . . . .	13
<b>3 Correspondence</b>	<b>15</b>
3.1 Definition . . . . .	15
3.2 Metamodel . . . . .	16
3.3 Data used for correspondence assessment . . . . .	17
3.4 Similarities based on structural aspects . . . . .	18
3.4.1 Similarity measure primitives . . . . .	18
3.4.2 Similarity of common model elements . . . . .	20

3.4.3	Similarity of software systems . . . . .	23
3.5	Assessment strategy . . . . .	23
<b>4</b>	<b>Matching strategies</b>	<b>25</b>
4.1	Input and output . . . . .	25
4.2	High similarity matching . . . . .	25
4.3	Bipartite matching . . . . .	26
4.3.1	Weighted bipartite matching . . . . .	26
4.3.2	Weighted bipartite matching with thresholds . . . . .	26
4.4	Iterative improvement matching . . . . .	27
4.4.1	Partial matchings . . . . .	27
4.4.2	Improvement strategies . . . . .	28
4.5	Further improvement of the matching . . . . .	29
<b>5</b>	<b>Correspondence assessment using metrics</b>	<b>33</b>
5.1	Correlating metrics . . . . .	33
5.1.1	Correlation analysis . . . . .	34
5.1.2	Regression analysis . . . . .	35
5.1.3	The composition of a metric profile . . . . .	35
5.2	Using metric profiles for similarity measurement . . . . .	35
5.3	Using metric profiles for outlier analysis . . . . .	37
5.3.1	Significance . . . . .	37
5.3.2	The 95% confidence interval . . . . .	37
5.3.3	Single scatter plot analysis . . . . .	37
5.3.4	Multiple scatter plot analysis . . . . .	38
5.4	Conclusion . . . . .	39
<b>6</b>	<b>Tooling</b>	<b>41</b>
6.1	Key requirements . . . . .	41
6.1.1	Functional requirements . . . . .	41
6.1.2	Constraint requirements . . . . .	42
6.2	Tool architecture . . . . .	42
6.2.1	Global overview . . . . .	42
6.2.2	The database model . . . . .	44
6.2.3	Common tools . . . . .	46
6.2.4	Common dialogs . . . . .	47
6.2.5	Data extraction tools . . . . .	47
6.2.6	Correspondence matching . . . . .	48
6.2.7	Difference analysis . . . . .	52
6.3	Implementation notes . . . . .	54
6.3.1	Common queries . . . . .	54
<b>7</b>	<b>Case studies</b>	<b>55</b>
7.1	Case characteristics . . . . .	55
7.2	Matching validation . . . . .	55
7.3	Case assessment approach . . . . .	57
7.3.1	Preparation . . . . .	57

<i>CONTENTS</i>	vii
7.3.2 UML Model extraction . . . . .	58
7.3.3 Source code model extraction . . . . .	59
7.4 Matching strategies . . . . .	59
7.5 Scatter plot analysis . . . . .	60
7.6 Classification of deviations . . . . .	61
7.7 Concluding remarks . . . . .	63
<b>8 Evaluation and conclusions</b>	<b>65</b>
8.1 Project overview . . . . .	65
8.2 Conclusions . . . . .	66
8.3 Directions for further work . . . . .	66
8.3.1 Matching clusters . . . . .	67
8.3.2 Assessing the correspondence of behavioral aspects . . . . .	67
8.3.3 Automatic correspondence assessment . . . . .	67
8.3.4 Correspondence between design and documentation . . . . .	67
<b>Bibliography</b>	<b>69</b>
<b>A An example of an experiment script</b>	<b>73</b>



# List of Figures

2.1	The MetricView tool highlights differences between design and implementation	12
3.1	The correspondence measure metamodel	16
3.2	The relevant subset of the UML metamodel	17
3.3	The correspondence assessment data	17
4.1	Matching all similarity values of at least 0.8	25
4.2	The matching with maximal weight when using weighted bipartite matching	26
4.3	The result of weighted bipartite matching with threshold 0.8	27
4.4	Using the visualization proposed by van Ham, the matching can be visualized in an intuitive way	30
4.5	If the design and the implementation are organized in packages and directories, the matches are most likely to appear in related clusters	31
4.6	Different colors can be used to indicate the reliability of each of the matches	31
5.1	A scatter plot of two metrics	34
5.2	Comparing the value predicted from a design metric with a real implementation metric value results in an error $\epsilon$ .	36
5.3	A scatterplot with a regression line and a 95% confidence interval can be used to identify outliers	38
5.4	Different scatterplots in one view are useful for finding outliers	38
6.1	The correspondence assessment process	43
6.2	The high level design of the correspondence toolset	44
6.3	The database model	45
6.4	The MySQL connector and entity caches	46
6.5	The XMI processor design	47
6.6	The Metric processor design	48
6.7	The user interface design of the matcher application	49
6.8	The matching framework	50
6.9	The sequence of a typical assessment run	51
6.10	The algorithms implemented in the toolset	52
6.11	The scatterplotter tool showing different plots	52
6.12	The design of the scatter plotter tool	53
7.1	Scatter plots showing measures for different aspects	61
7.2	The outlier selected seems to have changes only in the inheritance tree	61



# List of Tables

6.1	Priority levels of the requirements . . . . .	41
7.1	The characteristics of the industrial case studies . . . . .	55
7.2	The reliability of the matching strategies . . . . .	57
7.3	The categories of an experiment script . . . . .	58
7.4	The matching approaches and corresponding results . . . . .	59
7.5	The matching approaches and corresponding results . . . . .	60



# Acronyms

## General acronyms

<b>CASE</b>	Computer Aided Software Engineering
<b>DBMS</b>	Database Management System
<b>MDA</b>	Model Driven Architecture
<b>PIM</b>	Platform Independent Model
<b>PSM</b>	Platform Specific Model
<b>RPA</b>	Relation Partition Algebra
<b>SAAT</b>	Software Architecture Analysis Tool
<b>SQL</b>	Structured Query Language
<b>UML</b>	Unified Modeling Language
<b>XMI</b>	XML Metadata Interchange
<b>XML</b>	Extensible Markup Language
<b>XSLT</b>	Extensible Stylesheet Language Transformations

## Metrics

<b>CBO1</b>	Coupling between objects
<b>CLD</b>	Class to leaf depth
<b>DIT</b>	Depth of Inheritance tree
<b>NOC</b>	Number of children
<b>NumAnc</b>	Number of ancestors
<b>NumAttr</b>	Number of attributes
<b>NumOps</b>	Number of operations
<b>NumPubOps</b>	Number of operations with public visibility



# Chapter 1

## Introduction

### 1.1 Motivation

The Unified Modeling Language (UML) is the de facto standard modeling language in software development. Software projects develop UML models in the architecture and design phase to document and communicate design decisions, and for the purpose of analysis. The models can be analyzed using e.g. metrics [MLC04] to predict quality attributes of the system that is going to be implemented. UML models describe the system on a higher abstraction level than source code does. This enables the reader to understand the architecture without taking the burden of reading through all the details of the source code. Hence, UML models are not only used in the early phases of software development but also during maintenance when it is important to understand how the system works. For a better understanding of legacy systems that are not described by an UML model sometimes even the effort of reverse engineering an UML model is taken.

Model-based analysis results are only reliable predictors for the implementation if the source code corresponds to the model. For understanding a system correctly using a model, it is necessary that the model corresponds to the actual implementation.

Several factors can cause a lack of correspondence between UML model and source code. We present the most prominent causes for non-correspondence in the following:

- **Implementation mismatch.** In the implementation phase the programmers write source code that is not according to the UML model. This can be by purpose (“I know it better” or implementation convenience) or by mistake.
- **Evolutionary mismatch.** Software needs to be changed because of changing requirements, bugs that must be removed, or other necessary improvements. If only the design or only the implementation is changed, the correspondence between model and source code will be lost. In most cases only the source code is changed.

These risks are common in practice and hence, software engineering has to deal with degradation in correspondence. When models are used for analysis or understanding, the engineer must be aware of the degree of correspondence to judge whether analysis results are reliable predictors and whether the model allows for correct understanding of the implemented system.

There are also other reasons for correspondence assessment. Certain UML metrics can be used as predictors for the implementation of a system. It is for instance possible to predict the

complexity of system parts based on the design models. These predictions can be considered valid only if the design resembles the implementation.

Another application for correspondence assessment is the detection of plagiarism in practical assignments for students. Students that commit fraud typically tend to change names of the UML model elements. The structure of the design is less likely to change however. The models of different students can be compared. In this case high correspondence is undesired.

## 1.2 Research question

The purpose of this thesis is to develop techniques to reconstruct and assess the correspondence between two versions of a model. The correspondence of an implementation with its design is a special case of this. The implementation can be seen as a model too.

To judge on the correspondence of two models, we need to zoom in on the parts of these models. These parts will be called *model elements*. We need to compare each model element from one model with its correspondent in the other model. Therefore we need to find out which model elements are related. For large cases it will be too much work to define this relation manually. Therefore, automated techniques for matching are an important topic in this thesis.

Once the matching is established, the actual comparison can be carried out. There are many ways to perform this analysis. There will also be many kinds of differences. Some of them will be more crucial than others. Especially the differences that pose high risks for understandability will be the differences of interest. We will present a classification of all sorts of differences we found in the various case studies we performed. Given the kind and the cause of a difference we can estimate how serious this difference should be taken.

We presented our work at the 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering [vOLC05].

## 1.3 Outline

This document describes the development of techniques for assessing correspondence between a software design and its implementation. It is outlined as follows.

- Chapter 2 gives a brief overview of the current state of the art with respect to correspondence checking and related techniques. Existing tools are also discussed in this chapter.
- A formal definition of correspondence is given in chapter 3. Furthermore, the metamodel of the data is discussed here.
- An important part of correspondence assessment is formed by mapping design pieces to implementation pieces. Several matching strategies exist for this purpose. Chapter 4 is devoted to this topic.
- The similarity measures discussed in chapter 3 are all based on structural aspects of classifiers. We can also measure similarity by using software metrics. This is described in chapter 5

- The techniques described in chapters 3, 4 and 5 are implemented in some correspondence tooling. The requirements for these tools as well as the architecture and implementation notes are listed in chapter 6.
- The correspondence tools were used to assess the correspondence of several industrial cases. Chapter 7 discusses the approach and the results for the various case studies we performed.
- Chapter 8 contains some conclusions and directions for further work.



# Chapter 2

## State of the art

This chapter gives an overview of the techniques that are currently available in the field with respect to correspondence analysis. First the state of the art in research is discussed. Then we will discuss some useful tools. The chapter concludes with a short overview of the techniques and tools that are useful for us.

### 2.1 Existing work

#### 2.1.1 Relation with reverse engineering

Assessing correspondence between a design and an implementation implies that the UML design artifacts are being compared to source code artifacts. For such a comparison to work, the level of detail of the source code should match the level of the design. This means that the implementation details from the source code need to be filtered out. This results in a model of the source code that is at a similar level as a detailed design of the software.

Extracting architectures or detailed designs from source code is called reverse engineering. The possibility to reverse engineer source code properly is thus crucial for correspondence checking.

A lot of research has been performed on the reverse engineering of software systems. RIGI [SWM97] is a well known example of a toolset for reverse engineering. Other examples are the Dali Workbench [KC99], CPPX [DMH01], Columbus/CAN [FMB<sup>+</sup>01] and SNiFF+ [Gmb96]. These approaches have in common that they all try to reconstruct a higher level model based solely on the source code.

The problem with reverse engineering is that it is generally impossible to generate a complete and accurate model completely automatically. Often the help of a user is required for reverse engineering or details are simply missed.

Many current UML CASE tools also have reverse engineering features. Especially Java is supported heavily by CASE tools. Java resembles C++ but it is easier to parse. C++ has a very extended macro mechanism and many different language constructs. Parsing C++ code correctly is therefore quite hard in general. CASE tools that can reverse engineer C++ source code generally only consider class declarations.

## 2.1.2 Methods for assessing correspondence

### Class matching

One method for correspondence checking is proposed by Antoniol et al. [ACPT01]. It concerns matching classes in the design with classes in the implementation. A class has several properties that generally distinguish it from other classes:

- The name of the class
- The name and signature of the methods
- The name and datatypes of the fields
- The visibility of the class members
- The relations to other classes – inheritance, dependencies, etc –

Similarity of two objects can be expressed in terms of their common properties. If the name of a class in the design matches the name of a class in the implementation, this is said to be a common property. Antoniol et al. use it just for classes. The same approach could also be used for interfaces, structures and even for global functions.

### Module coupling

At a somewhat higher level, we can also check correspondence based on module coupling [TCL02]. Both in design and implementation we count for each pair  $(X, Y)$  of modules how often  $X$  references  $Y$ . If the module coupling counts of  $X$  and  $Y$  differ in the design and the implementation, the conformance rules are violated.

In [TCL02] it is simply counted how many relations with other modules there are with a given module  $X$ . It is based on the coupling of objects metric proposed in [CK94]. The coupling of objects method counts the number of relations from an object to others.

The module coupling method simply counts the number of relations between two modules. The method ignores the fact that there are different kinds of relations. An improvement would be to take different kinds of relations into account. Instead of counting all relations between two modules, only relations of a specific type are counted. Such counts are done for all kinds of relations.

This method only considers the relations between modules or subsystems. We are typically interested in the differences at the level of classifiers.

### Using logic for correspondence

Both design and implementation evolve. These artifacts influence each other continuously. This is called co-evolution. In [Wuy01], a first step is taken towards a development environment where all artifacts are related to each other.

The focus of co-evolution is on changes of artifacts. The most important part of the co-evolution development environment is thus a mechanism that synchronizes changes in different artifacts such as, for instance, a detailed design and an implementation. A logic meta-programming language was integrated in the object oriented development environment for this synchronization.

Synchronizing artifacts is related to finding differences. When synchronizing, one tries to eliminate differences. However, if differences are found between design and implementation, synchronization is required.

### 2.1.3 Methods for highlighting differences

#### Reflexion models

Software reflexion models [MNS95, MNS01] are a way to visualize differences between design and implementation. The method focusses specifically on differences in relations between classifiers. Differences between the matched classifiers themselves are not considered.

A high level model consisting of components and component dependencies must be provided as well as the source code. This method assumes that the mapping between design and implementation entities already exists. This mapping thus has to be provided by another tool or manual definition.

Based on this input, a software reflexion model is generated. First the relations between classifiers in the source code are determined. The relations found in the sources are matched against the relations in the design. This results in three classes of relationships between components. If the relation occurs in both the design and the implementation, it is called a *convergence*. If the relation occurs in the design but is absent in the implementation it is an *absence*. If the relation occurs in the implementation but not in the design it is called a *divergence*.

Then the results are visualized by depicting all components from the design. The absences, divergences and convergences are shown using different types of arrows.

#### UML diff

The UML diff tool [Gir02] looks for differences between UML models. This is done by comparing two UML models in full detail. The correspondence of the model is expressed in terms of the number of changes one has to perform on the one model in order to transform it into the other UML model.

The typical application of this method is to use it in a configuration management system to trace changes between different versions of a UML model. The idea is the same as a textual diff algorithm for text files. Here two versions of a text file are compared and all differences are highlighted.

The deviations between design and implementation can be large. In such a case the diff algorithm results in a very large and detailed list of differences between design and implementation. This is not very valuable for a developer. Only those differences that compromise system understanding are relevant.

### 2.1.4 Methods for preserving correspondence

#### Code generation

Many CASE tools currently available support code generation for popular programming languages. Java and C++ code generators seem a standard tool in CASE tools. A code generator generates an initial framework for the source code from the UML model.

The programmer starts implementing the software from a correspondent framework in this way. This prevents errors and helps to keep correspondence between design and implementation. Once the programmer starts the implementation, new features are easily added to the framework. Code generation is thus not a full proof method to avoid correspondence issues.

Some CASE tools have a model synchronization option. These tools can not only generate code but also reverse engineer it to update the model. These tools are rather limited. They choose either the UML model or the synchronization as up to date. Then they modify the other artifact such that it corresponds. This synchronization is limited to structure. Behavior diagrams are not updated.

### Model Driven Architecture

Currently a lot of research is done on the topic of Model Driven Architecture [SOMG04]. In the MDA approach a platform independent model (PIM) is used to develop an application. This model contains no platform specific information. Along with the PIM there are one or more platform specific models (PSM). A PSM is specific to a platform and programming language. The PSM is mostly generated automatically from the PIM.

If new technologies emerge, no new model has to be created. After all, the PIM is platform independent. If tools exist that can generate a PSM for this new technology, the existing PIM can be mapped to this model automatically.

The mapping from PIM to a PSM can be seen as code generation. But it goes further than the code generation that is currently in many CASE tools. The CASE tools usually only generate a framework. The implementation of the methods is done completely manually. The MDA tools also consider behavior. So the implementation of the methods is also partially automated.

The implementation of a MDA design corresponds to the design completely just after generation. The idea of the MDA approach is not to change the implementation manually. Changing the implementation means that a PSM is changed. The PSM should be generated from the PIM completely however. If programmers don't change the PSM, the implementation will correspond the design.

#### 2.1.5 Relation Partition Algebra

Relation Partition Algebra (RPA) is suitable for various inconsistency checks [MCB05]. Correspondence can be expressed in RPA. If both source code and design are transformed into RPA models, certain rules can be checked. For the checking, Philips has developed a toolset that can be used to calculate RPA relations.

In [MCB05] a few examples of RPA relations are given that deal with the correspondence between design and implementation. Rules were defined that have to hold for conformance. An implementation can thus be found conform to its design or not. But it is much more interesting to see where inconsistencies can be found instead of just the fact that they are there.

Instead of using relational operators like subsets and set equality etc. it is also possible to use union, intersection and difference. Using these kinds of operators, one can obtain the actual elements that cause violations of the rules. For instance a rule that is used to check for invalid includes in C++ code (taken from [MCB05]):

$$INCLUDES \subseteq (DEPENDENCY \cup AGGREGATION \cup (I \uparrow INHERITANCE^*)) \downarrow IMPLEMENTS \quad (2.1)$$

Here, the *DEPENDENCY*, *AGGREGATION* and *INHERITANCE* relations are sets of pairs of classes. Each pair in one of these sets represents a dependency, aggregation or inheritance relation between two classes in a detailed design. *INCLUDES* expresses which implementation files include which header files. *IMPLEMENTS* relates the designed classes with files where – parts of – the implementation can be found. *I* is an identity relation which relates each class with each self.

An implementation file can only include a header if the relation that results from this inclusion is also designed. The right part of the expression in equation 2.1 is the set of all includes relations that are allowed to exist if the implementation corresponds to the design.

$$INCLUDES \setminus (DEPENDENCY \cup AGGREGATION \cup (I \uparrow INHERITANCE^*)) \downarrow IMPLEMENTS \quad (2.2)$$

The expression in equation 2.2 results in a set of all includes relations that are not allowed according to the design. Each element in the set that results from 2.2 represents a difference between design and implementation.

It is possible to express other deviations in terms of RPA relations. The RPA provides a mathematical framework for correspondence assessment.

## 2.2 Existing tools

This section discusses some of the existing tools that are in some way related to correspondence checking.

### 2.2.1 CASE tools

Computer Aided Software Engineering (CASE) tools are used to support software engineering. We focus on CASE tools that are used to draw UML models. CASE tools more and more support analysis of the models drawn. This section lists two tools that are handy for the project.

There is a standardized file format for UML models. This is the XMI format [OMG02]. Many CASE tools are able to export their UML models to an XMI file. If we use XMI as our import format, we are able to process data from many different tools.

In reality the standard is not as common as it should be. It seems that every CASE tool developer has his own interpretation of the XMI standard. This results in many different flavors of XMI files.

### Rational Rose and XDE

Rational Rose is a Case tool that is based on the UML modeling technique. It provides all views described in the UML standard [RJB99]. For Rational Rose, a plug-in is available that translates the model into the XMI file format. In the SAAT project [Mus02], the Rational Rose XMI plug-in is used to convert the model into an XMI model.

## Enterprise Architect

Enterprise Architect is another modeling tool. The most recent version (5.01) supports the UML 2.0 standard. The tool is interesting since it has the possibility to reverse engineer source code in various languages. It is able to deal with code written in C++, C#, Java, VB.NET, Visual Basic, Delphi and PHP. Furthermore it is able to write UML models in various flavors of XMI formats. The last interesting property of the tool is that it can also read various XMI flavors. This makes the tool useful for converting an unsupported XMI dialect into a supported one.

### 2.2.2 Code tools

Code tools can be used to extract facts from source code. Most of the tools discussed in this section focus on C++ code. The reason for the focus to be on C++ code is that the case studies described in chapter 7 are written in C++. In section 2.2.1 Enterprise Architect was discussed. This tool has built-in code extraction support for various languages. This implies that we are not limited to C++

## Rigi

RIGI [SWM97] is a complete reverse engineering toolset. It contains a parser `rigiparse` that parses C code. After this parsing the `rigiedit` tool can be used to visualize the reconstructed architecture.

The main disadvantage is that RIGI can only parse C code. C++ code is not supported however. This makes RIGI unsuitable for our intentions.

## Columbus/CAN

Columbus [FMB<sup>+</sup>01] is another fact extractor. For academic purposes, it is available for free. After extracting the source code, the fact base can be stored in, for instance, GXL, RIGI or XMI format. The XMI format version is 1.0 or 1.1.

Apart from structure extraction, Columbus/CAN can also calculate metrics. Columbus/CAN extracts 67 different metrics at class level. These metrics measure various aspects of the source code. A description of the metrics can be found in the user manual [Fro03].

## SNiFF+

SNiFF+ is a commercial fact extractor [Gmb96]. Unfortunately, an academic program is not available. SNiFF+ stores its extracted facts in a binary format. These files are impossible to read since there is no documentation available on the file format. An API is provided however. Using this API, it is possible to extract everything that can be visualized within SNiFF+. So that includes call graphs, inheritance trees and so on.

## Code parsers

Instead of using a fact extractor of the shelf, it is also possible to develop a custom made fact extractor which extracts exactly those facts that are interesting. Writing a parser – especially for C++ – is a tedious job however. There are parser generator programs such as ANTLR

[PQ95]. Unfortunately there doesn't seem to be a sufficiently complete grammar for C++ code.

An alternative to generating a parser is to use an existing parser. The C++ Front End [EDG04] of the Edison Design Group is such a parser. Given C++ code, it produces an abstract syntax tree in internal memory. The parser comes as a library that can be linked into a project.

The main disadvantage of code parsers are that the syntax tree should be converted to a format that is suitable for comparison to a design. A lot of work will be involved in the definition of such a conversion.

### 2.2.3 Metric tools

#### SDMetrics

SDMetrics [Wüs04] is a tool that can calculate 43 different design metrics on UML designs. The input of the tool is an XMI file. The metrics can be exported to a comma separated value file. The built-in metrics are typically design oriented and most metrics focus on the structural aspects of the design. It is however possible to define custom metrics.

#### SAAT

The Software Architecture Analysis Tool [Mus02] is another tool that can calculate metrics given an XMI model. It was developed at the Eindhoven University of Technology. Also this tool allows the definition of custom metrics. SAAT uses a custom ASCII output format.

#### C and C++ Code Counter

Another tool for software metrics is the C and C++ code counter. This tool focuses on code which means that typical source code metrics, such as Lines of Code, are available. The tool does not support customization of metrics. The output format of the metrics calculated is XML.

The name suggests that the tool can calculate metrics for C and C++ code, which is correct. The tool can also calculate metrics for Java code however. When analyzing sources written in Java, this tool is useful to obtain implementation specific metrics.

### 2.2.4 Correspondence checking

#### Relation Partition Algebra

Philips has created a set of simple tools to calculate with relations in RPA. This toolset is part of the Philips Architecture Analysis Toolset. The tools in this set have not been updated lately. This is no problem for RPA since all interesting rules have been implemented.

For each of the RPA rules, a tool exists in the toolset that can apply the rule to a given RPA. The tools accept input on standard input and produce output to standard output. Therefore the tools can be "piped" to express more complex relations.

The RPA toolset is implemented in Java. Instead of using the command line tools, it is also possible to use the Java code directly. A Java program can directly use the classes from the Philips RPA package. This is a little more efficient since intermediate results of complex calculations are not passed through the command line.

The RPA toolset implementation is a prototype however. This has consequences for the efficiency of the tool. Calculating with large relations causes high memory usage. The tool was never optimized for efficiency making it unsuitable for large case studies.

## 2.2.5 Visualization tools

### MetricView

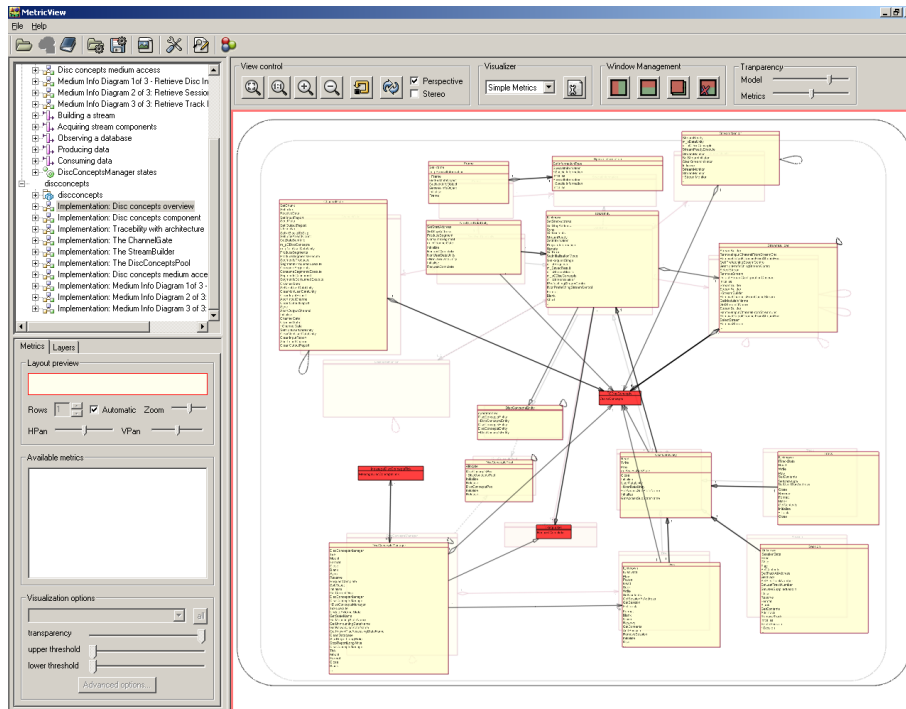


Figure 2.1: The MetricView tool highlights differences between design and implementation

MetricView [Ter05] is a tool that visualizes metric values overlayed on the original layout of a software design in UML. It was developed at the Eindhoven University of Technology. Originally the tool was intended to visualize software metrics in an intuitive way. An impression of the user interface is given in figure 2.1.

The user interface of MetricView is very much like a case tool. The main window shows a UML diagram. It is possible to split this window such that (different) diagrams can be shown simultaneously. On the left pane, there is a navigation tree that contains all elements in the model. Under the tree, the metric dashboard is located. This pane contains various settings to tweak the visualization of metrics onto the model.

Even though the primary goal of MetricView was to visualize metrics, new features are being added. Experiments have been done with respect to refactoring and model evolution. The result of an experiment about difference visualization is shown in figure 2.1. In this picture MetricView shows an overlay diagram. A graph of the implementation is depicted on top of the design. Correspondent classes are drawn on top of each other. This visualization allows us to find differences between design and implementation.

### Call matrices

Another interesting visualization approach is based on the idea of Multilevel Call Matrices [vH03]. In this approach, a matrix is constructed in which each cell  $M[i, j]$  indicates whether component  $i$  can call component  $j$ . The tool has been made hierarchical meaning that one can also zoom in on components. The value of the matrix cell indicates whether or not the relation is allowed. This is visualized by drawing the matrix on the screen and substituting colors for matrix values.

This approach can be used for other purposes as well. Any relation between two sets can be visualized using call matrices. A matching is also a relation between two sets of elements. This can thus also be visualized using the matrix visualization. The color of the matrix cells can also be varied. This can for instance be used to visualize the reliability of a relation.

## 2.3 Conclusions

The structure matching method described by Antoniol et al. is a useful approach for matching. We will implement this approach for finding the matching between design and implementation.

Many tools are available in the field for various tasks. Some of the case studies require us to use Rational Rose. This tool will only be used for generating an XMI file of the design model. Even though there are no models in the Enterprise Architect format, this tool is useful for reverse engineering Java, C#, Delphi and Visual Basic code. It reconstructs a UML model based on source code in any of these languages. Enterprise Architect is also able to read XMI flavors from many different tools.

C++ code is a more difficult language to parse. The extended macro mechanism and the large variety of language constructs cause many code extractors to miss things. Especially code extractors that are built-in in CASE tools ignore complex macro's. We chose Columbus/CAN for this since this tool performs quite well. It is able to generate a UML model of the source code which is exactly the representation we want.

For calculating metrics we can use different tools. Some tools such as SDMetrics and SAAT calculate metrics based on a UML model. These tools can be used for the design but also for the implementation, provided that it is reverse engineered into a UML model. Other metric tools are specifically for the implementation. These tools can also calculate a number of metrics that are specific to implementations. The lines of code metric is an example of such a metric.

Both the MetricView approach and the matrix visualization can visualize the matching between design and implementation. The MetricView visualization focuses on the deviations between design and implementation. The matrix view focuses on the matching itself. We are going to use the call matrix visualization to help the user in establishing the matching. Contrary to the overlay diagrams, the call matrix approach is easily updatable. Difference visualization will be outside the scope of this thesis. It will be future work.



## Chapter 3

# Correspondence

In this chapter we will define the notion of correspondence. The correspondence between design and implementation is expressed in terms of so called *similarity values*. In section 3.4 we will address a method for calculating these similarity values. In order to do this, we need a metamodel that defines what data is available for correspondence checking. The meta model is described in section 3.2.

### 3.1 Definition

An implementation is said to conform to its design if “everything that was designed is implemented as it was designed and nothing more”.

Correspondence of a software system is expressed in terms of the *model elements* appearing in the UML model and related parts of source code. That is, the implementation corresponds to the design if the implementation model elements correspond to the related design model elements. Design model elements can be anything that occurs in a UML model such as a class, an operation or a state in a state machine. An implementation model element is a piece of source code representing, for example, the implementation of a method or the declaration of an attribute.

We want correspondence between design and implementation for the following reasons:

- In order to implement new features in a system, one should first *understand* how the system works. It is convenient to use a design model for this. Reading thousands of lines of source code is not a very fruitful approach for this. The design should correspond to the implementation. Otherwise, understanding the design does not help in understanding the implementation.
- It is possible to *predict* certain quality aspects of an implementation given a design. Based on the design, certain software metrics are calculated. These metrics can indicate for instance which parts of the system will be complex. These predictions can be correct for the implementation only if the design corresponds to the implementation. Differences can influence the reliability of a prediction.

### 3.2 Metamodel

The definition of correspondence suggested that the level of correspondence is a function of the elements in the model. This means that we need a definition of what can be a model element. This could be different between design and implementation. Figure 3.1 shows our metamodel.

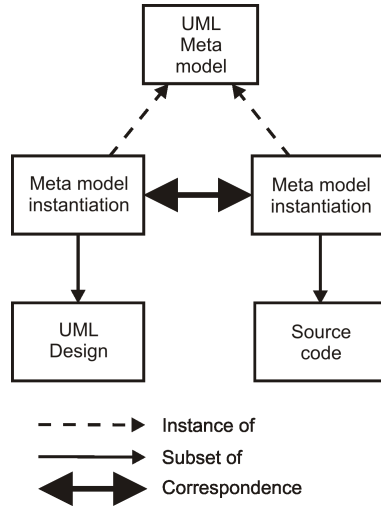


Figure 3.1: The correspondence measure metamodel

We base both our design and our implementation on the UML metamodel. That means that we have an instantiation of the UML metamodel for the design and an instantiation for the implementation. The data in these metamodels is being compared in order to assess correspondence.

We do not use all data from our design. Our UML design consists of a description of the structure and behavior. We leave behavior out of scope so we only use a subset of the data that is available in our UML model. The same holds for the implementation. All implementation details are ignored. Examples of implementation details are for instance selection and repetition statements or comments.

The data we can use for correspondence checking is thus defined by the UML metamodel. Figure 3.2 shows the subset of the UML meta model that we will use.

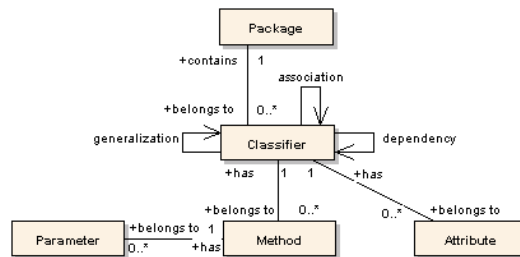


Figure 3.2: The relevant subset of the UML metamodel

The packages from the UML are very much like directories from the implementation source tree. If the implementation is organized in a hierarchical directory structure, then the

directories map to packages in the UML meta model. In some language such as C++ [Str91], there is a namespace construct. Namespaces can be used as an alternative packaging method.

Dependency and association relations between classes in the implementation are not modeled explicitly. It is possible however to determine the relations. A class  $A$  depends on another class  $B$  if  $A$  uses  $B$  in some way. One of the methods of  $A$  could for instance have a local variable with  $B$  as its type.

If one of the attributes of  $A$  has  $B$  as its type, this means that there is an association relation. It is in general not possible to determine the multiplicity of association roles. Take for instance the following class definition for  $A$ :

```
class A {
    private:
        B* _member;
};
```

Here, members can be a reference to an object of type  $B$  but it can also be a reference to an array of objects of type  $B$ . Hence multiplicities are generally not recoverable.

### 3.3 Data used for correspondence assessment

Figure 3.3 shows the data we use in the correspondence assessment. The blocks are data. An arrow between two blocks is a data transformation.

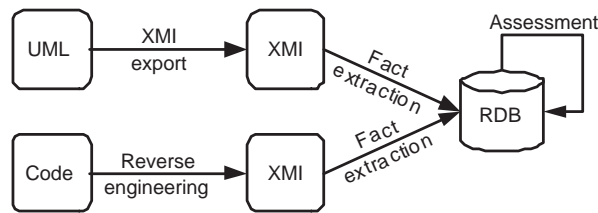


Figure 3.3: The correspondence assessment data

XMI is a standardized file format for storing UML models. Our techniques take UML models as input. Many UML CASE tools support export to XMI format. It is also possible to convert source code into XMI format.

Some reverse engineering tools support this. Alternatively, one can use a reverse engineering utility in a UML case tool. A UML model is created from the source code. This model can be exported to XMI.

For the correspondence assessment, we like to use a relational database. Such a database allows us to execute various queries easily. This means that we have to store the data from the XMI files in such a database. This is called fact extraction.

### 3.4 Similarities based on structural aspects

In literature there already exist some methods to compare models by just looking at structure. Our approach resembles the approach of Antoniol et al. [ACPT00]. The way we calculate the similarity between two classifiers is different however.

We compare UML elements using so called similarity measures. A similarity measure is a value that ranges between 0 and 1. If the similarity value is 1, it means that the elements being compared are completely similar. If the value is 0 then no commonalities are found at all.

For each type of model element a function can be defined to compare instances of this element type. There can thus be a similarity function for each model element specified in the UML meta model. We will only define similarity functions for model elements that are in the scope of our correspondence analysis.

Different model elements have aspects in common. All model elements have names for instance. Some model elements own a set of other model elements. A class owns a set of

attributes for instance. Before we define concrete similarity functions for model elements we will first define some general approaches for calculating similarity values. These will be referred to as similarity primitives.

After that, it will be possible to define concrete similarity functions easily.

### 3.4.1 Similarity measure primitives

This section discusses some general approaches for calculating similarity values.

#### Similarity based on names

The UML Meta model allows all elements to have a name. These names can be compared on equality. This approach is too strict. Small differences between names immediately result in inequality and thus no match. Small differences in names occur quite frequently however. Many people use coding standards that might cause changes in names.

The Hamilton distance [MP80] between strings is a better approach for matching names. The Hamilton distance counts the minimal number of changes (insertions, deletions or modifications) that has to be applied to convert one string into the other. It is denoted by  $d_H$ .

The Hamilton distance in itself is still not a good similarity measure. The reason for this is that it usually takes few changes to convert a small string into another small string. Therefore the Hamilton distance between two totally different short strings can be smaller than the Hamilton distance of two very similar long strings. Apart from that, the range of the Hamilton distance does not fit to the range of a similarity value.

To overcome these problems, the Hamilton distance between two strings is divided by the length of the longer of the two strings. This value is subtracted from 1 to get the similarity value into the right range. So given two names  $N_D$  and  $N_I$ , the similarity between these names is expressed as follows

$$sim_H(N_D, N_I) = \frac{|N_D| \uparrow |N_I| - d_H(N_D, N_I)}{|N_D| \uparrow |N_I|} \quad (3.1)$$

Consider for example the strings  $D = \text{cherry}$  and  $I = \text{chart}$ . Then  $|D| = 6$  and  $|I| = 5$ .  $d_H(D, I) = d_H(\text{cherry}, \text{chart}) = 3$ . Then

$$sim_H(D, I) = sim_H(\text{cherry}, \text{chart}) = \frac{3}{6 \uparrow 5} = \frac{3}{6} = 0.5$$

#### Similarity based on types

The attributes of a classifier as well as the parameters and return value of the classifiers methods can be typed. A data type can either be a basic data type such as `int` or `char` or (another) classifier.

If there exists a mapping between types from one software model to another – say from design to implementation – type information can be used as a similarity measure. Unlike names, types are being matched based on equality. The types of two attributes either match or they don't. That yields the following equation for types  $T_D$  and  $T_I$ .

$$sim_T(T_D, T_I) = \begin{cases} 1 & \text{if } T_D \text{ matches } T_I \\ 0 & \text{otherwise} \end{cases} \quad (3.2)$$

Note that this approach assumes that a mapping exists between design types and implementation types. For basic data types this is usually the case but when creating the mapping, as described in chapter 4, this information is not available as it is the result of the process.

Suppose for example that the design consists of class  $A$ . In the implementation we have  $B$  and  $C$ . We already have a partial matching in which  $A$  is related to  $C$ .

Then  $sim_T(A, C) = 1$  because these types match.  $sim_T(A, B) = 0$  because  $A$  and  $B$  are not related in the partial matching.

### Similarity based on flags

Some UML elements have some simple properties. A class can for instance be abstract or not. A method can be public, protected or private. These properties can be represented by a boolean or enumeration variable.

Let  $P(d, i)$  be a predicate that states whether  $d$  and  $i$  have the same value for a property  $P$ . If  $d$  and  $i$  would be methods, then  $P$  could check whether the visibility of  $d$  and  $i$  is the same. There can be several properties that must be checked. A method can for instance also be virtual or not. Let  $\mathcal{P}$  be the set of properties that need to be compared. Then the similarity of  $d$  and  $i$  is expressed by equation 3.3

$$sim_F(d, i) = \frac{\sum_{P \in \mathcal{P}} P(d, i)}{|\mathcal{P}|} \quad (3.3)$$

### Similarity based on sets

There are model elements that contain sets of other elements. A class has for instance a set of attributes or a package has a set of classes. It is possible to calculate the similarity between elements in the sets. But given two sets of elements  $S_D$  and  $S_I$ , what is their similarity?

It is assumed that there exists a (partial) one-to-one matching between elements from  $S_D$  and  $S_I$ . For sets of attributes or methods of a class, this usually holds. For classes in a package this might be a false assumption. Note that a one to one matching does not imply that  $|S_D| = |S_I|$ . As a result of the deviations elements in these sets might not be matchable.

For each pair  $(d, i) \in S_D \times S_I$ , it is possible to calculate  $sim(d, i)$ . The similarity of the sets should be some combination of the similarities between the elements. It is thus required to find out which elements from the sets belong together. This problem can be solved by using a weighted bipartite matching algorithm. The two sets of vertices are formed by the sets  $S_D$  and  $S_I$ . Between every  $d \in S_D$  and  $i \in S_I$ , there exists an edge which has  $sim(d, i)$  as its weight. The weighted bipartite matching algorithm can then find a matching  $M$  of maximal weight.

$$sim_S(S_D, S_I) = \frac{\sum_{(d,i) \in M} sim(d, i)}{|S_D| \uparrow |S_I|} \quad (3.4)$$

### Composite similarities

In this section various different primitives for similarity have been presented. For most model elements it is possible to calculate the similarity in different ways. It is best not to use just one

measure for similarity but to combine different measures. Take for instance the comparison of two attributes. Attributes have names and types. For both the name and the type, a similarity primitive is available. The most reliable similarity value takes both the similarity of the names and the similarity of the types into account.

Not all of the similarity measures are equally important. The name of an attribute says more about the similarity of the attribute compared to another than visibility of an attribute does. Therefore composite similarity between two elements  $E_D$  and  $E_I$  is defined as the weighted sum of different similarity values. To get the resulting value in the correct range, the weighted sum is divided by the sum of the weights.

$$sim_C(E_D, E_I) = \frac{\sum_m w_m \cdot sim_m(E_D, E_I)}{\sum_m w_m} \quad (3.5)$$

### 3.4.2 Similarity of common model elements

Using the primitives defined in section 3.4.1, it is now possible to express actual similarity functions of real model elements. For most model elements the similarity is a composition of different primitives each assessing a different aspect of the model element.

The weights of the compositions in this section are parameterized. This is because the optimal weights are case specific.

#### Similarity of classifier attributes

An attribute is mainly determined by its name, its type and visibility. The similarity of two attributes is thus a composition of the similarities between these aspects.

The name of an attribute usually is the most distinctive as the attributes all have different names. The weight parameter for the name similarity value  $w_n$  should thus be relatively large. As a result, this similarity value becomes more influential.

Data type information can only be used if it is available for both models and if there exists a mapping between data types from one model to another. For basic data types this is not a problem but for classifier data types this should be determined first.

$$sim(A_D, A_I) = w_n \cdot sim_H(\text{name}(A_D), \text{name}(A_I)) + w_t \cdot sim_T(\text{type}(A_D), \text{type}(A_I)) + w_v \cdot sim_F(\text{visibility}(A_D), \text{visibility}(A_I)) \quad (3.6)$$

Take for instance the following attributes:

```
public Attribute1: Integer;
public Attribute2: String;
```

$sim_H(\text{Attribute1}, \text{Attribute2}) = 0.9$ . The visibilities of the attributes are the same and their types are different. Hence the similarity between these two attributes equals  $w_n \cdot 0.9 + w_t \cdot 0 + w_v \cdot 1 = w_v + 0.9 \cdot w_n$ .

### Similarity of operations

Operations are determined by their name, their signature and their visibility. The signature consists of the *list* of parameters and the return type.

$$\begin{aligned} sim(O_D, O_I) = & w_n \cdot sim_H(\text{name}(O_D), \text{name}(O_I)) + \\ & w_t \cdot sim_T(\text{rettype}(O_D), \text{rettype}(O_I)) + \\ & w_v \cdot sim_F(\text{visibility}(O_D), \text{visibility}(O_I)) \end{aligned} \quad (3.7)$$

Take for instance the following operations:

```
public Operation(): Integer;
private Operation(): String;
```

$sim_H(\text{Operation}, \text{Operation}) = 1.0$ . The visibilities of the operations differ and their return types are the same. Hence the similarity between these two operations equals  $w_n \cdot 1.0 + w_t \cdot 1 + w_v \cdot 0 = w_t + w_n$ .

### Similarity of inheritance relations

An inheritance relation is defined by two class types: an ancestor and a child. Two inheritance relations are thus similar if they have the same parent and the same child types.

$$\begin{aligned} sim(IR_D, IR_I) = & w_{pt} \cdot sim_T(\text{parenttype}(IR_D), \text{parenttype}(IR_I)) + \\ & w_{ct} \cdot sim_T(\text{childtype}(IR_D), \text{childtype}(IR_I)) \end{aligned} \quad (3.8)$$

Suppose  $A$  and  $B$  are classes from the design.  $C$  and  $D$  are classes in the implementation.  $A$  and  $C$  correspond to each other. Suppose that  $A$  is a generalization of  $B$  and  $C$  is a generalization of  $D$ . Then the similarity between these two inheritance relations is 0.5. That is because the parent end points of the relation are related types. The child types do not match however.

### Similarity of dependency relations

Like inheritance relations, a dependency relation is defined by two class types: a supplier and a client. It is good practice in the design to assign a name for a dependency relation. If the model is extracted from source code however, the dependencies are not named at all. When comparing two design models, it is possible to take the name into account but since this thesis focusses on comparing design and implementation, the names are omitted in the similarity measure.

$$\begin{aligned} sim(DR_D, DR_I) = & w_s \cdot sim(\text{suppliertype}(DR_D), \text{suppliertype}(DR_I)) + \\ & w_c \cdot sim(\text{clienttype}(DR_D), \text{clienttype}(DR_I)) \end{aligned} \quad (3.9)$$

### Similarity of association relations

Association relations are more difficult than dependency and inheritance relations. Association relations can have two or more association ends. An association end consists of a multiplicity, a classifier type and a name. When extracting from source code names and multiplicities are usually not found correctly. Therefore only the classifier types of association ends are considered.

Comparing two association relations is thus comparing the set of classifier types. This is a straight instance of the set similarity primitive as discussed in section 3.4.1.

### Similarity of classifiers

By classifiers we mean model elements that appear in structural diagrams. Examples of these are classes and interfaces. These kinds of elements have the following commonalities:

- a name
- a set of operations
- a set of attributes (usually an empty set for interfaces)
- a set of relations to other classifiers. (Relations include associations, dependencies and inheritance).

Each of these can be measured for similarity. The name primitive from section 3.4.1 can be used to calculate the similarity value for the first one. The others are all instances of set similarity.

This gives us 4 different similarity values which are combined using the similarity composition primitive. The weights should be configurable for different cases. Sometimes naming works best and for other cases the operations might be more stable. The default setting is to assign equal weights to all four similarity values.

$$\begin{aligned} sim(C_D, C_I) = & w_n \cdot sim(name(C_D), name(C_I)) + \\ & w_o \cdot sim(operations(C_D), operations(C_I)) \\ & w_a \cdot sim(attributes(C_D), attributes(C_I)) \\ & w_r \cdot sim(relations(C_D), relations(C_I)) \end{aligned} \quad (3.10)$$

### Similarity of packages

Assessment of similarity of packages is only possible if this information is in some sense available in both design and implementation.

For the design this means that there is a model management view [RJB99] which organizes the design into a hierarchical structure.

For the implementation it is usually possible to use the location of code pieces in files and directories. Files and directories show close resemblance with packages from the design in the sense that packages are merely model elements containing classifiers and other packages.

Expressing the similarity of two packages is the straight result of the similarity of the classifiers. If a matching  $M$  is available between design classes and implementation classes, then it is possible to calculate the package similarity like with the set similarity primitive.

$$sim(P_D, P_I) = \frac{\sum_{(C_D, C_I) \in M} sim(C_D, C_I)}{|C_D| \uparrow |C_I|} \quad (3.11)$$

### 3.4.3 Similarity of software systems

In section 3.1 we presented correspondence to be a measure that expresses the similarity between a design and an implementation. Using the primitives for similarities from section 3.4.1 we were able to assign a value for the similarity of different model elements. In section 3.4.2 we presented approaches to calculate the similarity for some – relevant – parts of the UML meta model.

The similarity of an entire system can be calculated using the techniques described in the previous sections. There are two approaches to do this:

- The system can be seen as a set of classifiers. These classifiers then have their operations, attributes and relations. In that case the correspondence of the system is again an instance of the set primitive.
- It is possible to include package information. In that case, there exists some root package that contains the entire model in the design. For the implementation there is a root directory that holds the entire implementation. As discussed above, both of these can be regarded as a package. The correspondence then equals the similarity of the root packages.

## 3.5 Assessment strategy

The discussion in section 3.4 shows a way to recursively determine the similarity between a design and an implementation. This assessment can be done as detailed as required.

For some similarity calculations, such as similarity between relations, similarity between classifier types is considered. The primitive for types was defined in a way that suggests the existence of a mapping between classifiers from the design and the implementation. This mapping thus should be determined prior to calculating similarities based on types.

There is another reason for such a mapping to exist. Measuring similarity is a somewhat recursive procedure. For example, to assess the similarity of two classifiers, the similarity of the set of operations is considered. The primitive for set similarity suggests a bipartite matching which implies that the similarity between *all* pairs of an operation from a design class and an operation from the implementation class should be calculated.

If the similarity of the system is regarded as calculating the similarity of a set of classifiers, this method becomes by far too complex. It is thus better to find a mapping between design classifiers and implementation classifiers. After that only similarities of design classifiers and *related* implementation classifiers are calculated as the others are useless anyway.

The strategy for assessing correspondence between models is thus a two phase approach. In the first step, the mapping between classifiers is recovered. After that it is possible to calculate the similarity of a system in full detail.

## Chapter 4

# Matching strategies

In the previous chapter we introduced methods to measure the similarity of a design and an implementation. These similarity values can be used to determine a matching between design and implementation. In this chapter we present some approaches to find such a matching.

### 4.1 Input and output

The matching algorithms take a matrix  $M$  of similarity values as input.  $M$  stores a similarity value for each pair of a design class and an implementation class. These matrices can be created by using one or more similarity algorithms.

The output of a matching algorithm is a set of matches. Each match is a pair of a design class and an implementation class.

### 4.2 High similarity matching

High similarity matching finds a match between any two classes  $d$  and  $i$  for which  $sim(d, i)$  is above some threshold. This algorithm compares all values in the matrix with the threshold. This method is not limited to one-to-one matchings. It can also generate many-to-many matchings.

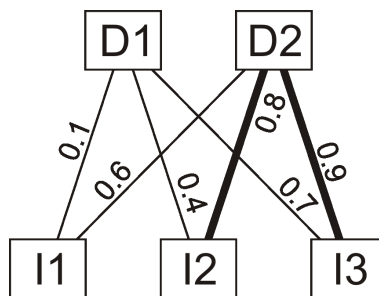


Figure 4.1: Matching all similarity values of at least 0.8

Figure 4.1 shows an example of high similarity matching. The threshold is set to 0.8 in this example. As a result a pair of classes is matched if and only if their similarity exceeds 0.8. Note that in this example the resulting matching is not a one-to-one matching.

### 4.3 Bipartite matching

When comparing a detailed design with an implementation, the matching often is one-to-one. That is, a class from the design matches with one class in the implementation and vice versa. Note however, that not all classes from design and implementation need to be matched however! If classes were introduced in the implementation, they should not match with any class from the design.

If it is known that there exists a one-to-one matching between classifiers from the design and classifiers from the implementation, then it is better to use a bipartite matching algorithm. This enforces a one to one matching between design and implementation whereas high similarity matching does not.

#### 4.3.1 Weighted bipartite matching

A weighted bipartite matching algorithm can be used to find a matching between design and implementation of maximal weight. The design classes and the implementation classes form two classes of vertices. There is an edge between each pair of a design class and an implementation class. The edge is labeled with the similarity value of the vertices.

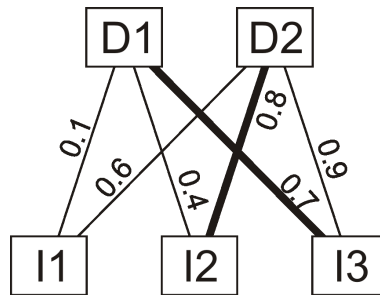


Figure 4.2: The matching with maximal weight when using weighted bipartite matching

Figure 4.2 shows a weighted bipartite matching in the graph. The picture immediately shows the shortcoming of this method. D2 and I3 have a very high similarity. They are not matched however since the matching depicted in the figure has a better overall result. The difference in the number of classes in the design and the number of classes in the implementation indicates that there are differences between design and implementation. Class I1 is a class that was introduced in the implementation in this example.

#### 4.3.2 Weighted bipartite matching with thresholds

This weighted bipartite matching approach has an important drawback. It searches for a matching that has maximal weight. It might thus relate classes that are completely unrelated. Suppose we have a design for which one class  $C$  has not been implemented. In the implementation one extra class  $X$  is inserted. The weighted bipartite matching algorithm will relate  $C$  and  $X$  since even a very small similarity value improves the matching.

The problem is easy to solve with a little pre-processing. We start the bipartite matching with a complete weighted bipartite graph. Before matching, we remove all edges from the graph for which the weight is below the threshold value. In this way only reliable edges are being matched.

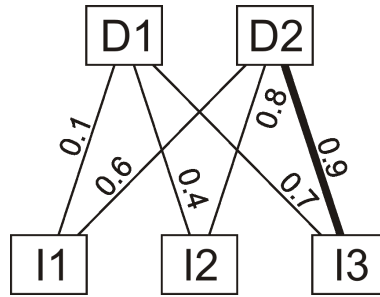


Figure 4.3: The result of weighted bipartite matching with threshold 0.8

In figure 4.3 we set a threshold of 0.8. That means that all edges with a weight lower than 0.8 are ignored. Now the high similarity between D2 and I3 does result in a match.

## 4.4 Iterative improvement matching

In practice there are parts of the implementation that resemble the design very much. Other parts seem to be much more different. It is hard to choose parameters for the similarity measurement in such a way that the resulting matching is optimal.

For some parts of the system matching based on names works very well whereas in another part the relations between classes are more reliable. The weighted bipartite matching approach discussed in section 4.3.1 does not allow us to select different parameters for different parts of the system.

The problem can not be solved by processing the system part by part. It is not certain that the decomposition of the design matches the decomposition of the implementation. This can deviate as well!

### 4.4.1 Partial matchings

When using thresholds with the weighted bipartite matching algorithm, we may obtain a partial matching. If there is a part in the system where matching on the names of the classes works very well, we obtain a partial matching of classes for which the names are very similar.

The parts that were not matched are too different with respect to their names. We can of course apply another strategy on the unmatched classes. That gives another partial matching. In this way we can perform as many iterations as desired.

Each iteration takes only unmatched model elements as input. Classifiers that are matched are no longer considered. The first iteration we select a strong algorithm that can create a reliable initial matching. Other less reliable methods are applied on the unmatched parts in later iterations.

It is also possible to use different similarity measures in a single iteration. The composite similarity measure primitive from section 3.4.1 allows us to combine different measures.

## 4.4.2 Improvement strategies

### Iterative name matching

The similarity of names is calculated using the Hamilton distance between the strings. Sometimes the names are identical in design and implementation. For other classifiers, the names might have been changed a little. In order to match these elements as well the threshold of the matching algorithm should be lowered. There is a chance that unrelated items get matched in this way. For bipartite matching, it might even mean that false negatives occur. Classes that should have been matched are not matched because the overall weight of the matching gets better as a result of that.

We can deal with this problem by running two iterations. In the first iteration, the threshold is set high. Only very similar classes are matched in this run. The second iteration is run on those elements that have not been matched in the first run. Now the threshold is set lower.

The benefit of this is twofold. On the one hand, the classes that are really similar are certainly matched. These matches are already found in the first iteration. On the other hand, the second iteration runs on the remainder of the classes. The search space is thus reduced. In general the correct matching is more likely to be found if the search space is smaller.

### Favor local matching

We can compose different similarity values into one. This was described in section 3.4.1. Suppose that the decomposition of the system resembles the directory structure in the implementation very much. This implies that classes in the same design package are likely to be found in the same source directory. The matching approach is able to deal with this.

Suppose there is a partial matching already. This matching is at class level. The matching can be lifted to the level of packages and source directories. In this way, we obtain a relation  $R_P$  between design packages and source directories. Now the similarity of two classes can be determined based on their packages as in equation 4.1.

$$sim_p(d, i) = \begin{cases} 1 & ((d, i) \uparrow) \in R_P \\ 0 & ((d, i) \uparrow) \notin R_P \end{cases} \quad (4.1)$$

Here,  $(d, i)$  is used to state that classes  $d$  and  $i$  are related. The relation between  $d$  and  $i$  is lifted to the level of the packages in which  $d$  and  $i$  are defined. Lifting of a relation is done using the RPA [MCB05] lift operator  $(\uparrow)$ . If  $d$  and  $i$  are defined in related packages, then  $(d, i) \uparrow$  should be in the set of related packages  $R_P$ .

Obviously, this approach should not be used alone. It relates all classes in a package to all classes in a related source directory! It can be used in combination with another similarity measurement however.

In chapter 5 we present the metric profile approach for measuring similarity between classes. On large data sets the metric profile approach is not very effective since the profiles are generally not distinctive enough. We can combine the similarity values of the metric profile approach with the similarity values of the package matching approach as in equation 4.2.

$$sim_c(d, i) = w_p \cdot sim_p(d, i) + w_m \cdot sim_m(d, i) \quad (4.2)$$

In this equation  $w_p$  is the weight for the package similarity value and  $w_m$  is the weight for the metric profile similarity value. With these weights, it is possible to calculate the similarity for any pair of classes. This information is provided to the weighted bipartite matching algorithm. After matching, the value of each match is checked against threshold  $T$  to filter garbage matches.

How to choose the parameters  $w_m$ ,  $p_m$  and  $T$ ? The result of the package similarity measure can be 0 or 1. The metric similarity value can be anything between 0 and 1. The matching criteria are as follows:

- If two classes are in related packages, the classes should only match if the metric similarity value is at least  $T_m^+$ .
- If two classes are in unrelated packages, the classes should only match if the metric similarity value is at least  $T_m^-$ .

These matching criteria can be transformed into set of equations 4.3:

$$\begin{cases} w_p \cdot 1 + w_m \cdot T_m^+ = T \\ w_p \cdot 0 + w_m \cdot T_m^- = T \\ w_p + w_m = 1 \end{cases} \quad (4.3)$$

This set of equations solves to:

$$\begin{cases} w_m = \frac{1}{T_m^- + 1 - T_m^+} \\ w_p = 1 - w_m \\ T = w_m \cdot T_m^- \end{cases} \quad (4.4)$$

The parameter assignment of equation 4.4 has some pleasant properties:

- If it doesn't matter whether or not two classes are in the same package then  $T_m^+ = T_m^-$ . The result is that  $w_m = 1$ ,  $p_m = 0$  and  $T = T_m^-$ . In this case, the package similarity measures simply cancel out.
- It is also possible to enforce local matching instead of favoring it. By choosing  $T_m^-$  to be larger than 1, classes simply can't be matched without being in related packages.

## 4.5 Further improvement of the matching

The matching strategies we presented here can be used to find a good initial matching. The matching will seldom be complete. Especially when there are large deviations, it becomes very hard to relate design classes with implementation classes.

The automatic approaches we discussed all assume that classes are similar on some aspect. Matching on names assumes that class names of relatable classes are similar. If the similarities between design classes and their implementation are not high enough it will be impossible to make a match in any way.

There is no way to ensure that a matching found is accurate and complete. It must thus be possible to improve the matching manually. The effort for manual matching should be minimal. What can be matched automatically, should be matched automatically. If a user has to define a manual matching in a system of a thousand classes, it is not likely that the matching will be defined.

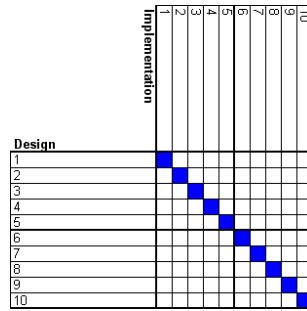


Figure 4.4: Using the visualization proposed by van Ham, the matching can be visualized in an intuitive way

Even if a large part of the system is matched, the user should get an intuitive view of this matching. Thereby the parts that are matched can be checked by the user. In section 2.2.5 we discussed the call matrices visualization [vH03]. This method can be used to visualize any relation between two sets of elements. It can thus be used to visualize the matching. Figure 4.4 gives an impression of the matrix visualization.

The user can easily change the matching. This can for instance be done by clicking a colored cell. This visualization provides a good overview of the matching. Even large matchings can be visualized in a relatively small window.

The approach becomes even more effective if the classes are sorted on the package or directory in which they are located. If the decomposition of the design matches the decomposition of the implementation, the matches between classes will very likely appear in related clusters. Figure 4.5 shows this. The matching suggests for instance that the cluster Pack3 is related to Dir2. There is no class in cluster Pack3 that is matched to a class in another cluster than Dir2. In the example there is one match between classes from Pack1 and Dir2. This match draws attention since there are no other matches between classes from Pack1 and Dir2.

Matches between classes in unrelated clusters will be visible immediately. These matches will be away from any matching cluster.

It is also possible to vary the color of the matrix cells as depicted in figure 4.6. This can for instance indicate the strength of the matching. If a design class looks completely the same in design and implementation, the match is more reliable. In that case, the cell could be colored with a very dark color. A light color would indicate a potential match which has to be validated manually.

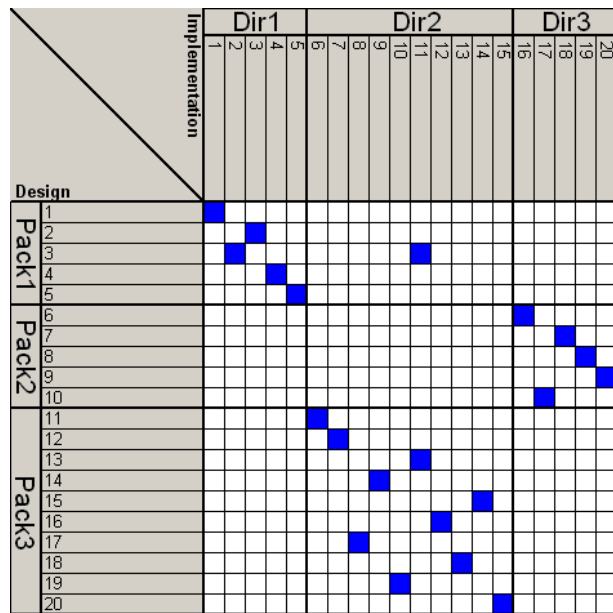


Figure 4.5: If the design and the implementation are organized in packages and directories, the matches are most likely to appear in related clusters

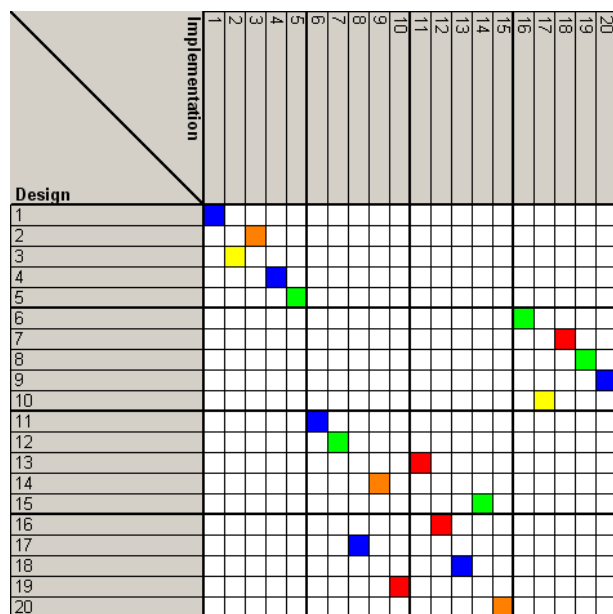


Figure 4.6: Different colors can be used to indicate the reliability of each of the matches



## Chapter 5

# Correspondence assessment using metrics

In chapter 3 an approach was described to calculate a similarity value for a design and an implementation. The method compared the structure of the design with the structure of the implementation and rated the correspondence by a similarity value. This similarity value was a composition of other similarity values. Each of these similarity values represented the similarity of a part of the system.

Instead of comparing the structure of the design and the implementation in full detail, it is also possible to compare software metrics. This chapter discusses the metric profile approach.

We will focus on classifiers. It is also possible to apply the same approach at the level of methods or at system level.

### 5.1 Correlating metrics

For each class we can calculate several metrics. In literature many metrics have been proposed both for design and implementation classes. There are three categories of metrics.

1. Metrics that are only available for design classes. An example is the number of diagrams a class occurs in. This metric makes no sense in an implementation.
2. Metrics that are only available for implementation classes. The lines of code metric is an example of this. In the design no code is written yet.
3. Metrics that are available in both design and implementation. An example is the number of methods of a class metric. In both design and implementation it is possible to simply count the number of methods.

The idea of the metric profile approach is to compare metrics from design classes with metrics of implementation classes. Metrics of the first category only exist for the design. There is no implementation metric that is suitable for equality comparison. A similar argument holds for the second category.

This does not mean that only metrics of the third category are useful. Metrics need not be compared on equality. We are interested in metrics that *correlate*. It is possible to investigate the correlation between every pair of a design metric and an implementation metric. If there

is a high correspondence between design and implementation, one would of course expect design metrics of the third category to correlate very well with their implementation variants.

### 5.1.1 Correlation analysis

The goal in this section is to find correlations between design metrics and implementation metrics.

Given are a design metric  $M_D$  and an implementation metric  $M_I$ . We want to know whether  $M_D$  and  $M_I$  correlate.  $M_D$  can be calculated for each class in the design.  $M_I$  can be calculated for each class in the implementation.

Suppose there exists a – partial – matching between classes in the design and classes in the implementation. The dataset for the correlation analysis is formed by pairs of matching classes. With this dataset, it is then possible to draw a scatter plot. Figure 5.1 shows one.

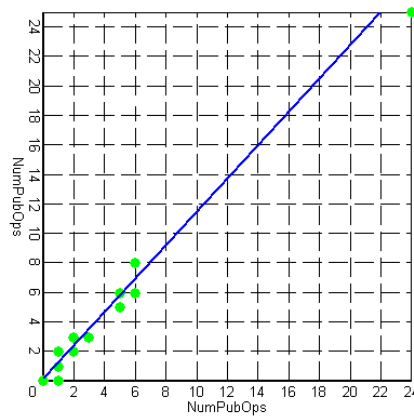


Figure 5.1: A scatter plot of two metrics

The horizontal axis shows the design metric. For figure 5.1, this is the number of inherited operations. The vertical axis shows the implementation metric. In this case this is also the number of operations inherited. Each dot in the scatter plot represents a pair of a design class and a related implementation class. The horizontal position shows the metric value for the design class. The vertical position shows the metric value for the implementation class.

In figure 5.1, the metrics are clearly correlated. A scatter plot can only suggest correlation. To validate a suggestion of correspondence, statistical analysis is needed.

The strength of the correlation can be assessed by the correlation coefficient  $\rho_{M_D, M_I}$ . The correlation coefficient is a value ranging from -1 to 1. If the value of the correlation coefficient is close to one of the extremes, the correlation is strong. If the value of the correlation coefficient is close to zero, there is little or no correlation.

The correlation coefficient can be calculated in different ways. The most well known variant is the Pearson method [MR03]. This is the method discussed in most basic statistics courses. It is calculated by the formula in equation 5.1. In this formula  $E$  represents the expected value.  $\mu_D$  is the mean value of the design metric and  $\mu_I$  is the mean value of the implementation metric.  $\sigma_D$  and  $\sigma_I$  are the standard deviations of the design metric and the implementation metric respectively.

$$\rho_{M_D, M_I} = \frac{E(M_D M_I) - \mu_D \mu_I}{\sqrt{\sigma_D^2 \sigma_I^2}} \quad (5.1)$$

Fenton [FP97] argues that it is incorrect to use the Pearson correlation coefficient if the data is not distributed normally. In general metric data is not distributed normally. Fenton suggests the use of ranking correlation coefficients such as the Spearman rank correlation coefficient or the Kendall robust correlation coefficient [Con80]. This argument is opposed by Briand et al. [LB96]. They state that this restriction proposed by Fenton is too strict. There is no agreed opinion about what correlation coefficient to use.

We calculated metrics for the design and the implementation of a piece of software. For this experiment it turned out that the Pearson and the Spearman correlation coefficient produced similar results. We therefore decided to use the Pearson correlation coefficient since it is easiest to calculate.

### 5.1.2 Regression analysis

If metrics are highly correlated, the data points are scattered around a straight line. This line is called the regression line. This line is determined by the equation:

$$y = \beta_0 + \beta_1 x \quad (5.2)$$

Given the value  $x$  of a metric of some class in the design, it is possible to predict the value of the correlating implementation metric. This is done as prescribed by equation 5.3.

$$E(M_I|x) = \beta_0 + \beta_1 x \quad (5.3)$$

The parameters  $\beta_0$  and  $\beta_1$  can be determined by the least squares method [MRH04]. This method determines  $\beta_0$  and  $\beta_1$  in such a way that the sum of the squared distances between the data points and the regression line is minimized.

### 5.1.3 The composition of a metric profile

In section 5.1.1 we discussed how to determine whether a design metric  $M_D$  correlates with an implementation metric  $M_I$ . There exist many metrics in the design and implementation. It is possible to analyze each combination of a design and an implementation metric for correlation. This results in a list of correlating metrics.

With this list we can create metric profiles for the classes in the design and the implementation. The metric profile of a class is a vector of values of different metrics for this class. For design classes, the vector will consist of design metrics. For implementation classes, the vector consists of implementation metrics.

Another requirement is that metrics from the design metric profile correlate pairwise with the metrics from the implementation metric profile. That is, the  $k$ -th metric in the design profiles correlates with the  $k$ -th metric in the implementation profiles.

## 5.2 Using metric profiles for similarity measurement

The idea of the metric profile is that it makes different classes in a model distinctive. So two classes in the design are different because the values in their metric profiles are different.

The metrics in the design profile correlate pairwise with metrics in the implementation profile. Suppose we have the metric profile  $\vec{M}_D$  of a design class  $D$ .

Using regression analysis, it is possible to predict an implementation metric profile  $\vec{M}_P$ . So the  $k$ -th component of  $M_P$  is estimated using the  $k$ -th value from  $M_D$ . This makes sense since each of the metrics in the design profile correlates pairwise with a metric in the implementation profile.

A pair of a design metric and an implementation metric can only be in the metric profile if their correlation is high. That is, the correlation coefficient should be close to 1 or -1. A class  $I$  from the implementation can only match with  $D$  if the metric profile  $\vec{M}_I$  of  $I$  resembles the predicted profile  $\vec{M}_P$ .

To see whether the predicted metric profile resembles a real metric profile from the implementation, we use the fact that metrics in the metric profile are pairwise correlated.  $\vec{M}_P$  is the metric profile that we expect the perfect matching class from the implementation to have. We thus compare  $\vec{M}_P$  with  $\vec{M}_I$  by calculating the differences in the values. These are the errors. Figure 5.2 shows this method for one of the components of the metric profile vector. The smaller the errors the better  $\vec{M}_I$  resembles  $\vec{M}_P$ . In this way, the predicted profile of a design class can be compared with every class from the implementation.

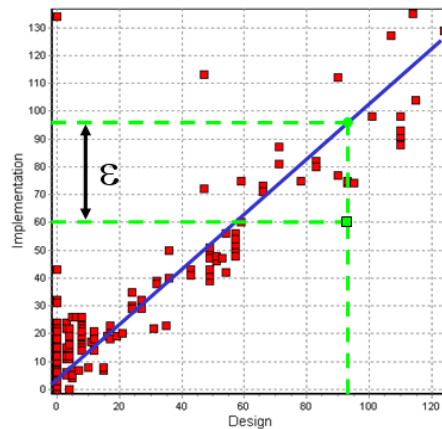


Figure 5.2: Comparing the value predicted from a design metric with a real implementation metric value results in an error  $\epsilon$ .

For the metric profile method to work, there are two requirements that must be met:

1. The metric profiles should be *distinctive*. This means that any two classifiers in a single model should have different profiles. A metric profile should be like a finger print. People can be identified by their finger print. Likewise, the classifiers should be identifiable by their profile.
2. The metric profiles should be *predictive*. This is used when matching design classifiers to implementation classifiers. This requirement implies that the metrics in the metric profiles should be highly correlated. Only if that is the case, regression analysis can be used to predict the profile of an implementation class that fits best to the design class.

These two requirements can be contradictory. The more metrics are added to the metric profiles, the more distinct the profiles will be. The second requirement states that only highly

correlating metric pairs should be added to the profile. If there are only few highly correlating pairs of metrics, it is impossible to fulfil both requirements. In such cases the metric profile approach will not produce reliable results.

## 5.3 Using metric profiles for outlier analysis

There is another purpose for metric analysis. Scatter plots can be used to identify suspicious classes in the matching. In scatter plots of correlating metrics, most data points are scattered around the regression line. Some of them are relatively far away from the regression line however. These classes do not follow the trend. This does not automatically mean that there are differences but it does make the class suspicious.

### 5.3.1 Significance

Not all scatter plots are suitable for scatter plot analysis. The metrics should correlate. If they don't, there is no trend that is followed by the classes that are not outliers. As a result it is impossible to distinguish outliers from inliers.

But correlation alone is not enough. Correlation in small data sets is less valuable than correlation in large data sets. It is easier to score high on correlation with a small data set. Large data sets usually also have a broader value range. These two properties make correlation more trustworthy. In statistics a so called *p-value* expresses the significance of the correlation. The closer this value is to zero, the more reliable the correlation is.

### 5.3.2 The 95% confidence interval

Outliers are relatively far away from the regression line. Items that are not exactly on the regression line are not necessarily outliers. In fact, most of the data points are not exactly on the regression line. They are usually scattered around this line. The further an item lies from the regression line, the higher the probability that it is an outlier. How far should an item be from the regression line to be an outlier?

A  $(100 - \alpha)\%$  confidence interval can be used to identify real outliers. The  $(100 - \alpha)\%$  confidence interval can be marked in the scatter plot. All classes inside this interval are not considered. The classes outside the interval are.

A confidence interval is only useful if the p-value is close to zero. Otherwise the confidence interval becomes very large. Figure 5.3 shows a scatter plot with a trendline and a 95% confidence interval.

### 5.3.3 Single scatter plot analysis

Using scatter plots in combination with significance values and confidence intervals it is possible to identify the real outliers. Figure 5.3 shows a scatter plot enriched with a regression line and the 95% confidence interval.

In this example, the correlation coefficient is 0.9490 and the significance is 0.0005. It is thus reasonable to assume that the metrics in the plot indeed correlate.

One of the classes is outside the confidence interval. Since the correlation is significant, this class is a real outlier. The confidence interval allows us to identify these classes easily.

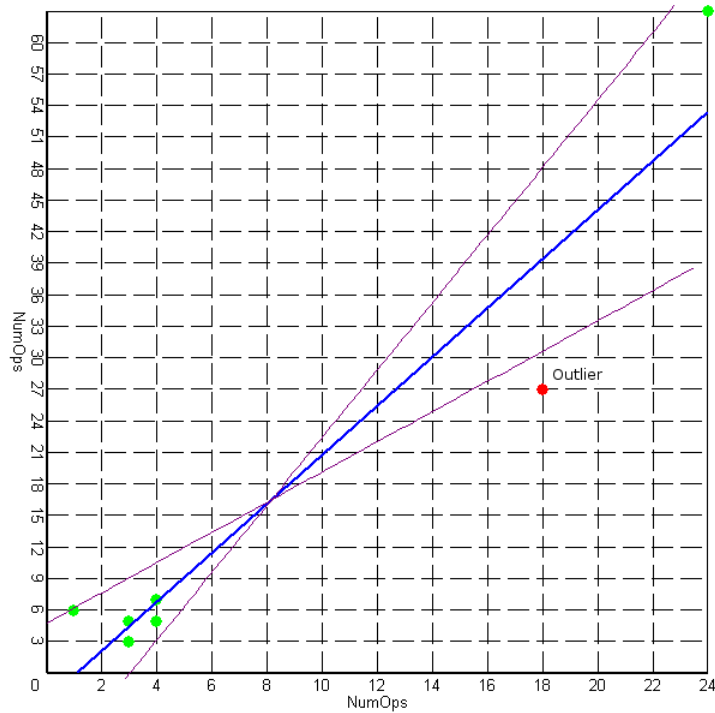


Figure 5.3: A scatterplot with a regression line and a 95% confidence interval can be used to identify outliers

### 5.3.4 Multiple scatter plot analysis

The scatter plot analysis can be even more effective if different scatter plots are drawn adjacent to each other. It is for instance interesting to see the scatter plots of all metrics in the metric profiles.

A class that is an outlier in different scatter plots, is extremely suspicious. By drawing different scatter plots and highlighting one of the classes in each of these diagrams, it is easy to see if and where the class is an outlier. This is depicted in figure 5.4.

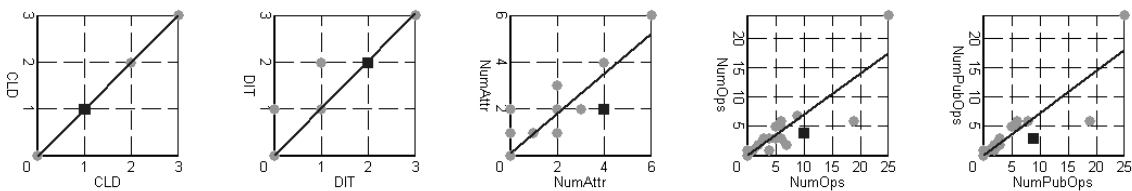


Figure 5.4: Different scatterplots in one view are useful for finding outliers

In this figure one class is highlighted. This is the squared dot. In two of the diagrams it is on the regression line so there it is definitely not an outlier. In two other plots it is away

from the regression line. In the latter two plots it is somewhat far away from the regression line. The confidence intervals have not been drawn in the pictures but the selected class is outside the interval in both of these diagrams. The third plot should not be taken too seriously. The range of the data is very small. In a small data range outliers are less obvious. We conclude that the class is not an outlier.

## 5.4 Conclusion

By correlation analysis it is possible to compare any pair of metrics. Once it is determined which metrics correlate, it is possible to create metric profiles. The metric profile should be a vector of values that is distinctive for each class in a model.

Since the metrics of the design profile correlate pairwise with metrics from the implementation profile, it is possible to compare design classes with implementation classes based on their metric profiles. It is possible to predict an implementation profile based on a design profile. This is the predicted profile. Classes are said to be similar if the predicted profile resembles the profile of an implementation class.

Before this comparison can be made, we need to determine which pairs of metrics are correlated. There are some pairs of metrics that correlate in many cases. Especially metrics that can be calculated for both design and implementation should correlate in any case. There can also be other pairs of correlating metrics. These are case specific however. Therefore we need a partial matching to exist before we can use the metric profile approach for matching. Based on this partial matching the correlating metrics are identified and the regression lines are determined.

Not only metrics that are available in both design and implementation, are useful. There are, for instance, various ways to measure complexity. In a design complexity can be expressed by the Complexity of services metric [Mus02]. This metric is difficult to measure in the implementation. In the implementation complexity can be measured by the lines of code metric however. These two metrics both measure complexity but they do it in different ways. The metrics can thus not be compared based on equality. However, if the design corresponds to the implementation, these metrics should correlate.

In this thesis we focus on structural aspects of classes since the representation of behavior is different in design and implementation. It is therefore difficult to compare designed behavior with implemented behavior. Correlation analysis provides a way to compare behavior. The behavior of the design and the implementation can be expressed in terms of metrics.

Metrics can also be used for searching suspicious classes. Suspicious classes show critical deviations between design and implementation. In scatter plots these classes are likely to stand out because they are far away from the regression line. We define far away by outside the 95% confidence interval.



# Chapter 6

## Tooling

This chapter describes the tooling used for assessing correspondence. First a number of important requirements are listed. These requirements form the basis for the design of the tool. The chapter finishes with some implementation notes.

### 6.1 Key requirements

This section lists the most important requirements for the correspondence tool. Only key requirements are listed; the list is by no means complete. The requirements are assigned a priority indicating their importance to the project. Table 6.1 lists the meaning of the priorities.

Priority	Meaning
1	The tool <i>must</i> fulfill this requirement in order to be of any use.
2	The tool <i>should</i> fulfill this requirement.
3	The tool <i>could</i> fulfill this requirement.
4	The requirement is <i>nice to have</i> . Requirements with this priority are very likely not met within this project.

Table 6.1: Priority levels of the requirements

#### 6.1.1 Functional requirements

##### Input requirements

###### REQ1 (1)

The input for the toolset consists of 2 UML models in XMI v1.1 format.

###### REQ2 (1)

Elements of the UML models may but need not be enriched with software metrics.

##### Correspondence checking requirements

###### REQ3 (1)

The tool searches for a matching between the 2 given models. The matching is at the level

of classifiers.

**REQ4 (1)**

The tool includes algorithms for matching based on the approaches described in chapter 4.

**REQ5 (1)**

The tool includes at least one algorithm that can make a one-to-one matching.

**REQ6 (4)**

The tools includes a clustering algorithm that can make one-to-many or many-to-many matchings.

### 6.1.2 Constraint requirements

#### Interface requirements

**REQ7 (3)**

The tool integrates with MetricView.

**REQ8 (1)**

The user is able to choose which approaches for matching are used.

**REQ9 (2)**

The user interface of the tool is at all times responsive. Lengthy operations are not allowed to block window drawing.

**REQ10 (2)**

In case of lengthy operations feedback to the user is given via for instance a progress bar.

**REQ11 (2)**

The language of the user-interface must be English

#### Maintainability requirements

**REQ12 (2)**

It should be easy to implement new algorithms for calculating similarities.

**REQ13 (2)**

It should be easy to implement new algorithms that can match based on similarity value matrices.

## 6.2 Tool architecture

This section presents an architecture for the correspondence toolset. First a global overview is presented. Then the components are described in more detail.

### 6.2.1 Global overview

#### Correspondence phases

Figure 6.1 shows the process of correspondence checking. The process consists of three phases.

**The data extraction phase** This part is actually a preparation phase. In this phase the source code and the design model are converted into a format that is suitable for the analysis. Furthermore, metrics are calculated and stored.

The result of this phase is a database with all data that is required in following phases.

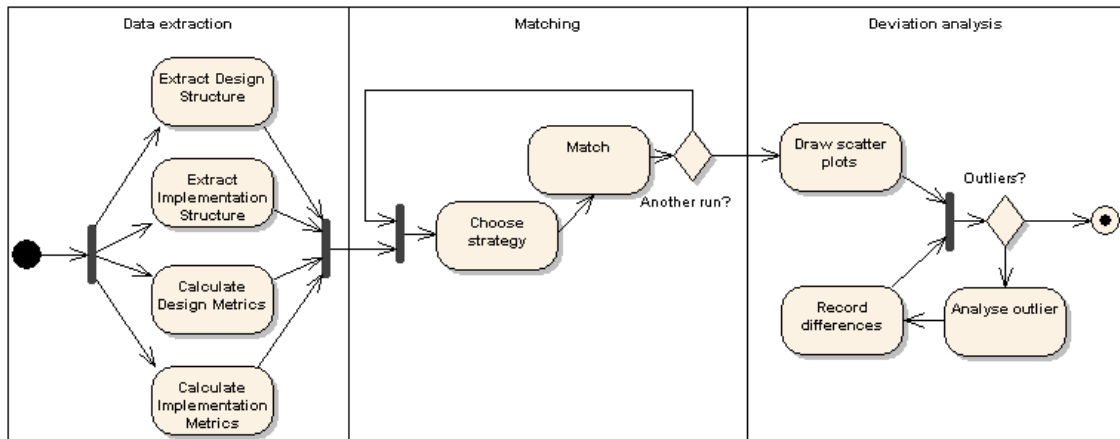


Figure 6.1: The correspondence assessment process

**The matching phase** In chapter 4 we presented methods for matching classes in the design to classes in the implementation. Furthermore we proposed to follow an iterative improvement approach. In each iteration a matching approach is chosen and executed. After this, it is decided whether or not to do another run.

This phase results in a – partial – matching between design and implementation.

**The deviation analysis** Once the matching is established, we can search for differences. In chapter 5 we proposed to use scatter plots for this purpose.

The phase starts with drawing interesting scatter plots. Interesting scatter plots show high correlation and have a low p-value.

From these scatter plots it is possible to identify the outliers. In this project we will identify the outliers manually but it is not very difficult to find outliers automatically.

### High level design

The phases of the correspondence assessment are a good initial decomposition of the tool architecture. However, there are some components, such as the database, that are used in each phase. Commonly used components will be in another package called common. Common dialog windows will be in the separate package DICDialog however. This results in the decomposition shown in figure 6.2.

All parts depend on the Common package as this package contains the common functionality. Furthermore each phase package depends on its predecessor. This is because the results of a phase are used in the next phase. The analysis phase also depends on the extraction phase since the metrics calculated in the extraction phase are used here.

The tools depend on three existing libraries:

- **wxWindows** is a user interface library that increases the portability of applications with graphical user interfaces.
- **Xerces C** is an XML parser. It is used to parse XMI files.
- The **MySQL** DBMS is used to implement the data model. The package includes a C library for client applications.

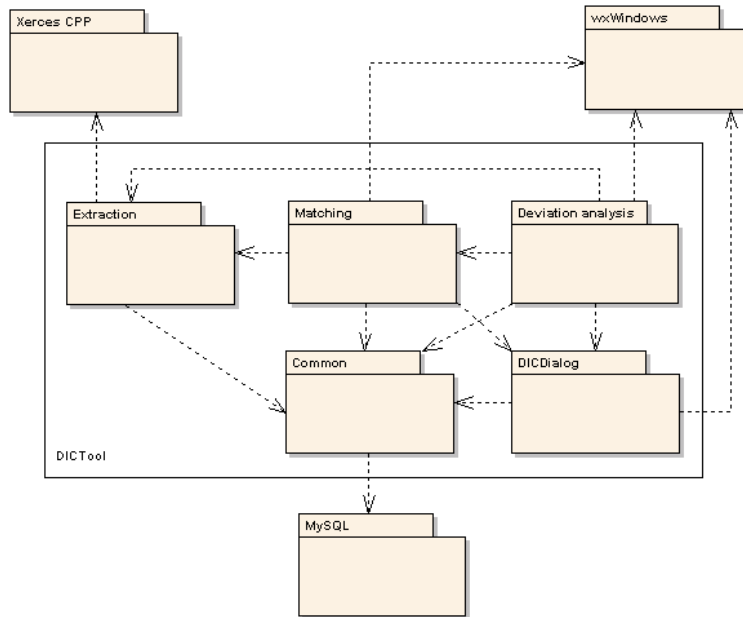


Figure 6.2: The high level design of the correspondence toolset

### 6.2.2 The database model

The result of the data extraction phase was a database with all necessary information. Figure 6.3 presents a data model for this.

The database stores a part of the structural information from the XMI models. The tables `Package`, `Classifier`, `Operation`, `Attribute` and `Relation` are used for this purpose.

#### Versioning

Both the `Package` and `Classifier` tables store a version attribute. This indicates from which artifact the entry originated. When analyzing the correspondence between a design and an implementation, the design entities would get version 0 and the implementation entities would get version 1 for instance.

An alternative would be to create separate classes for design entities and implementation entities. There are two reasons for not doing this:

- Using a version attribute allows us to abstract from the number of artifacts. We are not limited to exactly two artifacts. There are cases where there are more than two versions of a system available. There can for instance be two versions of the design and an implementation. The version attribute allows us to store information about all artifacts and not just two of them.
- The version attribute keeps the data model small and simple. If we would have introduced a table for design classifiers and another table for implementation classifiers, we also had to make copies of the `Operation`, `Attribute` and `Relation` tables.

With versioning there is no need to do this. An attribute, for instance, belongs to exactly one class. Thereby it is automatically bounded to a version. For relations we



The scatter plotter tool allows to store metric pairs that are for some reason interesting. Such pairs are stored in the `MMatch` table.

The `CM` table, stores metric values. Each entry in the table stores the value of one specific metric for one specific class. This table is actually a helper table to get rid of the many to many relationship between classes and metrics.

## Matching

The matching of elements is done at the level of classifiers. The `CMatch` table stores matchings between different versions.

Matchings can not exist within a version. Classes that are being matched should come from different versions.

With each entry in the `CMatch` table it is possible to store a reliability value. This indicates how reliable the match between the two classes is.

## Database management system

This data model can be implemented in many ways. We want to run SQL queries on the database. Therefore we will use an SQL database. We chose to create a MySQL database. Our motivations are the following:

- MySQL is an open source database management system. No license costs are involved.
- ODBC drivers are available. Many RAD tools communicate with databases through ODBC. In section 7.3.1 we discuss the need for RAD tools.
- There is good support for C++. Our tools are implemented in C++. MySQL comes with a library for C and C++. This library is more efficient than an ODBC connection.

### 6.2.3 Common tools

The common tools include some generic functions and classes. Most of these are related to database access. Since all tools somehow interact with the database, the database access is centralized in the common part. Figure 6.4 shows the MySQL connector.

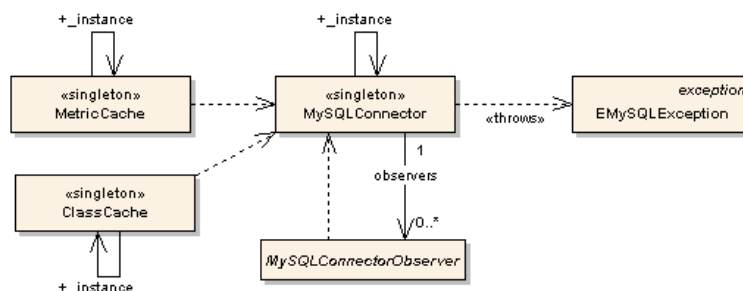


Figure 6.4: The MySQL connector and entity caches

The `MySQLConnector` class makes use of the observer pattern. It is possible to change the database at runtime. If the database changes, the data changes obviously. The observers

are notified to refresh their data view. It is not necessary to have more than one database connector in an application. Therefore this object is implemented as a singleton.

For the metrics and the classifiers in the database, caches are made. These caches store the most important data of the classes and metrics. The names and primary keys are examples. The data in these caches is used very often in the other components. It is too inefficient to query the database each time this data is needed.

#### 6.2.4 Common dialogs

The database settings dialog and the classifier properties dialog occur in both the matcher and the scatter plotter. These are called common dialogs.

Since these are a common feature, they could be added to the Common package. These dialogs depend on wxWindows. The common package would also depend on wxWindows. This is undesired since the common functionality is also used by applications that have no wxWindows functionality at all. For this reason, the common dialogs are gathered in a separate package.

#### 6.2.5 Data extraction tools

For data extraction we use a few existing tools. In chapter 2 these tools were discussed. We need an XMI model for both the design and the implementation. Apart from that, we need metrics for both artifacts.

#### Processing XMI data

The structure of both the design and the implementation is stored in an XMI file. These files must be parsed. The interesting information, such as classes, operations, packages and relations must be stored in the database. For this we have to create a small tool. Given an XMI file it stores the information in the database. Figure 6.5 shows the structure of this tool.

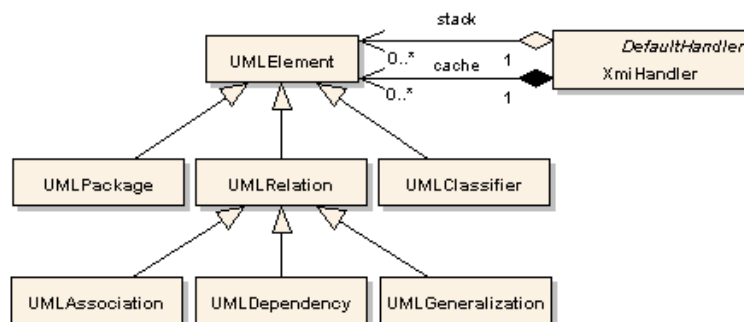


Figure 6.5: The XMI processor design

The parser uses the Xerces C library for parsing. This library supports DOM parsing and SAX parsing. DOM parsing builds a parse tree in main memory. This tree can be traversed afterwards. A SAX parser fires events for each XML tag found. The XML tags can be processed immediately once they are encountered in the source file.

Since XMI files can grow over hundreds of megabytes, SAX parsing is preferable. The only problem in XMI files is that objects can be referenced before they are defined. For instance,

a dependency relation between two classes can occur in the file before the classes themselves occur. The unique ID of a class is not known until the class is actually added to the database. Therefore we must maintain an object cache which maps XMI ID's to cached objects.

### Processing metric data

Metrics are calculated using existing tools such as SDMetrics, Columbus/CAN or SAAT. Each of these tools produces an output file in which the metric data is stored. SDMetrics and Columbus/CAN produce a comma separated value file. SAAT produces a custom made ASCII file.

We want the tool to read metrics from many different sources. For the project we focus on comma separated value files. The MetricView sources contain a parser for SAAT metrics. This parser can be rebuilt to fit in the model. Figure 6.6 shows the parser hierarchy.

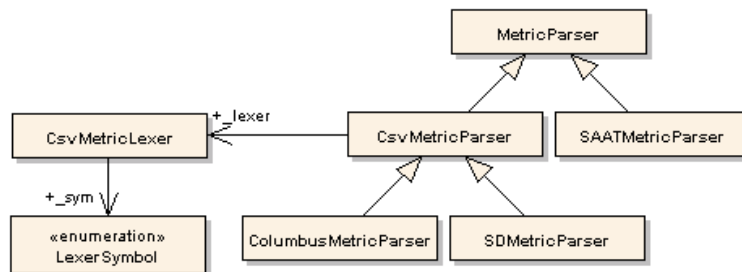


Figure 6.6: The Metric processor design

### 6.2.6 Correspondence matching

The matcher application is used for matching design classifiers to implementation classifiers. The matching can be performed automatically by using one of the built-in strategies or manually.

#### The User Interface

To visualize the matching, the method discussed in section 4.5 is used. The main window consists of a grid where the design classes are listed vertically and the implementation classes horizontally. Each of the grid cells is clickable so that the user can change the matching manually. The structure of the user interface part is depicted in figure 6.7.

**wxMatchDataset** This class represents the data to be shown in the match grid. It is almost an abstract class. It only implements a registration interface for data observers.

**wxMatchMySQLDataset** The data we present in this project is fetched from a MySQL database. **wxMatchMySQLDataset** partially implements **wxDataset** for MySQL data.

**wxClassMatchDataset** The abstraction of the data allows us to visualize different views. The initial idea was to visualize the class matching. **wxClassMatchDataset** implements a data set that can be used for this.

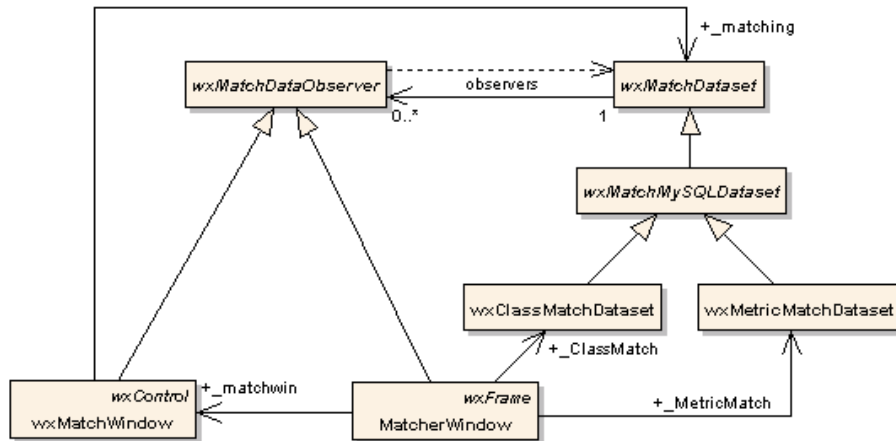


Figure 6.7: The user interface design of the matcher application

**wxMetricMatchDataset** There are metrics between design and implementation that are expected to correlate. These metric pairs are stored in the `MEquality` table. This is also a matching. For this matching we can also draw a grid. `wxMetricMatchDataset` provides the data for this.

**wxMatchWindow** This class is the widget which draws the grid. It responds to mouse clicks. The data that is drawn is provided by `wxMatchDataset`. This abstraction allows us to draw grids for different relations.

**wxMatchObserver** Both `wxMatchDataset` and `wxMatcherWindow` are observers of the dataset. If something changes in the dataset, they update their views.

### Automatic matching

The matching tool should support automatic matching. There are different strategies for automated matching. In this thesis we presented a few.

In chapter 4 we proposed to use a bipartite matching algorithm for matching. We use similarity values as weights for the edges. Then automatic matching consists of two steps: calculating similarity values and finding a maximal matching in the data.

For both of these tasks we need to implement algorithms. For calculating similarity values we proposed several approaches in chapters 3 and 5. We can create a family of similarity and matching algorithms in this way. This is an instance of the strategy pattern described in [GHJV00]. This results in the design of figure 6.8.

The algorithms in the strategy pattern should be instantiated in the program. Usually this is done by some kind of factory. We want a factory as well but implementing new algorithms shouldn't be too complex. Ideally no existing code should be changed for this purpose. With a dynamic factory this is possible. A dynamic factory [Cod04] is a factory where algorithms register themselves. The implementation of such a factory uses some advanced C++ language constructs.

The user can select similarity and matching algorithms for a matching iteration. Such an iteration is managed by a so called assessor. The user sets the algorithms to run at the

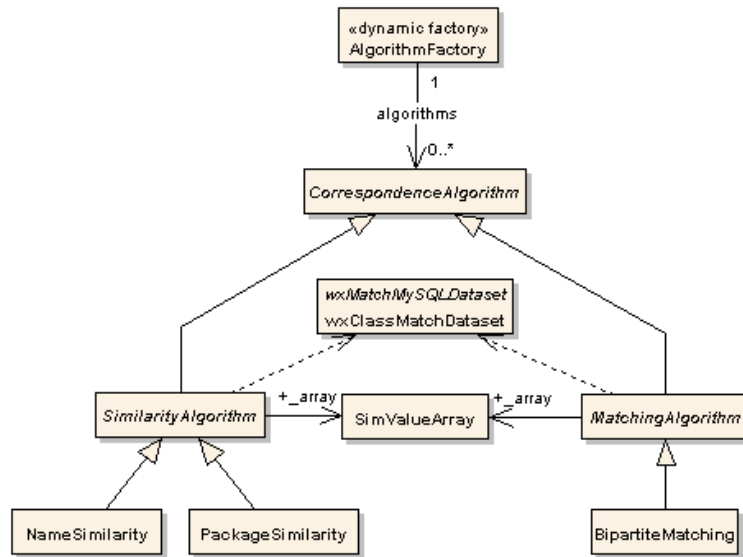


Figure 6.8: The matching framework

assessor. The assessor will return references to instances of the algorithms to the user. The user can use these to set parameters. This is depicted in part I of the sequence depicted in 6.9.

If the user is satisfied with the selected algorithms and the parameters, he tells the assessor to execute. This is shown in part II of the sequence diagram. The assessor will then perform the following steps:

1. Each of the similarity algorithms will be executed;
2. The resulting similarity matrices of each of the similarity matrices will be composed into a single similarity matrix. This is done using the composite similarity primitive from section 3.4.1.
3. The composite similarity array is provided to the matcher algorithm which will be executed.
4. The result of a matching algorithm is a – partial – matching. This partial matching is added to the global system matching.

### Implemented algorithms

In the prototype of the toolset we implemented a number of similarity and matching algorithms. The hierarchy of algorithms is depicted in figure 6.10.

**MetricSimilarity** This algorithm calculates similarity values between classes using metric profiles. The database must contain a list of metrics for the metric profiles in the `MMatch` table.

**OperationSimilarity** Similarity values are calculated by comparing the sets of operations of classes.

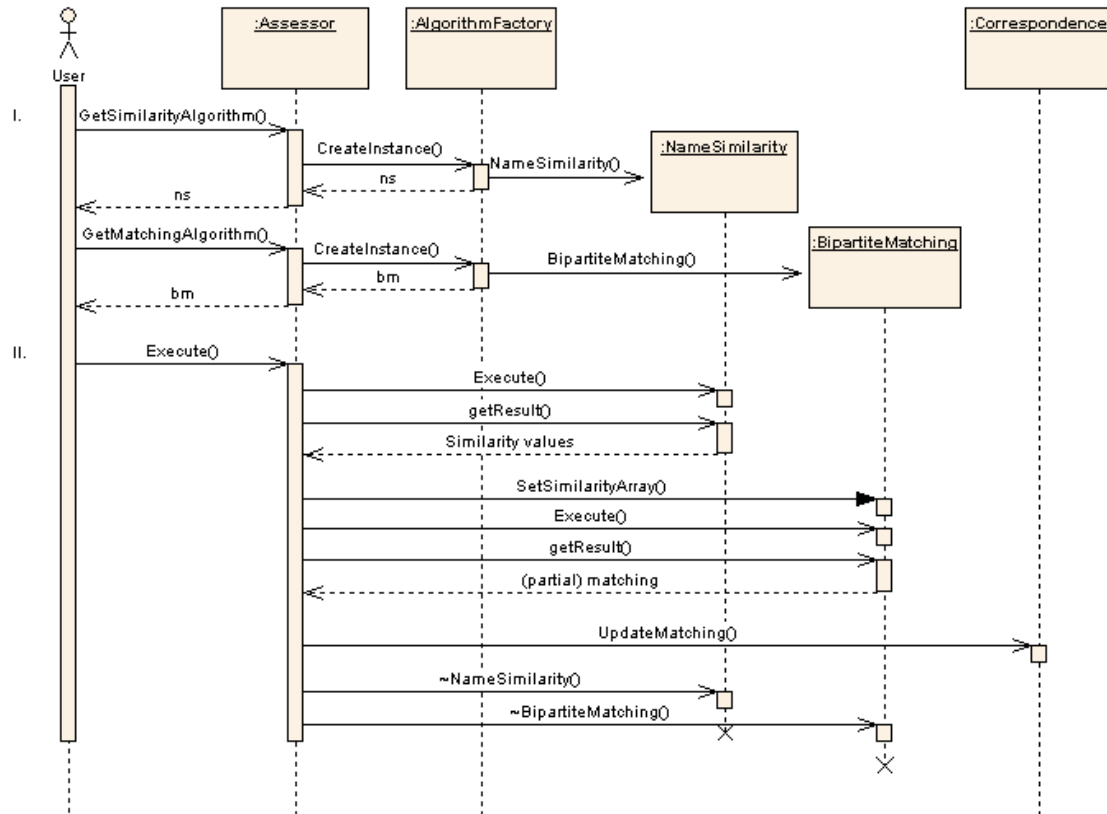


Figure 6.9: The sequence of a typical assessment run

**NameSimilarity** This algorithm calculates similarity values using the name primitive from section 3.4.1 on the names of the classifiers.

**InheritanceSimilarity** The similarity values are calculated by checking if the same inheritance relations exist between two classes.

**AttributeSimilarity** The similarity of the attribute sets of two classes is used here to calculate similarity values.

**PackageSimilarity** This algorithm sets the similarity value of two classes to one if they are in related packages. A partial matching must exist for this algorithm to work. It uses the existing partial matching to recognize which packages are related.

**HighSimilarityMatcher** The high similarity matcher registers a match for each pair of classes for which the similarity exceeds a threshold.

**BipartiteMatching** Given the similarity matrix this algorithm, searches a bipartite matching between design classifiers and implementation classifiers such that the weight (similarity values) of this matching is maximized.

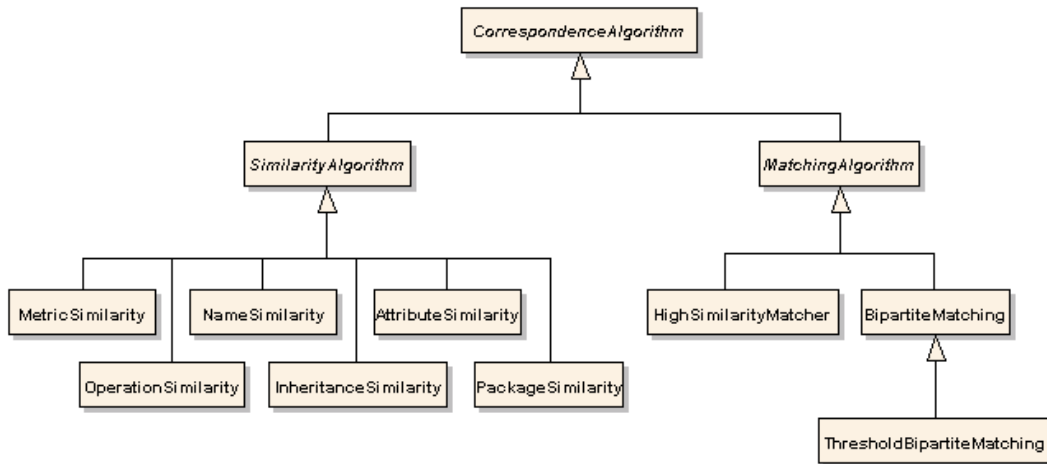


Figure 6.10: The algorithms implemented in the toolset

**ThresholdBipartiteMatching** This algorithm works like the bipartite matching algorithm but it ignores edges for which the similarity value is below a threshold.

## 6.2.7 Difference analysis

### Functionality

In chapter 5 we discussed the use of metrics for identifying outliers. In order to do this analysis, we need to be able to draw scatter plots quickly. So we created the scatter plotter tool.

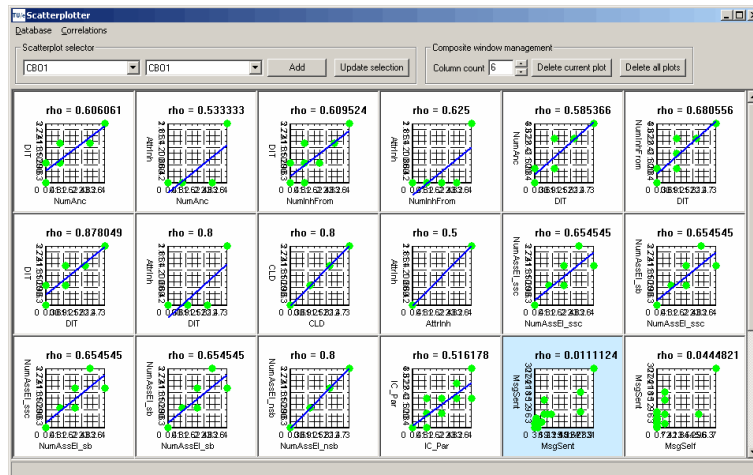


Figure 6.11: The scatterplotter tool showing different plots

Figure 6.11 shows the main user interface of the tool. The upper part of the window contains to selection boxes where a design metric and an implementation metric can be selected. In the bottom part of the window the scatter plots are displayed.

Classes, drawn in the scatter plots, should be selectable. If a class is selected in one plot,

it is also highlighted in all other plots. If multiple classes coincide in a plot, a dialog should appear to let the user choose the desired class.

If the user clicks a class with the right mouse button, its properties should be retrieved from the database. A dialog is displayed that presents information about the structure of a class and its metrics in both design and implementation.

## Design

Figure 6.12 shows the decomposition of the tool into classes. Only the most important local classes are displayed.

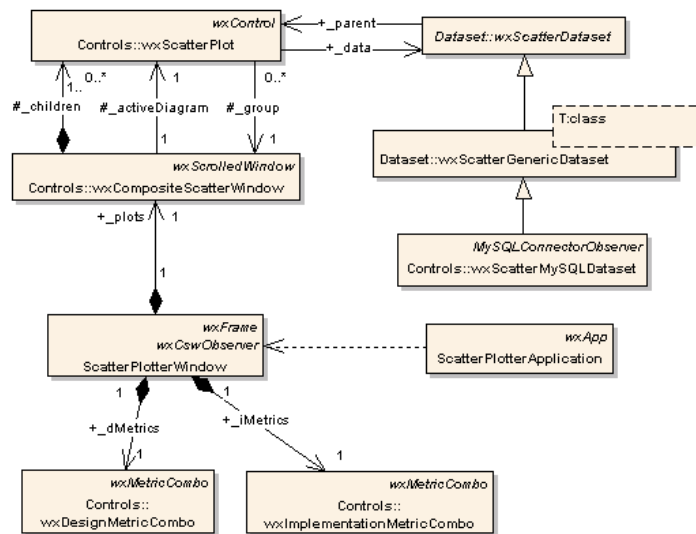


Figure 6.12: The design of the scatter plotter tool

**wxScatterPlot** This class inherits from `wxControl`. This class represents the graphical representation of a single scatter plot. Apart from classes, it can also display the regression line and the confidence interval. The user can select the data points in the scatter plot.

**wxScatterDataset** This class represents a data set to be displayed in a Scatter plot window. We chose to create an abstract class. This enables us to use the scatter plotter for other types of data later on.

In the `wxScatterDataset` class we can not make any assumption about the type of data except for the fact that it is an ordinal type. This means that we can visualize integral data and floating point data. But we can also visualize anything else as long as it is ordinal.

**wxScatterGenericDataset** This class provides implementations for the `wxScatterDataset` class. It is a template class so for many types a data set can be obtained by simply instantiating this template class.

It is not possible to merge `wxScatterDataset` and `wxScatterGenericDataset` into a single class. The reason is that `wxScatterGenericDataset` is a template class. The class can be instantiated by assigning a type to the template parameter. If the class is instantiated

with different types, one actually gets different implementations. In [Str91] the compilation of template classes is discussed in more detail.

**wxScatterMySQLDataset** This class inherits from a `wxScatterGenericDataset` template using datatype `double`. The data is obtained from the MySQL database which explains the name of the class.

**wxCompositeScatterWindow** The `wxCompositeScatterWindow` class serves as a container for `wxScatterPlot` objects. It inherits from the `wxScrolledWindow` class. This makes it easy to implement scrolling if the scatter plots don't fit in the window.

**ScatterPlotterWindow** This class implements the main application window. This window contains the composite scatter plotter window, the metric selectors and the menubar.

**ScatterPlotterApplication** `wxWidgets` requires to create an application class. It has two methods: `OnInit` and `OnExit`. The first is called when the program is started and the last is called if the application terminates. Usually the `OnInit` method is used to create a main window.

**wxDesignMetricCombo, wxImplementationMetricCombo** These two classes inherit from the `wxComboBox` widget. When constructed, they automatically retrieve a list of metrics from the database. The `wxDesignMetricCombo` gets design metrics and the `wxImplementationMetricCombo` gets implementation metrics.

## 6.3 Implementation notes

The purpose of this section is not to discuss the implementation in detail. However some parts of the implementation might require some further explanation.

### 6.3.1 Common queries

The tools run various queries on the MySQL database. We don't want these queries to appear anywhere in the code since updates in the data model become difficult in that way. All queries are located in the files named `CommonQueries.cpp` and `CommonQueries.h`. These files are part of the `Common` package.

# Chapter 7

## Case studies

### 7.1 Case characteristics

We tested the methods described in this thesis on various industrial case studies. Table 7.1 shows the characteristics of the cases we analyzed.

All designs used the UML notation. Different CASE tools were used for the designs. Different CASE tools produce different XMI output formats. For different CASE tools, different steps must be taken to convert the XMI data into the proper format. The number of classifiers and the numbers of diagrams give an indication of the size of the design.

The implementation language is again an important fact. This decides which reverse engineering tool is used to convert the code into an XMI model. The number of classifiers and lines of code give an indication of the size of the project.

<i>Case</i>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>	<b>G</b>
<i>Design</i>							
UML Case tool	Rose	Together	Visio	TL TAU	Rose	Rose	Rose
# classifiers	4	19	14	46	45	416	45
# use case diagrams	0	0	0	1	4	30	4
# structure diagrams	1	1	3	4	14	343	14
# behavior diagrams	0	0	28	0	25	546	28
<i>Implementation</i>							
Language	C#	C	C++	C++	C++	C++	UML
# classifiers	5	22	19	118	111	661	46
Lines of code	1,700	NA	4,987	15,517	NA	169,889	NA

Table 7.1: The characteristics of the industrial case studies

The first six cases compare a design with its implementation. The seventh case compares two versions of a design.

### 7.2 Matching validation

In this thesis we presented various methods for calculating similarity values and matching. We use some of the case studies to see how good the methods are. We started by defining a

matching manually. We applied the methods on two of the case studies and an artificial case T. We compared the results of the methods with our manual matching.

Each of the matching methods produces a similarity matrix as described in chapter 4. The matching algorithm was the high similarity matcher. This means that classes are matched as soon as their similarity exceeds some threshold. The threshold was case and strategy specific.

We used the following configurations:

- **Matching on names.** We used the name similarity algorithm for this.
- **Matching on structure.** Here we used a combination of the name similarity, attribute sets and operation sets algorithms. Each of these similarity algorithms was assigned the same weight.
- **Metric profile approach.** Before using this approach, we had to select metrics for the metric profile. We used the manual matching for this. We selected the metrics using the scatter plotter. Then we erased the matching and tried to reconstruct it using the metric profile algorithm.
- **Local metric profile approach.** Here we applied a combination of the package matching approach and the metric profile approach. The purpose of this combination was to see whether the matching improves if package information is used as described in section 4.4.2.

For case T, we filled the design and the implementation with exactly the same data. This means that there exists a perfect one to one match between design and implementation for case T.

The case studies are E and G. G was the case for which we had two versions of a design. Case E was the largest case where big differences exist between design and implementation. The results are listed in table 7.2. The result is expressed by the match ratio. Every pair of a design classifier and an implementation classifier can either match or not. Let  $N_D$  be the number of classifiers in the design and  $N_I$  the number of implementation classifiers. Then the match ratio is the number of correct answers as percentage of  $N_D \cdot N_I$ .

The artificial case T allows for a perfect matching. There are no deviations since the implementation is exactly the same as the design. Both structure matching and name matching result in a perfect match in such a case. The metric profile approach makes mistakes already. Since the design and implementation are the same, the metrics are also the same. We selected only metrics in our profile that have a correlation coefficient of 1. The metric profile approach resulted in 523 false matches. Including package information helps to reduce the number of errors but still 122 false positives remain.

Case E is one of the case studies. In this large case there are many deviations. Structure matching performs really poor. Many operations and attributes were introduced and removed in this case. It is thus hard to use this information to match elements. The metric profile approach performed the worst of all. Since we did not have the package information in the implementation, there is no improvement when package information is used to localize the matching.

Case G were two versions of a detailed design. Matching on names beats the other methods by far again. The matching was very good. With only a few manual improvements, the matching is complete. Matching on structure found much less results. Again the metric

<i>Case</i>	<b>T</b>	<b>E</b>	<b>G</b>
Number of design elements	253	416	109
Number of implementation elements	253	661	118
Number of matches in manual matching	253	364	105
<i>Matching on names of classifiers</i>			
Number of true positives	253	351	87
Number of false positives	0	84	1
Number of false negatives	0	13	18
Match ratio	100%	99.9%	99.8%
<i>Matching on structure of classifiers</i>			
Number of true positives	253	1	47
Number of false positives	0	5	85
Number of false negatives	0	363	58
Match ratio	100%	99.8%	99.4%
<i>Global matching on metric profiles</i>			
Number of true positives	253	294	101
Number of false positives	523	107,685	7,701
Number of false negatives	0	70	4
Match ratio	99.9%	60.8%	40.1%
<i>Local matching on metric profiles</i>			
Number of true positives	253	294	82
Number of false positives	122	107,685	1,442
Number of false negatives	0	70	23
Match ratio	99.9%	60.8%	88.6%

Table 7.2: The reliability of the matching strategies

profile approach gives the poorest results. In this case package information was available for both artifacts. This clearly improves the result of matching with metric profiles.

## 7.3 Case assessment approach

This section describes the analysis approach we performed to assess the correspondence. Most analyses were performed on site. This means that a careful preparation is important. On site it is usually not possible to quickly download a tool and get it to work because the regular tools can't do the job. All on site analyses were performed using a prepared laptop.

### 7.3.1 Preparation

Prior to each analysis a short experiment script was written. This script consisted of two parts:

- The tools that should be available on the demo laptop;
- The steps to be taken in the assessment approach

## Tools

For correspondence assessment we obviously use the correspondence toolset. These tools are described in more detail in chapter 6.

Chapter 2 also lists some tools for metric calculation and XMI extraction. The tools we use are:

- SDMetrics;
- Columbus/CAN;
- Enterprise Architect;
- Rational Rose;

But we also took some general purpose tools. Microsoft Office could be used to create a little presentation of the results on site. To be able to quickly manipulate the data in the database, we used MySQL control center and Borland Delphi. The first allows us to execute queries easily. The second is a programming tool that allows us to perform all kinds of manipulation on the database quickly. In case B for instance, all implementation classes were preceded by some constant string. It is impossible to remove these prefixes using SQL but with Delphi the job could be done quickly.

## Assessment approach

The script also listed the main steps of the correspondence assessment. The steps to be taken were divided into three categories. Table 7.3 lists these categories and their expected output.

Category	Output
UML Model Fact extraction	Design XMI file and class level metrics
Source code model fact extraction	Implementation XMI file and class level metrics
Analysis	Matching and classification of differences

Table 7.3: The categories of an experiment script

An example of an experiment script can be found in appendix A.

### 7.3.2 UML Model extraction

All CASE tools we encountered had an option to save the model in XMI format. Unfortunately each tool wrote its own ‘dialect’ of the standard. The analyses tools can not handle all of these dialects. The output of some CASE tools thus had to be converted into a proper format.

Especially Visio models are difficult to get in the proper format. Microsoft has a Visio plug-in that can export information in XMI format. It is rather limited since it only exports class definitions. Diagrams are ignored. For correspondence checking this is not a big problem. However, the tool also does not export association and dependency relations which is a bigger problem. Luckily only case C was designed in Visio. This case is pretty small so the association problem was solved by redrawing the diagrams in another tool.

The conversion of other dialects was done using Enterprise Architect. The XMI produced by the CASE tool was imported by Enterprise Architect and immediately exported again in the proper format. The design metrics were all calculated using SDMetrics.

### 7.3.3 Source code model extraction

With respect to source code, especially C++ is a hard language to handle. Especially the extended macro mechanism regularly gets fact extractors into trouble. In four cases C++ was used for the implementation. Most of these cases could be handled by Columbus/CAN. The preprocessing had to be done in advance. If the implementation was organized in a source tree, Columbus/CAN had difficulties finding include files. We wrote a small program that generates a batch file for preprocessing. All source files could be preprocessed beforehand in this way.

Columbus/CAN did have big problems with case C because of the dependencies on some library. Luckily Enterprise Architect can also deal with C++ code. Enterprise architect restricts itself to the class declarations in the header files. Therefore it misses some facts like dependencies between classes.

Case B was implemented in C which is not an object oriented language. The implementor of this system had a tool to convert this code into a UML model. Our input in this case was thus a ready to use XMI model of the implementation.

Case A was the easiest to handle. Enterprise Architect had no difficulties at all with the C# code of this case.

## 7.4 Matching strategies

In chapter 4 we discussed methods to match design entities to implementation entities. Table 7.4 lists these methods. Each method is referred to in this section by a code. The codes are also listed in this table.

Code	Matching approach
H	Manual improvement of the matching
N	Matching based on classifier names
S	Matching based on structure

Table 7.4: The matching approaches and corresponding results

Table 7.5 lists the details about the matching of design entities to implementation entities for the case. For each case one or more matching methods were used. This is indicated using the codes of table 7.4. We list for each case the *match ratio*. The match ratio is the number of classes that was matches as percentage of the total number of classes. This gives an impression of the quality of the matching.

The match ratios of the design are higher than the match ratios of the implementation in all cases. This is because the number of classifiers in the implementation is larger than the number of classifiers in the design.

Especially case G was matched really good. Case G was the case where two versions of a design were compared. It was impossible to improve this matching manually. The parts that were not matched were not intended to match.

<i>Case</i>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>	<b>G</b>
Matching approach	N	N	N	N	N	N, S, H	N
<i>Design</i>							
Match ratio	100%	58%	100%	91%	83%	77%	98%
<i>Implementation</i>							
Match ratio	80%	50%	35%	36%	34%	34%	96%
<i>Correspondence</i>							
Correspondence measure	57%	25%	38%	43%	NA	37%	79%

Table 7.5: The matching approaches and corresponding results

Additional to the quality of the matching we also calculated the correspondence value of the systems. Unfortunately, the calculator was not available when we assessed case E. Therefore E has no result here.

When calculating correspondence, weights have to be assigned. We used the following weight assignment in the assessment:

$$\begin{aligned}
 \text{sim}(C_D, C_I) = & \frac{1}{4} \cdot \text{sim}(\text{name}(C_D), \text{name}(C_I)) + \\
 & \frac{1}{4} \cdot \text{sim}(\text{operations}(C_D), \text{operations}(C_I)) \\
 & \frac{1}{4} \cdot \text{sim}(\text{attributes}(C_D), \text{attributes}(C_I)) \\
 & \frac{1}{4} \cdot \text{sim}(\text{relations}(C_D), \text{relations}(C_I))
 \end{aligned} \tag{7.1}$$

Table 7.5 shows the correspondence values for the various systems. Correspondence is a value between 0 and 1. Therefore the results are given as percentages. This measure does not take the cause of deviations into account. It merely measures what is different.

The correspondence value is not the most important output of the tool. Developers are generally not interested in this number. They rather see what deviations occur and what their impact is.

## 7.5 Scatter plot analysis

Once a matching was established we could analyze the the metric correlations to find outliers. Especially for case G we expect the many correlating pairs of metrics.

For case G we had the same set of metrics for both design and “implementation”. We could thus draw scatter plots of each metric in both versions of the system. We expect these to show high correlation.

Figure 7.1 shows 5 scatter plots. The 5 scatter plots depict different aspects such as inheritance, operations, attributes and association relations.

In figure 7.1 we selected a class that is far outside the confidence interval with respect to the number of operations (NumOps). This class is the square data point in the scatter plot. Apparently many new operations were introduced in the new version. Looking at the number of attributes plot (NumAttr), the class also lost an attribute. In the other plots, the class is on the regression line. There are thus only changes with respect to the operations and attribute sets.

In figure 7.2 we selected a class that was an outlier with respect to the depth of inheritance tree (DIT) metric. From this scatter plot we conclude that the inheritance tree was changed

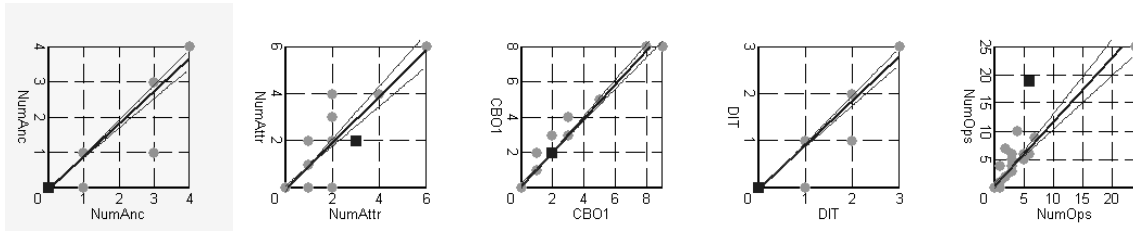


Figure 7.1: Scatter plots showing measures for different aspects

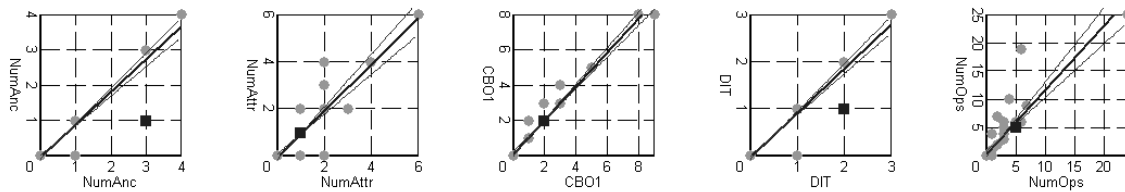


Figure 7.2: The outlier selected seems to have changes only in the inheritance tree

above this class. It is not surprising that the class is also an outlier in the number of ancestor (NumAnc) diagram. However, with respect to operations, attributes and associations nothing changes however.

The DIT diagram shows one other outlier. This is also the case for the NumAnc diagram. We expect these to be the same as well. And indeed they are. For both outliers in the DIT diagrams, the depth decreases. We checked whether the two outliers were related. It appeared to be different parts of the system.

From the pictures shown, we immediately get about five different classes that fall outside the confidence interval in different plots. These classes are subject to a detailed analysis. All cases are confidential, hence the results of the detailed analysis will not be discussed.

## 7.6 Classification of deviations

We selected a number of suspicious classes for each of the cases. These suspects were analyzed in detail. In this section we present what kinds of deviations we found in the case studies we analyzed. For each deviation we give a risk. This risk expresses the impact on system understandability if differences of this kind are present.

**Introduction of methods and attributes** In most cases new attributes and methods were introduced in the implementation. Most of these were declared private or protected. These methods are typical local helper methods to smoothen the implementation a little.

Only in few cases, public methods are introduced in the implementation. It is worse to introduce undocumented public methods than private methods. Public methods can be used by other classes. Ultimately this could cause new relations between classes that were not designed.

The risk for the introduction of private methods is low since their effect is local. The risk for the introduction of public methods is high if these methods are actually used by other parts of the system.

**Unimplemented methods** The design of case B defines some methods that are not used in the implementation. If in the design nothing is done with this method, this is not a big problem. In such a case it is just a minor design flaw.

If the method is used however, it becomes a bigger problem. A designed method may for instance occur in a sequence diagram. In that case it should be in the implementation as well otherwise the designed behavior can not be implemented.

**Introduction of classes** As can be seen from the case characteristics in table 7.1 in most cases the implementation consists of more classes than the design.

There are different causes for the introduction of classes. One of them is the introduction of local helper classes. In treelike data structures there is often the notion of a node. These nodes are not visible to the outside world. The risk in this case is low.

Ansi C++ does not have the notion of interfaces. It is common practice to use abstract classes in this case. As this is the best possible solution for handling interfaces it should not considered to be a problem.

Especially in case F the number of introduced classes is huge. The implementation has more than twice the number of classes as the implementation. In this case it often occurs that a base class is designed but specializations are introduced in the implementation. The huge number of introduced classes can indicate that the design was not detailed enough. The risk is thus related to the number of introduced classes.

In case A the only introduced class was a very important one. It was used by all other classes in the system. The purpose of the class was to implement some commonly used helper methods. Obviously the class is very important for other classes and thus it should be documented. In this small case it is not a big problem but if the case would be larger, it might become a problem. The risk in this case is thus high.

**Changes in the inheritance tree** We expected not to see many of these deviations. It is often risky to make changes in the inheritance tree. A single inheritance arrow can change a class dramatically both in structure and behavior. In general it can be said that any change in the designed tree causes a high risk for program understandability.

Case F showed an interesting deviation of this kind. For some part of the system, all classes inherited from a newly defined class. Near the declaration of this class we found a comment stating why this was done. This clearly was a design decision. The class should have been in the design but unfortunately it wasn't.

**Unused relations** In two cases we discovered relations in the design which were not used in the implementation. From the implementation point of view there was no reason for the relation to be there. This actually was quite a surprising result. We did expect relations to be introduced in the implementation but not the other way around.

It made us wonder why the relation is designed. We expect that this also happens if someone is trying to familiarize himself with the system. Therefore the risk is high.

**Relations not modeled** Relations that were not designed do appear in the implementation. Relations can be any relation between two classes in the design such as dependencies, associations and inheritance.

If relations between two classes in the system are missed, high risks are introduced. This typically holds if the missed relations are between different parts of the system. Typically relations drawn between classes give a first impression about how the different parts of the system cooperate.

It also occurs that in the implementation there are relations between the system and some library it depends on. In the design this is often modeled at a high level. It is not necessarily problematic to leave libraries out of scope in the design. In case C however, the relations with some library were drawn inconsequently. That is some relations were drawn whereas others are not. A reader of the model may falsely assume that the only connections with the library are the ones drawn.

## 7.7 Concluding remarks

We analyzed different systems with the methods described in this thesis. We assessed cases with different characteristics.

It turned out that names of classifiers provides the best method for matching. In each of the case studies we started by matching classifier names. For most cases the resulting matching couldn't be improved anymore.

We conducted an experiment to see how good the matching methods worked. From this experiment it also became clear that matching on names was the most reliable method for matching. In this experiment it also became clear that using metric profiles for matching was useless. Matching on structure can be done if the deviations are not too large.



## Chapter 8

# Evaluation and conclusions

In this chapter we briefly summarize the problem and our solution. Then we present our conclusions and the discussion about the results of the case studies.

### 8.1 Project overview

It occurs often that the implementation of a software system does not correspond to its design. This can be the result of evolutionary changes to the software or implementation errors. Program understandability is compromised by degradation of correspondence. The implementation of new features gets complicated, especially for programmers who never worked on the system before.

Therefore we want to compare the implementation with its design. The implementation is generally much more detailed than the design is. In order to compare the design with the implementation, we need to reverse engineer the implementation such that it comes at the same level of detail. We decided to use existing tools for this purpose.

We try to assess the correspondence by calculating a so called similarity value for the system. This value expresses the correspondence between design and implementation. For this measure to be calculated we need a matching between a software design and its implementation.

We used various strategies for matching. Some of them were proposed in existing literature. Examples of existing methods were matching based on classifier names and matching on structure. For the structure we used our own definition of similarity values. This makes the structure matching slightly different from existing work. We also tried to use software metrics for this purpose. This approach was our own contribution. If the search space is too large for a matching method, we could use the package matching heuristic. This approach assumes that classes are more likely to match if they are located in related packages.

We discovered that correlation analysis was a useful tool for finding deviations from the original design. The outliers in scatter plots helped us to find the most interesting differences in case studies.

Developers are generally not interested in the correspondence value. They are more interested in what differences are found and the impact of these differences.

## 8.2 Conclusions

We validated our techniques by analyzing industrial case studies. The primary goal of this is to validate the techniques described in this thesis. Apart from that the feedback from the companies is very valuable. These techniques are developed to improve the correspondence between design and implementation of software systems. The techniques are only interesting if they are useful in real projects. Performing case studies for industrial partners is a good method to see if the techniques are valuable in practice.

It is not only useful but also enjoyable to visit companies and run the tools on a real software project. It is quite satisfactory to see that the techniques produce output that is valuable to the developers.

From all of the matching strategies discussed, the matching based on classifier names performed best of all strategies. In all case studies we saw that names rarely changed between design and implementation.

Matching on structural aspects performed worse. This is a straight result of the deviations between the design and the implementation. Especially operations and attributes were changing frequently. Inheritance relations were a little more stable.

The metric profile approach is no help at all for matching elements in large systems. Metric profile matching only works if the metric profile is distinctive enough. There must be many different metrics in the profile. These metrics should measure different aspects. The correlation of the metrics in the profile should be high. If metrics correlate poorly, then the predicted implementation profile is unreliable. This frustrates the matching.

Matching should be checked manually. The automatic methods we use, only consider structure. The deviations in the structure frustrate the matching. Humans consider more than just structure when they are reconstructing a matching:

- Humans also check the behavior. Behavior of classes can also provide vital information for matching.
- Usually people read design documentation prior to matching. They know the system and use this to recognize the implementation of model elements from the design.
- Often domain knowledge can be used for the matching. If people encounter a car in a design they know it has wheels for instance. If they see “car” and “wheels” they automatically associate these with each other.

Code generation helps to keep correspondence. The initial framework is generated from the UML model and therefore the correspondence holds. As soon as the programmer starts the implementation, correspondence issues are still introduced easily though. In some of the case studies code generation was used. Even though code generation was used, the final version of the implementation still deviated from the design. Code generation is thus more useful to save some work in building an implementation framework.

## 8.3 Directions for further work

In this section we present some ideas for future research topics that are related to the correspondence assessment described in this thesis.

### 8.3.1 Matching clusters

In this thesis we assumed that there is a one-to-one matching between design elements and their implementation. In general, this is a false assumption. It happens quite often that a class from the design falls apart into different parts in the implementation. Clearly, a one-to-one matching doesn't suffice anymore.

The same happens if we want to compare different versions of a design. Suppose we want to assess the correspondence of an architecture design with a more detailed design. The matching will certainly not be a one-to-one matching in such a case.

There are many ways to cluster a model. It is a hard problem to find the right clustering. The deviations that may exist between the models, complicate the clustering even more.

### 8.3.2 Assessing the correspondence of behavioral aspects

In this thesis we focused on structural aspects of classes. Structural aspects are for instance operations, attributes and relations with other classes. All of these are typically visible in a class diagram.

The UML notation also provides many behavioral views such as sequence diagrams or state machines. It is also interesting to compare the designed behavior of a system with the behavior of the implementation.

Comparing the behavior of a system is much more difficult than comparing the structure. If the implementation is in an object oriented language such as C++ or Java, the representation of the structure of a class in design and implementation is quite similar. In both artifacts, a class has a name and lists of operations and attributes. The representation of behavior in the design differs a lot from the representation of behavior in the implementation.

### 8.3.3 Automatic correspondence assessment

Many software developers use version management systems such as CVS when developing a system. Each project day, the design and the implementation of a software system evolve. One of the companies we visited for a case study proposed to run a correspondence check on the repository each night. By the morning, a correspondence report would be available that identifies correspondence risks in the software system. In this way correspondence problems can be tackled in an early stage.

### 8.3.4 Correspondence between design and documentation

In one of the case studies we found a class that was marked as an outlier. In this case some new features were about to be implemented. The developer who was going to implement the new features had not worked on the original system. He read the architecture and design documents. We asked him about the outlier we found but he did not know that this class was in the system.

It turned out that the class was renamed between two versions of the design. The new class was available in both design and implementation, but the pictures in the design document were never updated. The potential risk was even bigger because this class appeared in the implementation with its old name and its new name.

This example illustrates that correspondence issues also play a role between design and documentation. Currently there are advanced CASE tools but these focus on the diagram-

ming. There are facilities to document the diagrams. These facilities are very limited however. It is not possible to create a complete design documentation based on these notes only.

On the other hand there are advanced text editors that are very suitable for writing a design or architectural document. These tools are very general text editors however. In I-Spec models [Jon00] it is already possible to include extended documentation within the model itself. The textual documentation can then be generated automatically. This approach can also be useful for UML models.

# Bibliography

- [ACPT00] Giuliano Antoniol, Bruno Caprile, Alessandra Potrich, and Paolo Tonella. Design-code traceability for object-oriented systems. *Annals of Software Engineering*, 9:35–58, 2000.
- [ACPT01] Giuliano Antoniol, Bruno Caprile, Alessandra Potrich, and Paolo Tonella. Design-code traceability recovery: selecting the basic linkage properties. *Science of Computer Programming*, 40(2-3):213–234, 2001.
- [CK94] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Transactions on Software Engineering*, volume 20, pages 476–493, 1994.
- [Cod04] The Code Project. Dynamic class factory. <http://www.codeproject.com/cpp/SimpleDynCreate.asp>, July 2004.
- [Con80] W. J. Conover. *Practical nonparametric statistics*. Wiley and Sons, New York, 2nd edition, 1980.
- [DMH01] T.R. Dean, A.J. Malton, and R. Holt. Union schemas as a basis for a C++ extractor. In *Eighth working Conference on Reverse Engineering*, 2001.
- [EDG04] Edison Design Group. C++ front end internal documentation. <http://www.edg.com>, November 2004.
- [FMB<sup>+</sup>01] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, and M. Tarkiainen. Columbus - tool for reverse engineering large object oriented software systems. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 16–27. University of Szeged, June 2001.
- [FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics, A Rigorous & Practical Approach*. PWS Publishing Company, 2nd edition, 1997.
- [Fro03] FrontEndART Ltd. Columbus/can user manual. <http://www.frontendart.com>, January 2003.
- [GHJV00] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, De Nederlandse editie*. Addison-Wesley Longman, 2000.
- [Gir02] Martin Girschick. Uml diff, erkennung und analyse von unterschieden in klassendiagrammen und sequenzdiagrammen. Master’s thesis, Technische Universität Darmstadt, 2002.

- [Gmb96] TakeFive Software GmbH. Sniff+ user's guide and reference. <http://www.takefive.com>, 1996.
- [Jon00] H. B. M. Jonkers. Ispec: Towards practical and sound interface specifications. In *IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods*, pages 116–135, London, UK, 2000. Springer-Verlag.
- [KC99] Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.
- [LB96] Sandro Morasca Lionel Briand, Khaled El Emam. On the application of measurement theory in software engineering. In *Empirical Software Engineering (Historical Archive)*, volume 1, pages 61 – 88, Jan 1996.
- [MCB05] Johan Muskens, Michel R.V. Chaudron, and Reinder J. Bril. Finding inconsistencies between views using relation partition algebra. Technical report, 2005.
- [MLC04] Johan Muskens, Christian F. J. Lange, and Michel R. V. Chaudron. Experiences in applying architecture and design models in multiview models. In *Proceedings of 30th EUROMICRO*, August 2004.
- [MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20(4):18–28, 1995.
- [MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, 2001.
- [MP80] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18 – 31, February 1980.
- [MR03] Douglas C. Montgomery and George C. Runger. *Applied Statistics and Probability for Engineers*. John Wiley and Sons, New York, 3rd edition, 2003.
- [MRH04] Douglas C. Montgomery, George C. Runger, and Norma Faris Hubele. *Engineering Statistics*. John Wiley and Sons, New York, 3rd edition, 2004.
- [Mus02] Johan Muskens. Software architecture analysis tool. Master's thesis, Technische Universiteit Eindhoven, 2002.
- [OMG02] Object Management Group. Xml metadata interchange (xmi) specification version 1.2. <http://cgi.omg.org/docs/formal/02-01-01.pdf>, January 2002.
- [PQ95] T. J. Parr and R. W. Quong. Antlr: a predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, 1995.
- [RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

- [SOMG04] Richard Soley and Object Management Group. Model driven architecture white paper. 2004.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.
- [SWM97] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: a visualization environment for reverse engineering. IEEE Computer Society, May 1997.
- [TCL02] Roseanne T. Tvedt, Patricia Costa, and Mikael Lindvall. Does the code match the design? a process for architecture evaluation. *International proceedings on software maintenance*, pages 393–401, 2002.
- [Ter05] Maurice Termeer. Metricview, visualization of software metrics on uml architectures. <http://www.win.tue.nl/metricview>, January 2005.
- [vH03] Frank van Ham. Using multilevel call matrices in large software projects. In *IEEE Info Vis 2003*, pages 227–232. IEEE Computer Society, October 2003.
- [vOLC05] Dennis J.A. van Opzeeland, Christian F.J. Lange, and Michel R.V. Chaudron. Quantitative techniques for the assessment of correspondence between uml designs and implementations. In *9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2005.
- [Wüs04] Jürgen Wüst. Sdmetrics user manual. <http://www.sdmetrics.com>, December 2004.
- [Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.



# Appendix A

## An example of an experiment script

### Experiment script

#### Case specifics

Company	TU/e
System	Scatterplotter
CASE tool	Enterprise Architect
Implementation	C++

#### Required tools

- Borland Database Engine
- Borland Delphi 7
- Design Implementation Correspondence Tools
  - Metric processor
  - XMI processor
  - Scatterplotter
  - Matcher
- Columbus/CAN
- Enterprise Architect
- MySQL Control Center
- MySQL server v4.1
- SD-Metrics

## Assessment approach

### UML Model Fact Extraction

1. **Save model as XMI**

In Enterprise Architect select the root package for the XMI export. Then choose Project, Import/Export, Export package to XMI.

2. **Run SDMetrics on the XMI file**

Open the XMI file using SDMetrics and store a comma separated value file of the data.

### Source code Model Fact Extraction

1. **Prepare reverse engineering**

- Use the PPScript tool to create a preprocessing batch file.
- Execute the batchfile.

2. **Reverse engineer source code**

- Start a new Columbus project and add the source files.
- Disable preprocessing in the settings dialog.
- Extract the information
- Save the extracted information in XMI format

3. **Calculate metrics**

- Save a metrics file from Columbus/CAN

### Source code Model Fact Extraction

1. **Clear fact database**

Run MySQL Control Center and execute the ClearDB.SQL file.

2. **Store metrics in database**

Run the MetricProcessor tool. Process both design and implementation metric files. Be careful to specify the correct source and metric type!

3. **Partial matching based on names**

Either run a Hamilton distance tool (Class\_Matcher) or if the names are very similar run an SQL query with the LIKE operator.

4. **Searching for correlations**

Run the MetricProfile scatter plotter. Specify a lower bound for the (absolute value of) the correlation coefficient and press "Find Correlations".

5. **Check exceptional dots in scatter plots**

For dots far from the regression lines, compare the UML model with the source code. (For now, difference finding is performed manually).