Quantitative Techniques for the Assessment of Correspondence between UML Designs and Implementations

Dennis J.A. van Opzeeland, Christian F.J. Lange, Michel R.V. Chaudron

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513
5600 MB Eindhoven, The Netherlands
{*c.f.j.lange,m.r.v.chaudron*}*@tue.nl*

Abstract. In this paper we discuss approaches to assess the correspondence between a software design and its implementation. We consider object oriented software systems which are designed using the UML notation. Correspondence is important for understanding the system since designs are easier to comprehend than large pieces of source code. To assess the correspondence of a system, we match entities from the design to pieces of source code. We define a matching based on classifiers. Several approaches are discussed to establish such a matching. These approaches are matching based on classifier names, matching based on metric profiles and matching based on structural properties of classifiers. Once this matching is defined, it is possible to detect and visualize the actual differences between design entities and parts of source code. The approaches have been validated through an industrial case study.

1 Introduction

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 [17] 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, which 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 a necessity 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.

The purpose of this study is to develop a method to make a match between model and implementation and to assess the degree of correspondence.

2 State of the art

2.1 Correspondence checking and reverse engineering

In order to compare a software design with its implementation, one has to abstract from implementation details. This is exactly what is done in reverse engineering. A lot of research has been performed on the re-engineering of (legacy) software systems. This resulted in many toolsets for reverse engineering. RIGI [19] is a well known and widely used example. There are also other approaches such as the Dali Workbench [9], CPPX [3] and SNiFF+ [7].

Correspondence checking is related to reverse engineering. There is an important difference however. In reverse engineering no design is available. This design is reconstructed from the source code. For correspondence checking a design *is* available which has to be compared with the implementation. The available design is sometimes called the *actual design*. The source code is used to generate a so called *as is design* of a software system [6].

Before it is possible to identify the differences between the intended and as is design, it should be determined which parts of the different designs are meant to be equal. A mapping has to be created that maps elements from the intended design to the as is design.

2.2 Defining the matching

In [6], Fiutem and Antoniol propose to trace implementation entities from design entities by looking at properties of the entities in a software system. Similarity

between a design classifier and an implementation classifier is expressed in terms of the properties they have in common. Typical examples of structural properties are:

- the name of the class
- the attributes of a class
- the operations of a class
- operation signatures
- attribute datatypes
- relations to other classes

The matching between design and implementation is defined by a maximal match algorithm [1].

Many differences may occur between design and implementation with respect to these so called structural properties of a class. It is for instance quite common that extra private or protected methods are introduced in the implementation. It is more reliable to base the matching on more information than just structural properties of classifiers.

Another approach for matching is the use of software metrics [10, 8]. This approach resembles our metric profile approach as discussed in section 4.1. However we use metrics to compare a design with an implementation instead of comparing two versions of an implementation as is done in this literature.

The comparison between design and implementation causes some difficulties with respect to metric matching. This is because of the fact that the set of available metrics in the design differs from the metrics available in the implementation. Even if a metric is available in both design and implementation, the method of measurement can be different. Therefore the value ranges can be completely different, thereby making direct comparison impossible. Take for instance a complexity metric. In the design it is possible to measure the complexity of a class in terms of the number of states in a related state machine [16]. In the implementation complexity can be measured by the lines of code metric for a class. This gives two metrics that measure the same thing but in a completely different way.

2.3 Listing the differences

Software reflexion models [14, 15] 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.

3 What is Correspondence?

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 conforms 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.

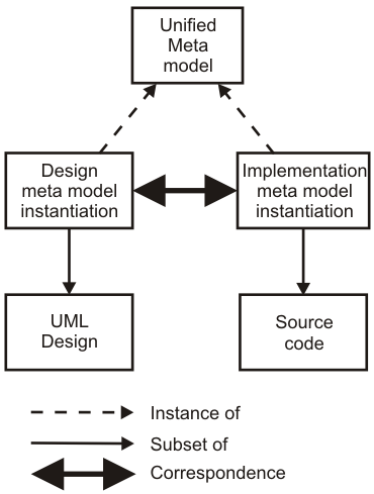

Fig. 1. The correspondence measure metamodel

For the assessment of correspondence, we use a metamodel which is inspired upon the UML meta model. For both design and implementation we instan-

tiate such a metamodel. This is depicted in figure 1. The instantiation of the design meta model is a subset of the UML model for which we want to check the correspondence. We currently do not consider all parts of the UML. State machines are for instance not considered. The same holds for the implementation meta model. It is a subset of the actual implementation where we left out all implementation details. This is shown in figure 1.

Correspondence between a design class and an implementation class can be expressed in terms of a *similarity value*. This value expresses the similarity between the two entities. There are various ways to calculate these similarity values. It is possible to express the similarity of the entities based on their names. This is done in section 4.1 but there are also other approaches.

Correspondence between a design entity d and an implementation entity i can then be defined as a weighted sum of – different – similarity values.

$$conf(d,i) = \sum_k w_k \cdot sim_k(d,i) \tag{1}$$

Since correspondence is expressed in terms of model elements, a mapping exists between design model elements and implementation model elements. Indeed the programmer of a software system always has such a mapping in mind. For large software systems it is a lot of work to make this mapping explicit however. Defining the mapping should thus be automated as much as possible.

Finding the matching automatically Automatic matching algorithms compare design elements with implementation elements and relate elements if they are similar for some aspect. An automatic mapping approach is difficult however. This is because of the differences between design and implementation.

Many different kinds of deviations from the design are possible. Introducing an extra operation or attribute in a class is already a deviation. Instantiating a class in the implementation of some operation could introduce a new dependency if it was not designed.

Not all of these deviations have the same impact on the correspondence. Clearly each deviation degrades correspondence but inserting an extra private operation has less impact than inserting a public operation. This is because private operations are only accessible for other members of the class. Private operations are typically there for implementation convenience. Public operations, on the other hand, are accessible for other classes as well. Public attributes are thus accessible to completely other parts of the system. If such a public operation is used by another part of the system, this may result in extra dependencies between different components in the system. Dependencies between parts of the system are usually very important for system understanding.

Sometimes new classes are introduced in the implementation. In languages like C++ it is possible to declare a class in the context of another class or even local in a function body [20]. Such a class is only accessible within the limited scope in which it was defined. Local or nested classes are typical examples of

implementation convenience. These kinds of classes might be absent in the design but this is not as bad as the absence of a regular class.

These examples illustrate that changes that have local effects are less influential on the correspondence than changes that (could) cause system wide effects.

3.2 Finding the differences

As soon as the matching between classifiers is defined, it is relatively easy to discover the actual deviations. One simply compares all properties of the design classifiers with the properties of its matching implementation classifier and lists the differences.

These aspects could be structural but it is also possible to consider behavioral aspects. This is a little tricky however. The behavior of a software system is almost never completely specified in terms of UML. Parts of the behavior can be specified by using state machines or activity diagrams [18]. These diagrams are typically high level and therefore difficult to relate with the sources of the software system.

For collaboration diagrams and sequence diagrams it is possible to perform some kind of checking. It should be possible to find a code path that mimics the behavior described in the diagram.

4 Our Approach(es)

4.1 Description of Approaches

Our goal is to find a method to match implementation classes to design classes. As stated before a distinctive property or a set of distinctive properties of classifiers is needed for this. There are various distinctive properties that can serve as a basis for matching approaches. In this section, we present some approaches that can be useful for defining the matching.

Subsequently, we will consider matching based on classifier names, matching based on metric profiles and matching using package information. Finally we will propose a combination of these strategies to get a best possible matching.

Matching based on names The names of classifiers are often quite a reliable information source for matching. It is quite common that a design entity and the corresponding implementation entity have resembling names. Coding conventions might introduce slight differences. The edit distance for strings [12] can be used for handling small differences.

A small edit distance value is not enough however. The strings `a` and `b` have an edit distance of 1 as well as the strings `UpdateManager` and `CUpdateManager`. The latter pair is more trustworthy for matching however. A small edit distance is more trustworthy if the compared strings are larger.

These observations lead to an expression for similarity based on class names. This expression is presented in equation 2.

$$sim(N_D, N_I) = \frac{|N_D| \uparrow |N_I| - d_{\text{edit}}(N_D, N_I)}{|N_D| \uparrow |N_I|} \tag{2}$$

The metric profile For each classifier in a design or implementation a number of metrics can be calculated. A combination $(m_1, m_2, \ldots, m_n)$ of metrics defines a characteristic pattern or profile of the classifier.

Metric profiles can be defined in both design and implementation. Furthermore, metrics from the design can be compared with metrics from the implementation. This need not be an equality comparison. It suffices if a metric in the design correlates with a metric in the implementation. The question now is which metrics correlate.

The level of correlation is expressed by the correlation coefficient ρ. The correlation coefficient is a real value in the range $[-1, 1]$. If $|\rho|$ is close to 1, the correlation is said to be strong. If $|\rho|$ is close to 0, there is no correlation at all.

There are different ways to calculate correlation coefficients. The straightforward way is Pearson's method [13]. This method does not work if the data set does not have a normal distribution [4]. In those cases, rank correlation coefficients like Spearman's Rho or Kendall's Tau [2] are more reliable.

Pairs of correlating metrics can be found by empirical analysis of case studies. First, the matching between design and implementation is created manually. Possibly other (automatic) approaches for matching can be used to simplify the process of making a matching.

The second step is to calculate metrics for both design and implementation. Many tools are currently available in the field to calculate metrics. For the experiment we used SDMetrics [22] for design metrics and Columbus/CAN [5] for implementation metrics. SDMetrics calculates 43 different metrics –after manual insertion of three custom metrics– and Columbus/CAN calculates 67 different metrics. This results in two datasets with metric data, one for the design and one for the implementation.

Using the metric datasets we can try to find correlations between design metrics and implementation metrics. Since there are many metrics and many classes available in the test case, we prefer to use a tool for this.

This empirical analysis results in a list of metric pairs that have strong correlation. The metrics in these pairs form the metric profiles for design and implementation classes. For each metric pair a regression line can be determined. Given a regression line it is possible to predict the value of the implementation metric given a design metric value. An implementation class fits to a design class if the predicted implementation values are closest to the real values. Since a high correlation between design and implementation metric indicates better predictability, it is a good idea to use the correlation coefficients as weights.

Let D be the set of design classes, I the set of implementation classes and M the set of metric pairs. For all $m \in M$, the correlation coefficient ρ_m and the

regression coefficients ($\beta_{0,m}$ and $\beta_{1,m}$) are known. Then the metric profiles of $d \in D$ and $i \in I$ can be compared as follows:

$$sim(d,i) = \sum_{m \in M} \rho_m |\beta_{0,m} + \beta_{1,m} d_m - i_m| \tag{3}$$

If $sim(d,i)$ is close to 0, the match is good. If $sim(d,i)$ grows larger, the match becomes worse. The best matching design class for some implementation class is the one that minimizes $sim(d,i)$.

$$d \simeq i \equiv sim(d,i) = \downarrow_{d' \in D} sim(d',i) \tag{4}$$

Package information In UML designs, the classes of larger systems are often organized in packages. This information is stored in the development view of the 4+1 View Model [11]. The source code of these systems is modularized in different files which in turn are ordered in a directory structure.

If a relation is found between a design package and a source directory, then it is likely that all implementation classes in that directory match to design classes in the related package.

Unlike the other approaches, package information can not establish a matching of individual classes from design and implementation. It can cluster classes in groups however thereby narrowing the search for other approaches.

A combination of strategies As will be discussed in section 6, none of the approaches can establish a matching alone. A combination of the methods might improve the results. One combination is shown in figure 2.

From the case studies we analyzed, it became clear that the class names approach is already very reliable. This will also be the case for the case study presented in section 5. For this case, over 87 % of the design classes is correctly matched by just considering classifier names. As this approach is quite stable for other cases as well, it is a good idea to derive an initial –partial– matching using this approach.

This partial relation can be used to link design packages to source directories and files. Thereby, the search space for other methods can be limited since a design class and an implementation class are only likely to be related if they are part of related packages.

Finally, the other approaches can be used on the unmatched classes in the package clusters to improve the matching further. If still some classes remain unmatched there is no alternative but to require human intervention for the matching.

For the matching based on the metric profile, one needs correlating metrics. Of course it is possible to define a generic metric profile that holds for any project. The correlations between metrics are likely to be significantly different from project to project. It is a better idea to make the metric profile project specific. In that case, the partial matching that is already available can be used

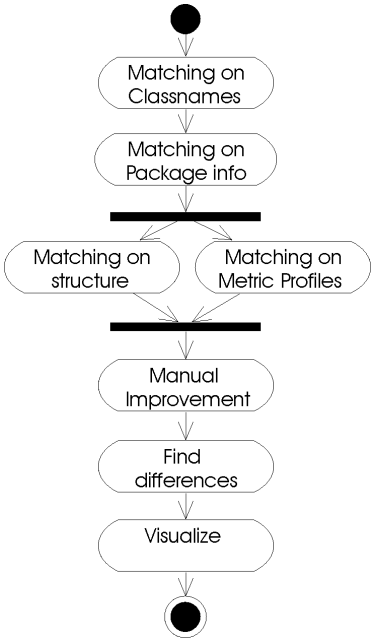

Fig. 2. An approach for matching

to calculate correlations. The selection of metrics for the metric profiles of classes thereby also becomes project specific. The custom made metric profiles can then be used for matching other classes in the system.

For this project specific selection of metric profiles to work, the calculated correlations should be significant. The significance of correlations depends on the size of the underlying dataset. In this case, the dataset is consists of pairs of classes that are already matched. If the dataset is small, sufficient significance of the correlation coefficients might be a problem.

Anything that can't be matched by hand, can be regarded as a difference between design and implementation.

4.2 Tooling

This section describes the tools that are used for the correspondencechecking approach.

MetricView The matching algorithms will be implemented as a component in MetricView [21]. MetricView originally was a tool for visualizing metric data in a UML model. An impression of the tool is shown in figure 3.

The main window shows one or more UML diagrams. For correspondence checking purposes it is convenient to show a design model behind an implemen-

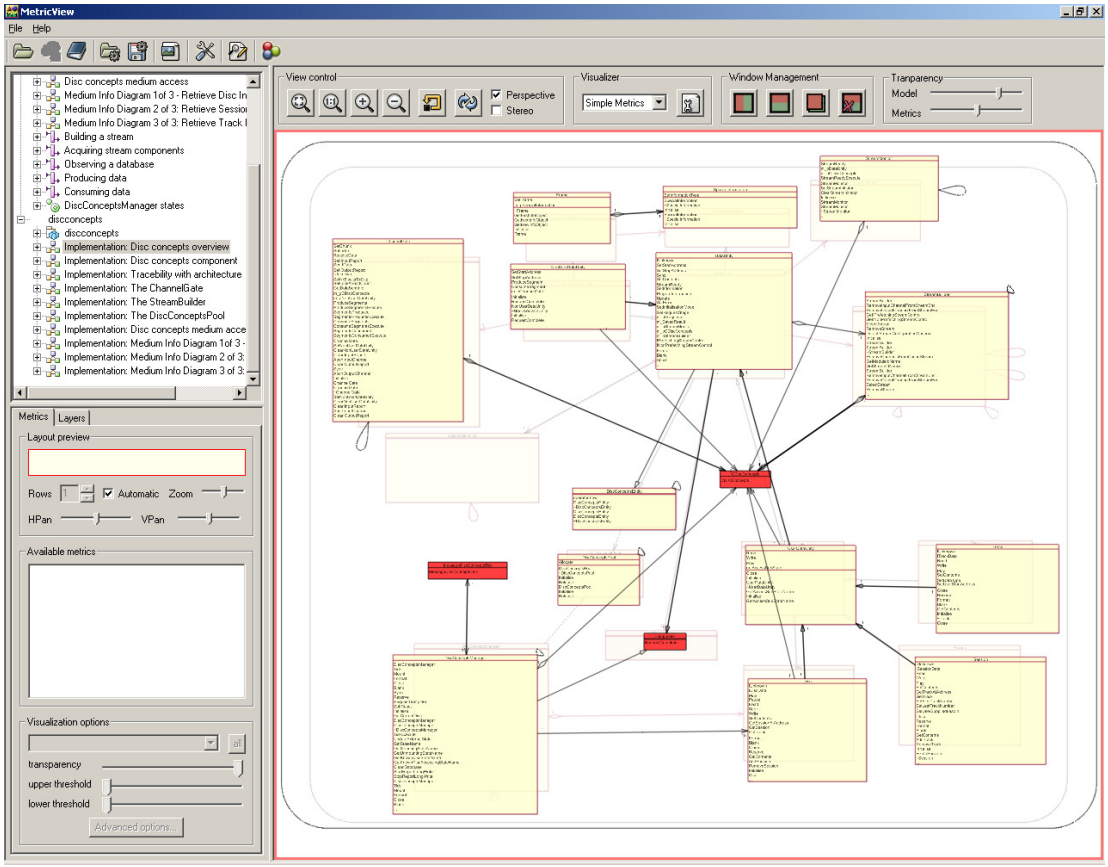

Fig. 3. The MetricView tool highlights differences between design and implementation

tation model. An implementation class is drawn on top of the matching design class. Classes that do not match are highlighted.

Recently some new features were added. Correspondence checking is one of them. For correspondence checking the following changes have been made to the original version of MetricView:

- It is possible to load both a design and an implementation model at the same time. This induces the introduction of some kind of project management.
- It is possible to load metrics in different formats than SAAT [16] output (Columbus/CAN [5], SDMetrics [22]).
- GUI support for matching properties was introduced.

5 Case Study

In this section we describe the results of our approach that resulted from an industrial case study.

5.1 Description

Using the matching criteria described in section 4, we analyzed a software system created in an industrial environment. This system is the firmware for a DVD recorder. Both UML designs and an implementation were provided. The design consists of 346 classes and the implementation has 777 classes.

5.2 Results

Matching based on names The matching based on classifier names works pretty fine for the design. Over 87 % of the classifiers is matched based just on the name of the classifier. The metric profile approach was used to find suspicious cases in this matching. It is for instance suspicious if the number of operations in the implementation is much larger than the number of operations in the design. 15 couples of a design metric and an implementation metric were compared. This resulted in 39 distinct suspicious cases which were validated manually. No false positives were found.

Matching based on metric profiles The statistical analysis of the metrics resulted in a list of metric pairs that showed a high correlation in the case study. Since it is unclear which calculation of the correlation coefficient gives the most reliable results, we calculated the Pearson, Spearman and Kendall correlation coefficients for this case. The highest correlations are discussed in this section.

Table 1 shows the metric pairs that have the highest Pearson correlation coefficients. The meanings of the metric names are listed in appendix A.

The high Spearman correlation coefficients are listed in table 2. Table 3 lists the high correlation coefficients for Kendall's Tau correlation coefficient. The significance values are below 0.001 for all correlations in the table.

What stands out immediately is that only a few different metrics from design and implementation have high correlations. There are 43 different design metrics but only 7 of them correlate strongly with implementation metrics. The same holds for the implementation metrics. Only 7 of the 67 metrics correlate strong enough to some design metrics.

One would expect that metrics that are available in both design and implementation correlate strongly. This is indeed the case for OpsInh vs. NMI (which both count the number of inherited methods). A very high correlation coefficient of 0,9242 is found. But the number of ancestors design metric seems to correlate better with the Depth of inheritance tree metric ($\rho = 0,7851$) in the implementation than with the corresponding number of ancestors metric ($\rho = 0,6759$).

Design metric	**Impl. metric**	ρ
OpsInh	NMI	0,924
OpsInh	NM	0,903
OpsInh	NAM	0,902
OpsInh	ProM	0,889
NumAssEl_ssc	DAC1	0,866
NumAssEl_sb	DAC1	0,864
Fan-out	DAC1	0,857
CBO1	DAC1	0,853
OpsInh	PubA	0,851
Fan-in	DAC1	0,846
OpsInh	NAI	0,842
DIT	DIT	0,833
OpsInh	DIT	0,829
NumAssEl_ssc	OCAIC	0,829
NumAssEl_sb	OCAIC	0,827
CBO1	OCAIC	0,821
Fan-out	OCAIC	0,821
OpsInh	ProA	0,819
Fan-in	OCAIC	0,817
Fan-out	DAC	0,817
CBO1	DAC	0,816
NumAssEl_ssc	DAC	0,813
NumAssEl_sb	DAC	0,812
Fan-in	DAC	0,803

Table 1. The table of correlating metrics using Pearson's method for correlation

Most of these metric pairs that express the same measure seem not to correlate very well.

Some strange and unexpected couples do show high correlation. Take for instance the pair (OpsInh, PubA) in table 1. OpsInh counts the number of inherited operations and PubA counts the number of public attributes. These couples are not likely to be very reliable even though the correlation coefficient is high. If we filter out the really nonsense metric pairs only 5 pairs remain with high correlation.

Using Spearman's correlation coefficient instead of Pearson's correlation coefficient does not really help. It brings some other metric pairs, but again only few distinct metrics take part in highly correlated pairs and most of the correlations are unexpected.

When we examine the Kendall correlation coefficients, the list of relatively high correlation coefficients shrinks a lot. The highest correlation coefficient is 0,754 which is pretty bad. The metrics for which this value is found, were not expected to correlate. NumAnc counts the number of ancestors whereas NAI counts the number of inherited attributes. Most of the pairs found using

Design metric	**Impl. metric**	ρ
NumAnc	NAI	0,861
DIT	NAI	0,856
OpsInh	NMI	0,830
NumInhFrom	NMI	0,829
OpsInh	NAI	0,827
NumInhFrom	NAM	0,822
NumInhFrom	DIT	0,814
DIT	NMI	0,811
NumAnc	NMI	0,810
NumInhFrom	NM	0,806
NumInhFrom	ProA	0,805
OpsInh	NM	0,803
OpsInh	NAM	0,794
NumInhFrom	NAI	0,792
DIT	DIT	0,790
NumAnc	DIT	0,789
DIT	NM	0,787
NumAnc	NM	0,786
NumAnc	NAM	0,784
OpsInh	DIT	0,783
OpsInh	ProM	0,782

Table 2. The table of correlating metrics using Spearman's method for correlation

Kendall's method are also found using Spearman's method. But Spearman's method misses two interesting matches that Kendall's method does find: NOC vs. NOC and CLD vs. CLD.

Only metric pairs that have strong correlation are good predictors for a class matching. The relatively small metric profiles suggest that the matching will not be very reliable. Indeed this is the case. The resulting matching consists of completely unrelated classes. The method only succeeds in relating a few classes that should have been matched.

Matching based on package information For the matching based on package information to be helpful, the results should consist of small clusters. Small clusters limit the search space of other approaches a lot. If the packaging in the design resembles the directory structure of the implementation, other approaches for matching can generate better results.

5.3 Differences found in the case study

Once the matching is determined, it is easy to check for differences. The metric profiles seem to be very convenient for this purpose. Metric pairs that deviate much from the regression line in a scatter plot can be considered suspicious.

Design metric	**Impl. metric**	τ
NumAnc	NAI	0,754
DIT	NAI	0,751
OpsInh	NAI	0,719
NumInhFrom	DIT	0,709
NOC	NOC	0,707
OpsInh	NMI	0,706
CLD	CLD	0,703
NumDesc	NOD	0,703
NumDesc	NOC	0,703

Table 3. The table of correlating metrics using Kendall's method for correlation

Changes in the Inheritance tree Quite surprising is the fact that the inheritance tree is changed from design to implementation. By tracing back a special case the following comment was found:

> Class Component is the abstract base class of all components in the datapath.

Apparently all other classes in the application inherit from class Component in the implementation. That looks very much like a design decision. Unfortunately this design decision is not reflected in the detailed design. This is a typical deviation from the design.

Specialization of classes It also occurs that in the implementation new classes are introduced that inherit from matched classes.

Introduction of attributes In the source code many attributes are declared in classes that were not designed. The introduction of attributes may cause the introduction of association relations between classifiers. If such relations exist anyway, especially the introduction of private attributes can considered to be implementation convenience. Protected attributes have a little more impact since these attributes are accessible to specializing classes as well.

Introduction of methods For quite a number of classes, extra methods are introduced in the implementation. A special case of this is the introduction of get- or set-methods. In the analysis, a class was found that has no setters in the design but has a *public* setter for each getter in the implementation. Public setters can be used by other parts of the system which may cause the introduction of new dependencies. In the case study this actually happened.

Unused dependencies Not all of the differences are caused by introductions in the implementation. A design class `A` depends on some other class. The source code of `A` does not show any reason for this dependency to be in the design. This gives rise to the question whether something has been forgotten in the implementation.

6 Conclusions and Future Work

In this paper we proposed a method for comparing the implementation of a software system with its design described in UML. The purpose of this comparison is to judge on the correspondence between this design and its implementation and to find possible deviations.

Design and implementation are compared via mapping onto a common metamodel. Design elements are entities like classes or interfaces and implementation elements are pieces of source code representing a class. Therefore, the comparison requires a mapping between design elements and implementation elements. We chose to define a mapping based on the classifiers in the software system.

There are various ways to compare classifiers. We proposed different approaches for matching. For our case study none of the approaches succeeded in defining a complete matching. Name matching performed best (with 87% of the design classes matched). This is because most classes that were meant to correspond had almost equal names.

The use of only the metric profile approach for matching gave a random result. The reason for this was that the number of metrics in the metric profiles was small. For the case study there were only 5 sensible strongly correlating metric pairs. Due to this, the metric profiles were not distinctive enough. The way to improve this, is to limit the search space of the metric profile approach. The package information approach can be used for this.

Even though the metric profile approach is not capable of matching large sets of entities, it is useful for the discovery of suspicious deviations. Data points in a scatter plot that are far from the regression line, are outliers. These outliers can be considered suspicious.

6.1 Future Work

The question now is what degree of correspondence is necessary. This depends on the way the design is used within the project. One purpose of a design after implementation may be to help understand the software. If differences between design and implementation do not complicate this, they are not really problematic. Inserting some private operations in the implementation is generally not a big deal. However, deviating from the inheritance tree is likely to cause problems.

If critical differences are found, they have to be resolved. There are two ways to do this. The implementation can be adapted such that it reflects the design or the other way around. Which of these alternatives is best depends on the situation.

In this situation, a challenge is to automatically update the UML model such that the design reflects the implementation. This is not difficult for differences like introduction of an operation. If new classes with relations to existing classes are introduced, drawing these classes in an existing diagram is challenging. The layout of an existing diagram should be changed such that the new classes neatly fit in it but original diagram is not changed more than necessary. In literature there exist algorithms for automatic layout a model from scratch. There are no

algorithms that modify an existing diagram into an updated version with the new elements inserted.

References

1. 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.
2. W. J. Conover. *Practical nonparametric statistics*. Wiley and Sons, New York, 2nd edition, 1980.
3. 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.
4. Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics, A Rigorous & Practical Approach*. PWS Publishing Company, 2nd edition, 1997.
5. 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.
6. R. Fiutem and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 94. IEEE Computer Society, 1998.
7. TakeFive Software GmbH. Sniff+ user's guide and reference. http://www.takefive.com, 1996.
8. Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.
9. Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.
10. K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 44, Washington, DC, USA, 1997. IEEE Computer Society.
11. Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.
12. 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.
13. Douglas C. Montgomery and George C. Runger. *Engineering Statistics*. John Wiley and Sons, New York, 3rd edition, 2003.
14. 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.
15. 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.
16. Johan Muskens. Software architecture analysis tool. Master's thesis, Technische Universiteit Eindhoven, 2002.
17. 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.

18. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
19. Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: a visualization environment for reverse engineering. IEEE Computer Society, May 1997.
20. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.
21. Maurice Termeer. Metricview, visualization of software metrics on uml architectures. http://www.win.tue.nl/metricview, January 2005.
22. Jürgen Wüst. Sdmetrics user manual. http://www.sdmetrics.com, December 2004.

A Metric names

Design metrics

Metric	Description
CLD	Class to leaf depth
DIT	Depth of inheritance tree
NOC	Number of children
NumAnc	Number of ancestors
NumDesc	Number of descendants
NumInhFrom	Sum of number of parents and number of implemented interfaces
NumProOps	Number of protected operations
OpsInh	Number of inherited operations

Implementation metrics

Metric	Description
AID	Average inheritance depth of a class
CLD	Class to leaf depth
DIT	Depth of inheritance tree
NA	Total number of attributes
NAM	Number of attributes and operations
NIA	Number of inherited attributes
NIM	Number of inherited methods
NM	Total number of operations
NOA	Number of ancestors
NOC	Number of children
NOP	Number of parents
ProA	Number of protected attributes
ProM	Number of protected operations
PubA	Number of public attributes
PubM	Number of public operations