

# **Reengineering of Design Deficiencies in Component-Based Software Architectures**

by

Marie Christin Platenius





**UNIVERSITÄT PADERBORN**

*Die Universität der Informationsgesellschaft*

Fakultät für Elektrotechnik, Informatik und Mathematik

Heinz Nixdorf Institut und Institut für Informatik

Fachgebiet Softwaretechnik

Warburger Straße 100

33098 Paderborn

# Reengineering of Design Deficiencies in Component-Based Software Architectures

Master's Thesis

Submitted to the Software Engineering Research Group

in Partial Fulfillment of the Requirements for the

Degree of

Master of Science

by

MARIE CHRISTIN PLATENIUS

Im Spiringsfelde 9

33098 Paderborn

Thesis Supervisor:

Jun.-Prof. Dr.-Ing. Steffen Becker

and

Prof. Dr. Uwe Kastens

Paderborn, October 2011



# Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

## Original Declaration Text in German:

### Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

---

City, Date

---

Signature



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem . . . . .	2
1.2	Solution Approach . . . . .	3
1.2.1	Relevance Analysis . . . . .	3
1.2.2	Reengineering Strategies . . . . .	4
1.2.3	Architecture Prognosis . . . . .	4
1.3	Limitations . . . . .	4
1.4	Overview . . . . .	5
<b>2</b>	<b>Foundations</b>	<b>7</b>
2.1	Component-Based Software Architectures . . . . .	7
2.2	Combined Reengineering Process . . . . .	7
2.3	Clustering . . . . .	9
2.3.1	Metrics . . . . .	10
2.3.2	Component Detection Strategies . . . . .	12
2.3.3	Thresholds . . . . .	14
2.3.4	Component Indicating Graph . . . . .	15
2.3.5	Limitations of the Clustering . . . . .	16
2.4	Bad Smells . . . . .	16
2.4.1	Interface Violation . . . . .	16
2.4.2	Communication via Non-Transfer-Objects . . . . .	17
2.5	Running Example . . . . .	18
<b>3</b>	<b>Reengineering Process</b>	<b>21</b>
<b>4</b>	<b>Relevance Analysis</b>	<b>25</b>
4.1	Rating Concept for Relevant Components . . . . .	26
4.1.1	Motivation . . . . .	26
4.1.2	Integration in the Reengineering Process . . . . .	27
4.1.3	Rating Strategies . . . . .	27
4.1.4	Rating Result . . . . .	30
4.2	Rating Concept for Relevant Bad Smell Occurrences . . . . .	32
4.2.1	Motivation . . . . .	32
4.2.2	Integration in the Reengineering Process . . . . .	34
4.2.3	Rating Strategies . . . . .	34
4.2.4	Rating Result . . . . .	38

<b>5</b>	<b>Reengineering Strategies</b>	<b>39</b>
<b>6</b>	<b>Architecture Prognosis</b>	<b>43</b>
6.1	Motivation . . . . .	43
6.2	Integration in the Reengineering Process . . . . .	44
6.3	Comparison Criteria . . . . .	44
6.4	Prognosis Calculation . . . . .	46
<b>7</b>	<b>Realization</b>	<b>49</b>
7.1	Overview . . . . .	49
7.2	Storage of Metric Values . . . . .	50
7.2.1	Metric Values Model . . . . .	51
7.2.2	Integration of the Metric Values Model in the Clustering Process . . . . .	52
7.3	Relevance Analysis . . . . .	53
7.3.1	User Interface . . . . .	54
7.4	Reengineering Strategies . . . . .	55
7.5	Architecture Prognosis . . . . .	55
7.5.1	Executing the Reengineering Strategy . . . . .	57
7.5.2	Starting a Clustering with SoMoX . . . . .	57
7.5.3	Calculating the Prognosis Results . . . . .	57
7.5.4	User Interface . . . . .	58
<b>8</b>	<b>Evaluation</b>	<b>61</b>
8.1	Store Example . . . . .	61
8.2	CoCoME . . . . .	63
8.2.1	Procedure . . . . .	64
8.2.2	Results . . . . .	64
8.3	Palladio FileShare . . . . .	72
8.3.1	Procedure . . . . .	72
8.3.2	Results . . . . .	73
8.4	Discussion . . . . .	75
8.4.1	Clustering . . . . .	76
8.4.2	Component Relevance Analysis . . . . .	77
8.4.3	Bad Smell Relevance Analysis . . . . .	78
8.4.4	Architecture Prognosis . . . . .	78
8.4.5	Reengineering Process . . . . .	79
<b>9</b>	<b>Related Work</b>	<b>81</b>
9.1	Bad Smell Detection . . . . .	81
9.2	Reengineering Processes . . . . .	82
9.3	Validation of the Relevance of Bad Smells . . . . .	82
9.4	Architecture Prognosis . . . . .	83



<b>10 Summary and Future Work</b>	<b>85</b>
10.1 Summary . . . . .	85
10.1.1 Discussion of the Limitations . . . . .	86
10.2 Future Work . . . . .	86
10.2.1 Future Work for the Relevance Analysis . . . . .	87
10.2.2 Future Work for the Reengineering Strategies . . . . .	88
10.2.3 Future Work for the Architecture Prognosis . . . . .	89
10.2.4 Miscellaneous Future Work . . . . .	89
 <b>Appendix</b>	
<b>A Specifications</b>	<b>91</b>
A.1 Bad Smell Specifications . . . . .	91
A.1.1 Interface Violation . . . . .	91
A.1.2 NonTOCommunication . . . . .	92
A.2 Reengineering Strategies . . . . .	94
A.2.1 Reengineering Strategies to Remove Interface Violations .	94
A.2.2 Reengineering Strategies to Remove Communication via Non- Transfer Object . . . . .	96
 <b>B Recovered Architectures</b>	<b>99</b>
B.1 Store Example, initial Clustering . . . . .	99
B.2 CoCoME, initial Clustering . . . . .	101
B.3 Palladio FileShare, initial Clustering . . . . .	103
 <b>C Eclipse Plug-Ins</b>	<b>107</b>
C.1 Required Plug-Ins . . . . .	107
C.2 Realized Plug-Ins . . . . .	108
 <b>D User Guide</b>	<b>111</b>
 <b>Bibliography</b>	<b>113</b>



# List of Figures

1.1	The reengineering life cycle (from [DDN03]) . . . . .	2
2.1	Reengineering process with combined clustering and pattern detection (adapted version from [TvDB11]) . . . . .	8
2.2	Overview on the clustering with SoMoX (from [Kro10]) . . . . .	9
2.3	The merging of classes of a component candidate into a single component via the component merge strategy (from [Kro10]) . . . . .	13
2.4	The creation of new composite components from a component candidate via the component composition strategy (from [Kro10]) . . . . .	14
2.5	A filtered component indicating graph for the threshold $t = 0.4$ . . . . .	15
2.6	The bad smell <i>Interface Violation</i> (from [vDB11]) . . . . .	17
2.7	Bad smell <i>Communication via Non-Transfer-Object</i> (from [vDB11]) . . . . .	17
2.8	Packages and classes from the example store system . . . . .	18
2.9	Recovered architecture of the example store system . . . . .	18
3.1	Reengineering process . . . . .	22
4.1	The component relevance analysis in the reengineering process . . . . .	26
4.2	Example for the calculation of the relevance values result . . . . .	31
4.3	Relevance values results for the running example . . . . .	32
4.4	Example system with one relevant and one less relevant bad smell . . . . .	33
4.5	The bad smell relevance analysis in the reengineering process . . . . .	34
4.6	Results of the bad smell relevance analysis for the running example . . . . .	38
5.1	Reengineering strategy that removes the call as activity diagram . . . . .	39
5.2	Reengineering strategy that extends the interface as activity diagram . . . . .	40
5.3	Source code example for interface violation and reengineered systems . . . . .	40
5.4	Class diagrams for the original and the reengineered system . . . . .	41
5.5	The reengineering strategy selection in the reengineering process . . . . .	41
6.1	Recovered example architectures . . . . .	43
6.2	The architecture prognosis in the reengineering process . . . . .	45
6.3	An architecture prognosis for the example system . . . . .	47
7.1	Component architecture of the developed tools and their environment . . . . .	50
7.2	Meta model for storing the metric values . . . . .	51
7.3	Simplified illustration of the method that is responsible for the recovery of components in the clustering . . . . .	52

7.4	The classes used for the relevance analysis . . . . .	53
7.5	Relevant Components View . . . . .	54
7.6	Relevant Bad Smells View . . . . .	55
7.7	Realization of the architecture prognosis . . . . .	56
7.8	The architecture prognosis view . . . . .	58
8.1	Conceptual architecture of the extended store example . . . . .	62
8.2	Detected component structure and component relevance ratings .	62
8.3	The bad smell occurrences in the store example . . . . .	63
8.4	The components from the clustering on CoCoME and their relevance	64
8.5	Detected bad smells in the selected component of CoCoME . . . .	67
8.6	Interface violation occurrences in CoCoME, rated by their relevance	67
8.7	Communication via non-transfer object occurrences in CoCoME, rated by their relevance . . . . .	68
8.8	The reengineering strategies selection page from the Architecture Prognosis Wizard . . . . .	69
8.9	The Architecture Prognosis View for the selected reengineering on CoCoME . . . . .	70
8.10	The original and a predicted components in CoCoME . . . . .	71
8.11	Discovered components in Palladio FileShare . . . . .	74
A.1	IllegalMethodAccess (InterfaceViolation) Structural Pattern . . .	92
A.2	Invalidated IllegalMethodAccess Structural Pattern . . . . .	93
A.3	NonTOCommunication Structural Pattern . . . . .	93
A.4	Reengineering Strategy 1 for Interface Violation: remove call . . .	94
A.5	Reengineering Strategy 2 for Interface Violation: add method dec- laration to interface . . . . .	95
A.6	Reengineering Strategy to Remove Invalidated Interface Violation Occurrences . . . . .	97
A.7	Reengineering Strategy to Remove Non-Transfer Object Commu- nication occurrences . . . . .	98
B.1	Components recovered from the extended Store Example . . . . .	101
B.2	Results of the initial Clustering in CoCoME . . . . .	101
B.3	Recovered Components in Palladio FileShare . . . . .	104

# List of Tables

8.1	Configuration used for the clustering on CoCoME . . . . .	65
8.2	The components detected in CoCoME, the detected bad smells per component and relevance ratings . . . . .	66
8.3	Configuration used for the Clustering on Palladio FileShare . . . .	73
8.4	The components detected in Palladio FileShare, the detected interface violations and relevance ratings . . . . .	75
8.5	Metric values with and without interface violations . . . . .	76
B.1	Configuration used for the Clustering on the Store System . . . .	100



# 1 Introduction

Nowadays, most software engineers have to work with large software systems. One possibility to cope with the complexity a software system is to follow the design principles of a *component-based software architecture*. As Szyperski et al. point out, a clear architecture is “the pivotal basis of any large-scale software technology” [SGM02]. In component-based software architectures, software systems are composed of reusable modules, the so-called software components. This approach supports the maintenance of the system by providing an overview of the system’s components. In the last years, component-based software architectures received much interest because the concept allows to reuse components in other systems and thereby reduce development costs and increase the quality of a system [SGM02].

But many software systems are adapted and extended over a long period of time. During this time, the software inevitably ages [Par94]. In particular, if changes are done by developers that are not familiar with the original system and its concepts, the conceptual architecture of the system can be (unintentionally) modified. The design erodes [VGB02] and with every modification, the risk for introducing design deficiencies like anti patterns [BMMM98] or bad smells [Fow99] increases.

Design deficiencies have a serious impact on a system’s design and by this, they decrease a software system’s quality. Removing these deficiencies naturally improves the system’s quality. The removal of design deficiencies can be accomplished by *reengineering*. Software reengineering aims at improving an existing software system so that it can “continue to be used and adapted at an acceptable cost” [DDN03]. Thus, reengineering not only means to remove design deficiencies but also to restructure a system to fix problems or to prepare it for further development and extension.

As Figure 1.1 illustrates, the reengineering life cycle consists of a *reverse engineering* phase and a *forward engineering* phase [DDN03]. At first, the design of the system to be reengineered has to be reconstructed from the system’s source code. This process is called reverse engineering. The design is then modified according to the adapted requirements. A subsequent forward engineering phase results in the modified or recreated source code of the system.

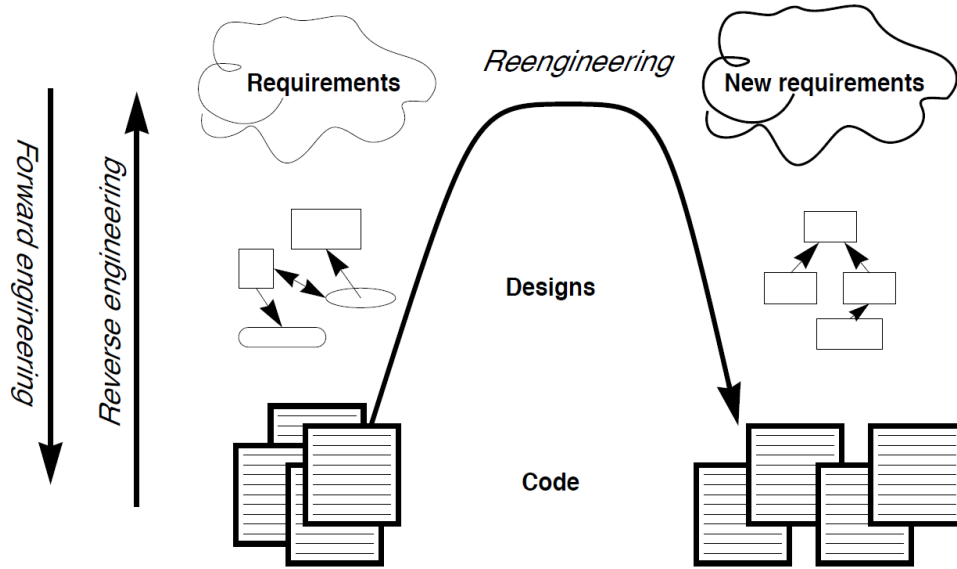


Figure 1.1: The reengineering life cycle (from [DDN03])

## 1.1 Problem

As mentioned above, the reengineering of a software system includes several tasks and methodologies. This thesis focuses on the analysis and removal of design deficiencies as part of a reengineer's challenges. Removing design deficiencies in a large system is a complicated, error-prone and time-consuming task and it confronts a reengineer with several problems.

First, she has to identify the design deficiencies in the system. There are already several approaches to detect design deficiencies in a software system. One of them has been explored by Travkin [Tra11].

The search for design deficiencies in a large system can be time-consuming, even with the help of tools that automatically detect candidates for design deficiencies. The time required to search for design deficiencies in the whole system increases with the size and the complexity of a system. To get results for a system earlier, the search scope has to be narrowed down. A solution for this problem is to focus only on a part of the system. Here, the reengineer is confronted with the next problem: She has to identify which part of the system is a good starting point for the detection of design deficiencies.

A design deficiencies detection run can result in a high amount of discovered design deficiency candidates. But not all of those detected candidates are actually problematic and depending on the context in which a design deficiency occurs, some may be more critical than others [vDB11]. Hence, the reengineer has to decide, which of those occurrences are relevant and should be removed to improve the software system's architecture.

At this point, the next problem occurs: how to accomplish the removal of a



design deficiency? Often several possibilities exist for this. Thus, the reengineer has to identify appropriate ways to accomplish the removal. This can be done by consulting design experts or adequate literature, if necessary.

Obviously, the system's structure is modified by removing a design deficiency. In order to decide how to remove a design deficiency, the consequences on the system structure have to be taken into account. It has to be ensured, that no parts of the system are damaged unintentionally and that no new design deficiencies are introduced. This is a problem because in general the reengineer has no overview of the consequences of the reengineering on the system's architecture and cannot directly see, which parts of the system change.

The next step, the actual removal of the design deficiency, is an error-prone task if it is done manually because there is a risk to change parts of the system's structure and behavior unintentionally. Because of this, the removal of selected design deficiencies should be automated.

## 1.2 Solution Approach

The proposed solution for the described problems is an automatic *relevance analysis* with a subsequent *architecture prognosis*. The relevance analysis simplifies the reengineer's decision on which part of the system the search for design deficiencies should be started and it suggests which of the detected deficiencies should be removed. The architecture prognosis simplifies the decision for a *reengineering strategy* that accomplishes the removal of a design deficiency by predicting its consequences on the system's architecture. These additional steps have to be integrated into a reasonable reengineering process.

### 1.2.1 Relevance Analysis

The relevance analysis consists of two steps. In the first step, the different parts of the system are analyzed to identify a part where the search for design deficiencies could be worthwhile. For this, the characteristics of different system parts have to be analyzed and compared.

In the second relevance analysis step, the design deficiencies in the selected part and their impact on the software architecture of the system are analyzed.

To determine a design deficiency's relevance, the relevance analysis tries to answer the following two questions for each candidate: 1. Is the candidate a critical design deficiency that has to be removed or is it acceptable and can be tolerated? 2. Would the removal of the candidate have an impact on the architecture or will the design remain unchanged? Design deficiencies whose removal seems to modify the system's design are particularly relevant because they were probably introduced unintentionally and distort the originally intended architecture.

### 1.2.2 Reengineering Strategies

In most cases there are different possibilities to accomplish the removal of a design deficiency. These different reengineering strategies have different consequences for the system. If the reengineer wants to remove a design deficiency, she has to decide which of the available strategies fits her requirements best. These different strategies to handle it have to be presented to the reengineer.

In order to allow an automated removal of the different design deficiencies, the reengineering strategies have to be specified formally. For this, literature like e.g. [Fow99] can be consulted.

There might also be situations in which the removal of a design deficiency cannot be accomplished automatically. In these cases, the reengineer has to remove it partly or completely on her own.

### 1.2.3 Architecture Prognosis

The application of the reengineering strategies to remove a design deficiency can have different consequences on the system's architecture. But typically the reengineer has no overview of those consequences and does not know how the architecture changes. In order to discover the consequences of a strategy, the design changes have to be predicted and have then to be presented to the reengineer. This process does not change the original system because it presents only a forecast of a possibility for modified version of the system.

With this prediction, the reengineer is able to select the best strategy to accomplish the removal of a design deficiency in a specific system because she has an overview of the resulting consequences. The risk of accidental changes is minimized due to clearly showing the reengineer which architectural consequences follow from the selected strategy.

## 1.3 Limitations

The analyses presented in this thesis focus on software written in an object-oriented language. In addition they only deal with component-based software architectures, as defined in Chapter 2.

According to this, only design deficiencies at the architectural level are investigated and not, e.g., code bad smells [Fow99].

There is a great variety of different design deficiencies and many possibilities to accomplish their removal. This thesis illustrates its concept exemplarily on occurrences of the bad smell *Interface Violation* [vDB11]. Furthermore, the bad smell *Communication via Non-Transfer-Objects* [vDB11] is taken as a second example.

The concepts presented in this thesis are based on heuristics. They are intended to support the reengineer, but the final decisions are intentionally left to the human. Because of this, the process proposed in this thesis is designed as a semi-automatic process.

Chapter 3 gives further information on the steps that are realized within the scope of this thesis. More conceptual ideas that could not be realized within the scope of this thesis are presented in Chapter 10.

## 1.4 Overview

The remainder of this thesis is organized as follows. Chapter 2 introduces the foundations for this thesis.

The Chapters 3 to 6 deal with the conceptual part of this thesis. Chapter 3 provides an overview of the proposed process. Chapter 4 details on the concept of the relevance analysis, while the reengineering strategies are illustrated in Chapter 5. The conceptual chapters end with a detailed description of the architecture prognosis in Chapter 6.

Chapter 7 deals with the realization of the concept, whereas Chapter 8 addresses the evaluation. In Chapter 9 related work is presented and discussed. Chapter 10 summarizes this thesis and draws conclusions. It also presents ideas for future work.



## 2 Foundations

This chapter introduces the foundations that will be required throughout the following chapters of this thesis. These include component-based software architectures and the proposed reengineering process. Furthermore it gives an overview of the clustering-based reverse engineering and presents different bad smells.

### 2.1 Component-Based Software Architectures

In component-based software architectures, software systems are composed of reusable modules, the so-called *software components*. The main idea is to create the opportunity to reuse components in other systems and thereby reduce the development costs and increase the quality of the system [SGM02]. Component-based software development has become a widely accepted software development approach because of its cost-effectiveness [KP05].

Components can consist of classes (so-called *basic components*) or can be composed from other components (*composite components*).

The communication between components is done via interfaces and connectors. *Transfer objects* serve as data containers for the messages between components. The caller component “fills” a transfer object with data and passes it to the called component that needs that data [vDB11]. Transfer objects are not part of the system architecture.

Architecture models based on components and their relations between each other describe a software system on an abstract level and thereby provide a good overview to the software engineer. For this reason, they facilitate the task of maintaining or extending an existing software system.

But architecture models (if existing) are often incomplete or out-dated so that the understanding of an unknown system is a tedious task and extending it becomes complicated. Because of this, it is useful to recover architecture models for existing software systems. The recovery of a component-based software architecture can be done by clustering-based reverse engineering approaches.

### 2.2 Combined Reengineering Process

*Reverse Engineering* is the task of analyzing software systems in order to understand a system and recover its design documentation. There are two main types of reverse engineering approaches: *clustering-based reverse engineering* and

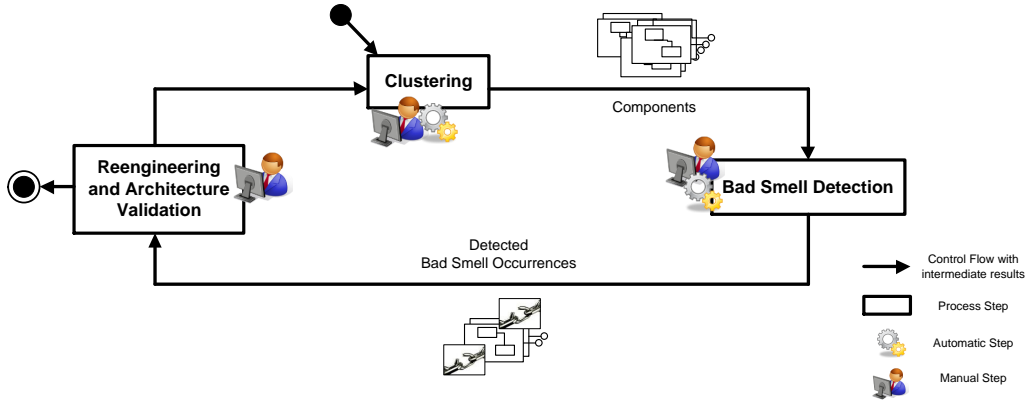


Figure 2.1: Reengineering process with combined clustering and pattern detection (adapted version from [TvDB11])

*pattern-based reverse engineering.* Clustering-based approaches group a system's elements into components in order to provide an overview of the system [DP09]. Pattern-based approaches try to detect specified patterns which simplifies the understanding of the original developer's design intentions [KSRP99].

Both approaches have their drawbacks. One major problem with the clustering-based reengineering is that it can only recover the structure of the components but not their purpose. In contrast, pattern-based approaches suffer from a long run-time needed to analyze a system and often result in an unpractically large set of detected pattern implementations [TvDB11]. To control these disadvantages, a combination of both approaches seems promising, as illustrated in [Tra11].

To combine clustering-based and pattern-based reverse engineering approaches, an iterative process is suitable. Figure 2.1 depicts the process as proposed in [TvDB11].

The process starts with the source code of the software system to be analyzed. The source code is the input for the clustering analysis, which groups the system into components and thereby recovers an initial architecture. One tool that does a clustering analysis is SoMoX [BHT<sup>+</sup>10, Kro10]. The clustering with SoMoX is described in detail in Section 2.3.

In the next step, a pattern detection recovers bad smell occurrences for each of the architecture's components. The Reclipse Tool Suite [vDMT10a, vDMT10b] provides tool support for automatic pattern detection.

Finally, it has to be decided how to handle the detected bad smell occurrences. This means, it has to be determined which bad smells should be removed and how to accomplish this. Currently, this step, including the corresponding reengineering of the architecture, has to be done manually by the reengineer.

Here, it should be emphasized that reengineering is not equatable with a refactoring, since refactoring is defined to change a systems internal structure without

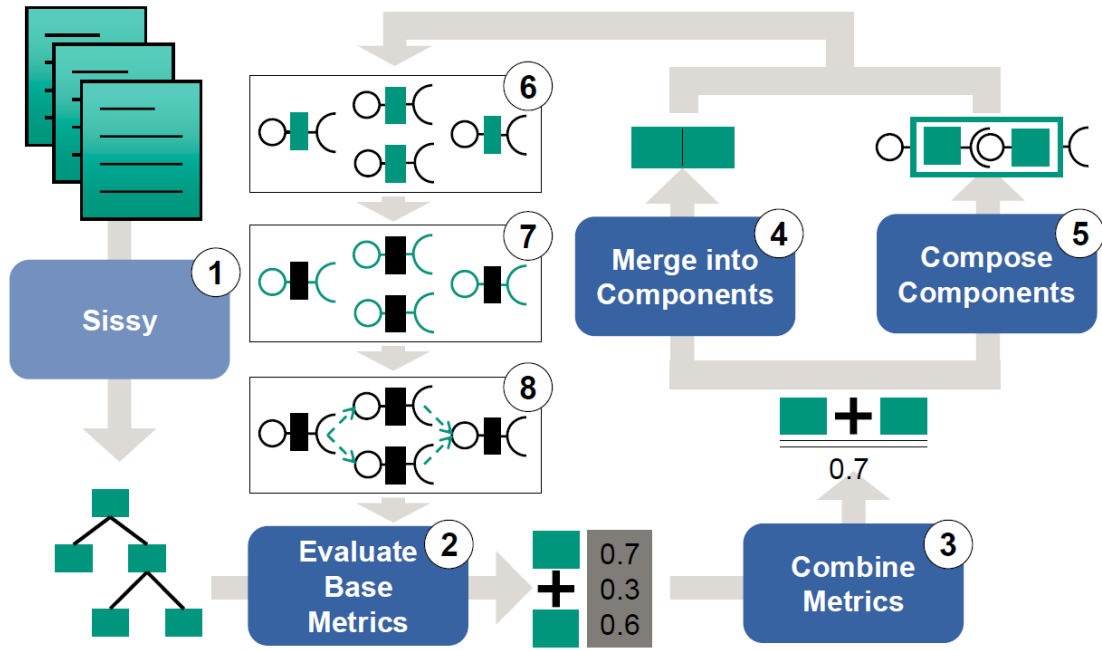


Figure 2.2: Overview on the clustering with SoMoX (from [Kro10])

changing its external behavior [Fow99]. In opposition to this, the reengineering strategies covered in this thesis do not always preserve the behavior.

After the reengineering, the system’s architecture may have changed. To evaluate the consequences of the modifications and to further improve the architecture by detecting new deficiencies, the process can be reiterated.

The presented approach is used as a foundation of this thesis. It is realized by using SoMoX for the clustering step and Reclipse for the pattern detection, as Travkin describes in his thesis [Tra11].

## 2.3 Clustering

This section details on the clustering-based reverse engineering process with SoMoX. It is mostly based on Krogmann’s thesis [Kro10].

Clustering-based reverse engineering approaches aim at the reconstruction of a software system’s architecture. For this purpose, the system’s elements are structured into different components. Because this thesis focuses on object-oriented systems, the structured elements are classes in this case.

The clustering process used in SoMoX is illustrated in Figure 2.2. To execute a clustering, the source code of the system has to be analyzed. SoMoX uses a generalized abstract syntax tree (*GAST*) of the source code to analyze a system. This *GAST* is a language-independent representation of object-oriented source code. For creating the *GAST* from the source code, the parser *SISSy* [SSM06] is used (1).

After the parsing step, SoMoX starts with the actual clustering steps (2-8). The clustering is an iterative process, in which each iteration aims at a higher abstraction level of components [BHT<sup>+</sup>10]. Each iteration builds on the results of the previous iterations and creates an architecture model which describes the components detected until that iteration. When starting the process, the initial components are formed from single classes and interfaces. The process ends, if no further component abstractions are found.

For the clustering, first a number of code metrics is evaluated on the GAST representation (2). Metric values are evaluated for so-called *component candidates*. A component candidate is a tuple of two sets of classes. Sets of component candidates later result in components. The metrics are described in detail in Section 2.3.1.

There are two steps that decide if a component candidate is converted into a component: the *merging* step and the *composition* step. SoMoX determines which classes are merged into which components and which components are composed to composite components with the help of a combination of these metric values (3).

In the merging step (4), the classes of component candidates are merged together into one component. In the composition step (5), composite components are composed from component candidates. The decision if component candidates are merged or composed is made by detection strategies based on the metric values and thresholds. Detection strategies represent component detection heuristics and can be subdivided into strategies that suggest a merging step and strategies that suggest a composition step. The strategies are described in Section 2.3.2. The thresholds for merging and composition are changed over the iterations to lower the probability of a component merging and increase the probability of a composition. The thresholds are explained in detail in Section 2.3.3.

In the next step (6), the detected components of an iteration are integrated in the architecture result model. After that, the component interfaces are assigned (7). Similar to the detection of components, separate strategies exist for the detection of interfaces. Finally, the connectors between the components are created (8).

If no merge and no composition have been performed in the current iteration, the clustering terminates.

In a next step, the metrics have to be recalculated for the changed parts of the system and a new iteration can start.

### 2.3.1 Metrics

In contrast to source code metrics from object-oriented programs, for components only few metrics are available [CKK01, KP05, WYF03]. Therefore, most of the basic metrics used in SoMoX are adaptations of object-oriented metrics described by Martin [Mar94]. The metrics used in SoMoX are calculated for component candidates and deal with sets of classes, which represent a component candidate.



The metric value for a component candidate is a real number in the interval between 0 and 1.

The following presents an overview of the metrics that are relevant for this thesis.

**Coupling** The coupling metric used in SoMoX is an adaptation of the existing object-oriented coupling metric and reuses Martin’s concept of afferent coupling and efferent coupling. Afferent coupling is the number of types outside a component candidate that depend on types within the component candidate, while efferent coupling is the number of types inside a component candidate that depend on types that are outside the component candidate. *Coupling* is calculated as the ratio of accesses inside a component candidate to the total number of accesses. Assuming, that  $A$  and  $B$  are sets of classes, the concrete definition is depicted in Formula 2.1.

$$Coupling(A, B) := \frac{R(A, B)}{R(A, all)} = \frac{InternalAccesses}{ExternalAccesses} \quad (2.1)$$

Here,  $R(A, B)$  represents the number of accesses from  $A$  to  $B$  and  $R(A, all)$  stands for the number of accesses from  $A$  to all classes of the system. An access can be an access of a type, a method or a field. Coupling is not commutative, i.e.  $Coupling(A, B) \neq Coupling(B, A)$ .

**PackageMapping** The idea behind the package mapping metric is that the classes that belong to a common component are often located in the same package structure. The package structure is regarded as a tree structure. The metric is defined in Formula 2.2.

$$PackageMapping(A, B) := NonLinearMapping\left(\frac{commonRootHeight(A, B)}{maxHeight(A, B) - commonRootHeight(A, B)}\right) \quad (2.2)$$

In this formula,  $maxHeight(A, B)$  represents the maximum package tree height for elements of  $A$  and  $B$  and  $commonRootHeight$  represents the height of the deepest common node in the package tree of  $A$  and  $B$ . *PackageMapping* depends on *NonLinearMapping*, which filters out components that only share a very top-level package, by using a threshold of 0.2, as depicted in Formula 2.3.

$$NonLinearMapping(x) := \begin{cases} x & \text{if } x > 0.2 \\ 0 & \text{else} \end{cases} \quad (2.3)$$

**InterfaceViolation** The *Interface Violation* metric calculates the ratio of the number of accesses between two classes bypassing interfaces and the number of

all accesses. The definition is illustrated in Formula 2.4.

$$InterfaceViolation(A, B) := \frac{RI(A, B)}{R(A, all)} \quad (2.4)$$

$RI(A, B)$  represents the number of accesses from  $A$  to  $B$  that bypass interfaces. The metric value is 0 if the whole communication between  $A$  and  $B$  is accomplished via interfaces.

### 2.3.2 Component Detection Strategies

Each component detection strategy is used to identify characteristics of a potential component, like high coupling or interface communication. Strategies combine the metrics from Section 2.3.1 to form higher level recognition mechanisms.

The strategies operate on component candidates and evaluate whether a component candidate should become a component. A strategy results in a value between 0 and 1, where 1 suggests to convert a candidate into a component and 0 suggests to reject the component candidate. The result values from all strategy evaluations for one component candidate are aggregated into a value that indicates whether a component should be created from the candidate, or if the candidate should be rejected. Strategies are composable so that interdependencies among them can be expressed.

The component detection strategies used in SoMoX are Interface Adherence, Interface Bypassing, Consistent Naming, Abstract/Concrete Balance, Hierarchy Mapping, Subsystem Component, Component Merge and Component Composition.

The strategies needed for this thesis are explained below.

**Interface Adherence** The *Interface Adherence* strategy is based on the interface violation metric. The strategy checks whether component candidates are coupled at the code level prior to indicating interface communication. Interface adherence then results in a rating value of 0, if no coupling is present at the code level. Apart from that, component candidates with a clear interface communication style get a high interface adherence rating, which is derived from interface violations (see Formula 2.5).

$$InterfaceAdherence(A, B) := \begin{cases} 1 - \max(IV(A, B), IV(B, A)) & \text{if } \max(Coupling(A, B), Coupling(B, A)) > \varepsilon \\ 0 & \text{else} \end{cases} \quad (2.5)$$

**Hierarchy Mapping** The *Hierarchy Mapping* strategy is used to gain a language-independent component detection mechanism which evaluates the adherence

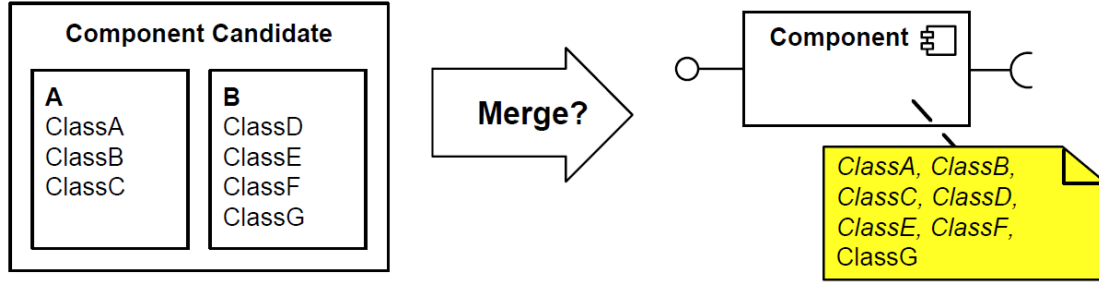


Figure 2.3: The merging of classes of a component candidate into a single component via the component merge strategy (from [Kro10])

of component candidates to hierarchies expressed in packages and directories. For Java-based systems, hierarchy mapping results in the same value, as the package mapping metric, while for systems written in C++, hierarchy mapping is equal to the metric *Directory Mapping*.

The most important strategies are the component merge strategy and the component composition strategy as they are used to make the final decision for a merge or a composition. The component merge and the component composition strategies share common sub-strategies.

### Component Merge

The *Component Merge* strategy decides whether to merge the elements of a component candidate into a single component, as depicted in Figure 2.3. A component merge lets the classes of a component candidate become members of one component. Merging is applied in the earlier iterations of the clustering to gain a higher abstraction level of basic components. This is controlled by a dynamic merge threshold (see Section 2.3.3).

The concrete formula of the merge strategy is given in Formula 2.6.

$$\begin{aligned}
 \text{ComponentMerge}(A, B) := & \frac{(w_1 * \text{InterfaceBypassing}(A, B) + \\
 & w_2 * \text{ConsistentNaming}(A, B) + \\
 & w_3 * \text{AbstractConcreteBalance}(A, B) + \\
 & w_4 * \text{HierarchyMapping}(A, B))}{4} \quad (2.6)
 \end{aligned}$$

The component merge strategy comprises interface bypassing, consistent naming, abstract/concrete balance and hierarchy mapping. The weights  $w_x$  for the sub-strategies are real numbers in the interval between 0 and 1. The weights are used to manually adapt the detection strategies to a specific system.

The component merge metric identifies situations where classes of a component candidate are strongly coupled, bypass interfaces in internal communication,

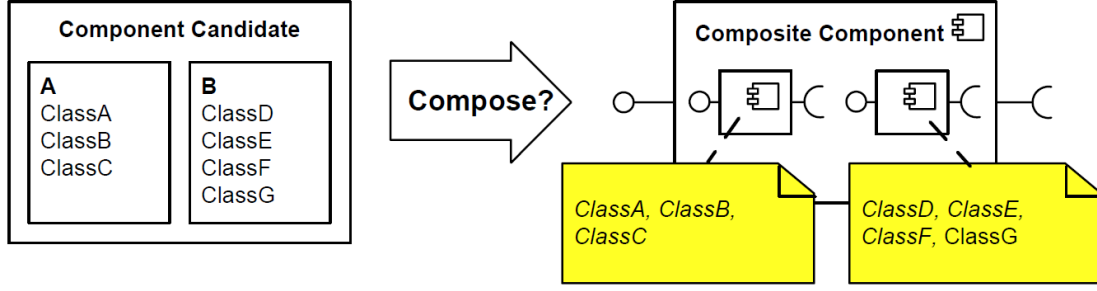


Figure 2.4: The creation of new composite components from a component candidate via the component composition strategy (from [Kro10])

have a consistent naming scheme and are located in the same area of the system hierarchy.

### Component Composition

The *Component Composition* strategy is responsible for the decision whether to convert a component candidate into a composite component, as depicted in Figure 2.4. This strategy prefers components which communicate via interfaces, which is the most important difference to the component merge strategy. Furthermore, in addition to the substrategies used in component merge, the subsystem component strategy is used to identify composition scenarios.

The concrete formula of the composition strategy is given in Formula 2.7.

$$\begin{aligned}
 \text{ComponentComposition}(A, B) := & (w_1 * \text{InterfaceAdherence}(A, B) + \\
 & w_2 * \text{ConsistentNaming}(A, B) + \\
 & w_3 * \text{AbstractConcreteBalance}(A, B) + \\
 & w_4 * \text{HierarchyMapping}(A, B)) \\
 & w_5 * \text{SubsystemComponent}(A, B)) \\
 & / 5
 \end{aligned}
 \tag{2.7}$$

The component composition strategy comprises interface adherence, consistent naming, abstract/concrete balance, hierarchy mapping and subsystem component. The weights can differ from the component merge strategy.

The dynamic threshold assures that high-level components with “a weak manifestation in artifacts” [Kro10] are identified for component composition.

### 2.3.3 Thresholds

SoMoX uses two separate thresholds for the merge step and for the composition step. These thresholds are dynamically changed over the iterations, which influences the abstraction levels of the resulting components. By this, the increasing

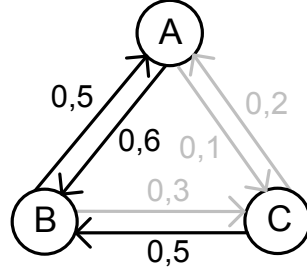


Figure 2.5: A filtered component indicating graph for the threshold  $t = 0.4$

abstraction in later iterations is ensured.

While the threshold for a component merge is increasing, the threshold for a component composition decreases. This lowers the probability of a component merging and increases the probability of a composition with each iteration.

The initial threshold values and the final threshold values as well as the decrementation / incrementation step width are configured by the user before the clustering.

The dynamic thresholds are only adapted if no new component has been identified in an iteration.

### 2.3.4 Component Indicating Graph

The algorithm which decides whether to merge or to compose components from a component candidate operates on a graph structure. Each element of a component candidate is represented by a vertex. Each component candidate is represented by a directed edge between those vertices with a weight that is derived from the component detection strategies. To determine a merge or a composition of a component candidate, the edges are filtered with regard to the thresholds, as shown in Figure 2.5. Component candidates that remain connected, are converted into components. In the example in Figure 2.5 the threshold is 0.4 and the edges that do not pass the threshold are marked gray. As a consequence A, B and C will be merged into the same component because they are still connected without the edges below the threshold.

The merge and the composition steps operate on the same graph structure, but the component detection strategies from which the graph is built differ and with them the edge weights which results in other sets of filtered edges.

Before the start of a new clustering iteration, only the metrics for the changed parts of the unfiltered graph are recalculated.

The component connectors are also derived from this graph.

### 2.3.5 Limitations of the Clustering

Besides the drawback that the clustering can only recover the structure of the components but not their purpose, as mentioned in Section 2.2, the clustering-based reverse engineering approach has some more drawbacks.

Another drawback is that all clustering decisions are based on metric values. The use of metrics cannot capture all architecture-relevant information and some useful information are too complex for these metrics, which are at a high level of abstraction [vDB11].

Another major problem occurs if the system to be clustered contains design deficiencies. One important metric for the clustering is the metric *Coupling*. Classes which are strongly coupled will probably be grouped together in the same component while uncoupled classes may be placed in different components. There are many design deficiencies like anti patterns and bad smells that increase the coupling between classes. One example for this is the bad smell *Interface Violation* (see Section 2.4). Engineers may unintentionally introduce such design deficiencies and thereby increase the coupling between classes that originally were not intended to belong to the same conceptual component. But the clustering results obviously reflect the actual architecture instead the conceptual architecture. Because of this, bad smells can adulterate the clustering decisions and as a consequence a misleading architecture model is created.

## 2.4 Bad Smells

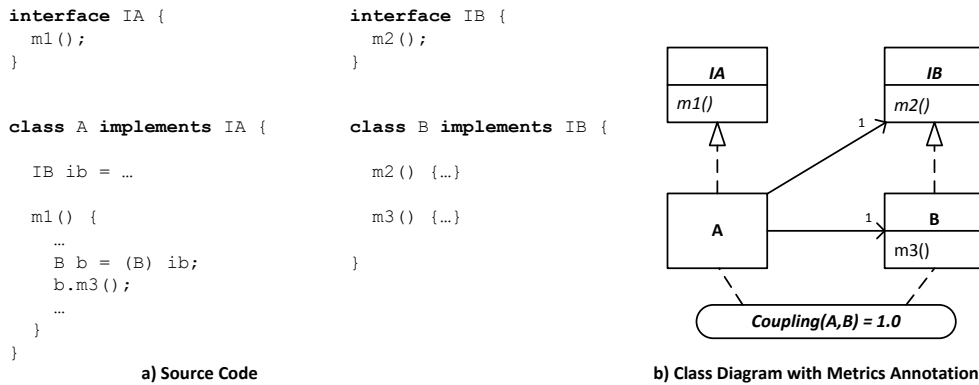
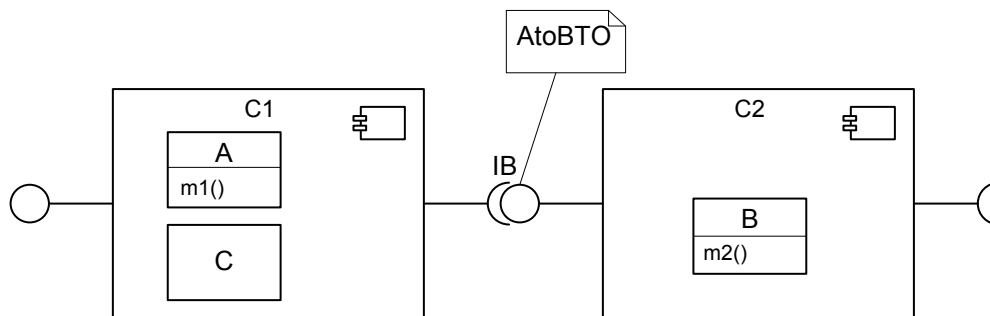
Software is often developed, adapted and maintained over a large period of time involving many different engineers. Because of this, it is often the case that design and implementation deficiencies like *Anti Patterns* [BMMM98] and *Bad Smells* [Fow99] are introduced.

A bad smell is a sign of a potential problem in a software system's design.

In the following sections, different bad smells that are used in this thesis are described.

### 2.4.1 Interface Violation

The bad smell *Interface Violation* describes a situation where an interface is intentionally bypassed. An example is illustrated in Figure 2.6 as source code extract (a) and as class diagram (b). The example system consists of two classes: A and B. The class A that implements an interface IA and the class B that implements an interface IB. Suppose that class A calls the method `m2()` of the interface IB and also the method `m3()` of the concrete class B that implements IB. To access the method `m3()`, class A has to downcast its IB object to the concrete type B. The expected way for the classes to communicate would be to rather communicate exclusively via the interfaces. However, this is not possible in this situation because

Figure 2.6: The bad smell *Interface Violation* (from [vDB11])Figure 2.7: Bad smell *Communication via Non-Transfer-Object* (from [vDB11])

the interface IB does not define the method `m3()`. Such a design flow could have been done by an unexperienced programmer.

Interface violations lead to a high coupling of the classes that are involved.

The Reclipse specification of interface violation used to detect this bad smell is depicted and described in Appendix A.1. The name of the specification is `IllegalMethodAccess` and it is an extended version of the specification created by Travkin [Tra11].

### 2.4.2 Communication via Non-Transfer-Objects

As pointed out in Section 2.1, in component-based software architectures, transfer objects should be used for the data exchange between two components.

In the system depicted in Figure 2.7, the two components C1 and C2 are connected via the interface IB and should therefore exchange data via the transfer object AToBTO. Instead of letting the class A pass a reference to C to the class B, it should use the transfer object and fill it with data from C. The consequence, if the communication is done directly and not via the transfer object, is that the coupling between A and B would be increased. Furthermore, class B gets access to all functionality of C, which is not intended by the conceptual architecture.

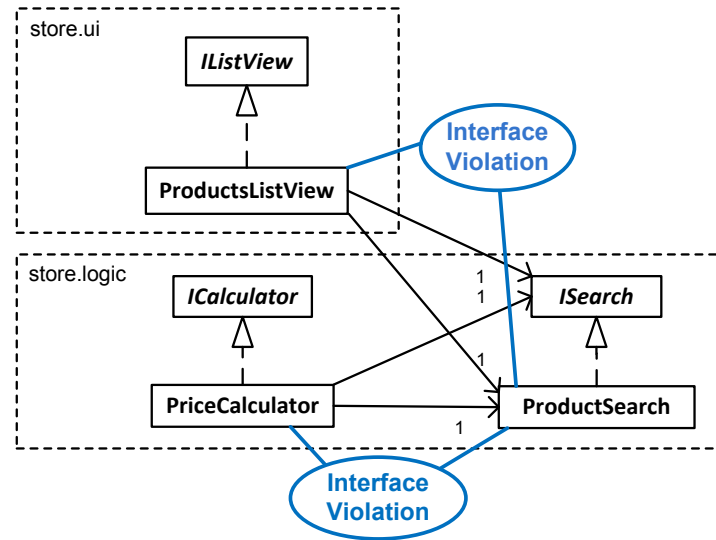


Figure 2.8: Packages and classes from the example store system

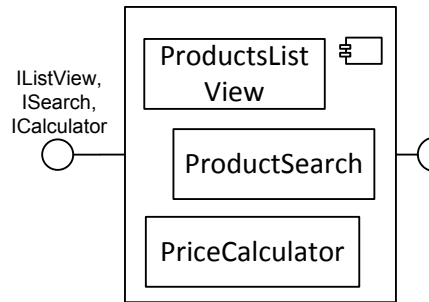


Figure 2.9: Recovered architecture of the example store system

The specification of this bad smell is shown in the appendix in Section A.1.

## 2.5 Running Example

Throughout this thesis, a simple program that represents a store system is used as a running example. The relevant classes are depicted in Figure 2.8. The store system contains the interfaces `IListView`, `ICalculator` and `ISearch` that are implemented by the concrete classes `ProductsListView`, `PriceCalculator` and `ProductSearch`. `IListView` and `ProductsListView` belong to the package `store.ui` which contains the view parts of the system. `ProductsListView` represents a view that shows a list which contains different products. `IListView` defines a general interface for views with list elements. The remaining elements belong to the package `store.logic` which consists of classes that are responsible for the business logic. The class `PriceCalculator` is used to calculate a products price, and the class `ProductSearch` implements a search algorithm for products.



The system contains two occurrences of the bad smell *Interface Violation*: one between `ProductsListView` and `ProductSearch` and another one between `PriceCalculator` and `ProductSearch`.

The architecture that is recovered with the clustering in SoMoX for this system is depicted in Figure 2.9. All classes are merged into the same component which has the interfaces `IListView`, `ISearch` and `ICalculator`.



# 3 Reengineering Process

As described in Chapter 1, the reengineer needs to be supported in several decisions:

- In which part of the system is the search for bad smells worthwhile?
- Which detected bad smells should be removed?
- How should the removal be accomplished best?

To solve these problems, an automatic relevance analysis with a subsequent architecture prognosis is proposed.

The new reengineering process is depicted in Figure 3.1. The rectangles represent process steps and the arrows represent the control flow between them. Most of the control flow arrows are annotated with the artifact that is the result of the previous step and is used in the next step. Additional icons differentiate between the steps that can be performed automatically and the steps that need user intervention.

The new reengineering process is based on the process proposed by Travkin et al. [vDB11] and presented in Chapter 2.2 but several steps were added.

Like the original process, the new process starts with a **Clustering** analysis that clusters the given software system into a component structure as described in Section 2.3. Thereby an initial architecture of the system is recovered. The clustering is done automatically, but the reengineer is involved in order to configure it.

Because of potential design deficiencies that may adulterate the clustering results, a bad smell detection follows. As Travkin describes [Tra11], a bad smell detection should be executed on one or more of the selected components separately, due to performance reasons. As a consequence, the reengineer has to select components from the initial architecture to build the search scope, before the bad smell detection can start. At this point, the contributions of this thesis start. To support the reengineer in her decision, which components are a worthwhile input for the bad smell detection, an automatic analysis can indicate components that seem to be critical. This **Component Relevance Analysis** rates the components that result from the clustering and thereby suggest a sensible input for the bad smell detection. Section 4.1 illustrates this procedure in detail.

After one or more relevant components have been chosen, the **Bad Smell Detection on Selected Components** can start. The detection is performed automatically, but the reengineer has to specify the bad smells that are to be detected. The

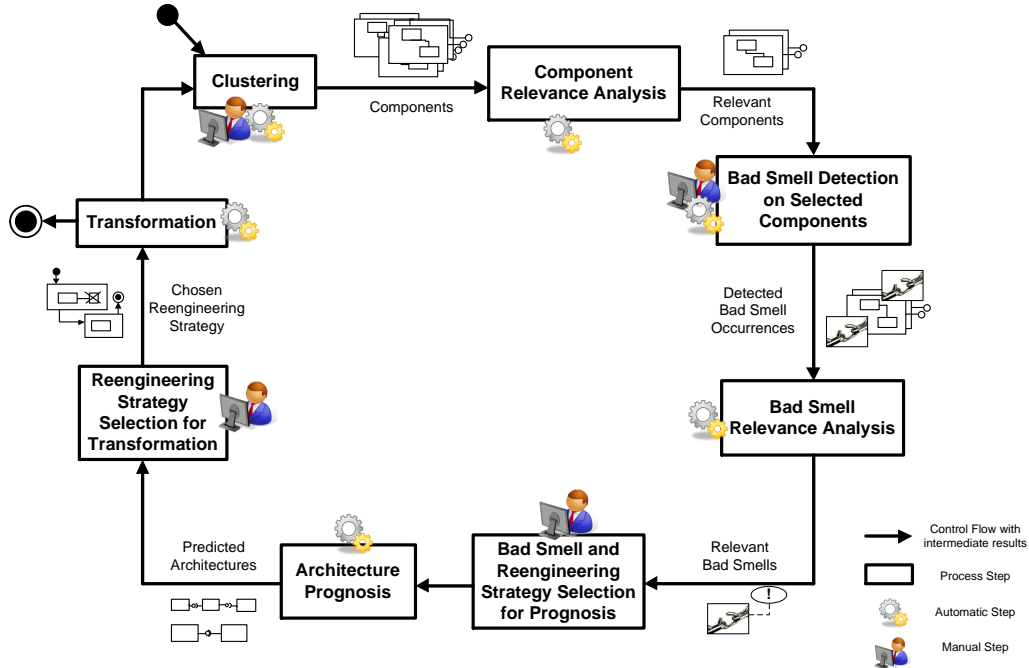


Figure 3.1: Reengineering process

detection results in a set of detected bad smell occurrences. This set may contain a large number of detection results and among these can be potential bad smell candidates that do not necessarily represent real design deficiencies. Section 4.2.1 details on this problem. Because of the presence of these detected bad smell occurrences that do not represent design deficiencies, the second step of the relevance analysis is needed: the **Bad Smell Relevance Analysis**. This analysis takes the unfiltered set of detected bad smell occurrences as input and rates the bad smell occurrences' relevance, and by this evaluates, which bad smell occurrences should be reengineered first. The rating is done automatically. Section 4.2 details on the rating algorithm.

With the help of the relevance analysis, the reengineer gets an overview of the severity of the bad smell occurrences. In the following step, the reengineer can select a relevant bad smell occurrence that should be removed. Hitherto, the removal of the bad smell had to be done manually by the reengineer. To accomplish the removal of the selected bad smell, she also has to select an adequate reengineering strategy that performs the removal. To support the decision for a reengineering strategy an **Architecture Prognosis** can be executed. The architecture prognosis takes the selected bad smell occurrence and a reengineering strategy as input. Based on this input, the reengineer gets a preview of the system's design as it looks after the removal of the bad smell. For each available reengineering strategy, an architecture prognosis can be done. The architecture prognosis is illustrated in

---

Section 6. It is executed automatically.

Using the information gained from the architecture prognosis, the reengineer can choose her preferred way to remove the bad smell in the next step (**Reengineering Strategy Selection for Transformation**) to execute the actual transformation step. As a last step, the selected strategy can be applied by executing an automatic **Transformation**.

The resulting system with the new architecture can then be the input for a new clustering iteration. In the future this step could be improved if an architecture created in the architecture prognosis is used, instead of executing a new clustering. After the clustering the reengineer has to decide if she is satisfied with the newly recovered architecture, or if she wants to start a new iteration of the reengineering process, to further improve this architecture by removing further bad smells, if possible.

The process contains steps that are executed automatically and steps in which the reengineer is involved, which makes this process semi-automatic. Thereby the important decisions are left to the human, but automatic tools provide support to simplify this by helping the reengineer to make more informed and thereby better decisions.

The clustering and the bad smell detection are already available, as pointed out in Chapter 2. This thesis focuses on the relevance analysis and the architecture prognosis. These steps are specified in more detail in the following chapters.



## 4 Relevance Analysis

In the reengineering process presented in Chapter 3, one or more components that were identified in the clustering, can be selected to be the search scope of the subsequent bad smell detection. For this, the user has to decide, in which component an analysis could be worthwhile. In the original reengineering process, she had to do this manually. This is a time-consuming task because it requires a close inspection of all components.

After the bad smell analysis is executed, the reengineer sees herself confronted with a high amount of detected bad smell occurrences. An occurrence of a bad smell in a software system is not necessarily a design flaw. Depending on the context in which the bad smell occurs, some bad smell occurrences may be more critical than others (see Section 4.2 for further explanations).

For this reason, the relevance of each detected bad smell occurrence has to be analyzed so that it can be determined if the occurrence should be removed or not. Currently, this has to be done manually by inspecting each bad smell occurrence. Such an inspection includes a detailed look at all the classes, methods and attributes that are involved in the bad smell occurrence as well as inspecting their context. Furthermore the discovered characteristics of the bad smell occurrences have to be compared with each other. As a consequence, this inspection obviously is a tedious task.

In this thesis I present a concept to automatically determine the components' relevance for a bad smell detection. This automated analysis simplifies and speeds up the decision-making process for the reengineer and helps her to give a more informed decision. Section 4.1 details on this approach.

Furthermore an analysis is presented, in which the bad smell occurrences and their impact on the software architecture of the system are analyzed automatically. The concept for this analysis is pointed out in Section 4.2.

This leads to a *Relevance Analysis* that involves two steps: the identification of relevant components and the identification of relevant bad smells.

Both steps are realized by a rating concept that determines relevance values by using a composition of different strategies.

The remainder of this chapter proceeds with describing the identification of relevant components and then the identification of relevant bad smells. In both sections, first the concept is motivated, then the integration in the reengineering process is illustrated. Next the rating strategies are explained in detail and at last it is illustrated how the rating result is calculated from the strategies.

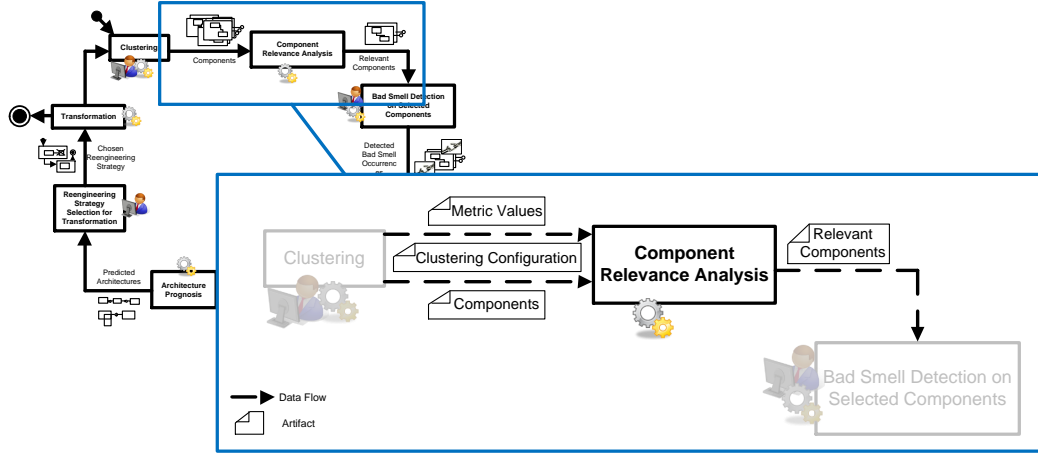


Figure 4.1: The component relevance analysis in the reengineering process

## 4.1 Rating Concept for Relevant Components

The first step of the relevance analysis is the *component relevance analysis*. It is executed after the clustering and before the bad smell detection.

### 4.1.1 Motivation

To improve a system's quality, bad smells in the system's architecture are to be detected. As pointed out in Section 2.2, a drawback of the bad smell detection is its performance. Even for middle-sized systems it can take several hours till the reengineer gets usable results. Furthermore, the set of results usually is unpractically large. As a consequence, the input for the analysis has to be reduced to avoid these problems.

As Travkin proposes, to narrow down the search scope, one or more components of the system under analysis can to be selected for the bad smell detection [Tra11]. The components for the selection on which part of a system the bad smell detection is executed are taken from the architecture model that is created during the clustering.

To reduce the time required for the whole detection and reengineering process, the reengineer should start his search in a relevant component. A relevant component is promising to contain design deficiencies whose removal have a significant impact on the system's architecture.

In the component relevance analysis, relevant components are identified to guide the user's decision in which component of a software system the search for bad smells could be worthwhile.



### 4.1.2 Integration in the Reengineering Process

f is performed after the clustering and requires the metric values, the clustering configuration and the components from the clustering, as depicted in Figure 4.1. It results in a set of relevant components that are proposed to be the subject to a bad smell detection. The bad smell detection is the subsequent step.

### 4.1.3 Rating Strategies

Two different strategies are used to rate a component's relevance: the *Closeness to Threshold Strategy* and the *Component Complexity Strategy*.

**Closeness to Threshold** In this strategy, the current merge and composition thresholds are regarded. The decision for or against a merge or a composition is derived from the metric values that are composed to a merge and a composition metric. The occurrences of bad smells can adulterate the metric values as shown in Section 2.3. It is possible that if the values for a merge or composition metric are close to the threshold, the decision for or against the merge or composition could be wrong because of bad smells. Because of this, the modification of components that originate from such potentially adulterated decisions, could have a great impact on the architecture, when they are modified. This makes them relevant to search for bad smells.

To determine a concrete relevance value according to this assumption, in addition to the thresholds, the merge and composition metric values are required. The metric values are determined for the component candidates from the different iterations in the clustering, but only the resulting components from the clustered architecture model are an output of the clustering. So the component candidates from the iterations that correspond to the components from the architecture model have to be derived from the components. This is done by comparing the classes of the component candidates and the classes that the components contain.

The iterations with the lowest current merge threshold are the first iterations in a clustering run because the current merge threshold is increased over the iterations and the probability for a merge decreases. According to this, the most relevant components are components that contain classes from component candidates whose merge value in the iterations with the lowest current merge threshold was narrow to the current merge threshold.

On the other hand, the iterations with the lowest current composition threshold are the last iterations because the current composition threshold is increased over the iterations and the probability for a composition increases. Thus, also components that contain classes from component candidates whose composition value in the iterations with the lowest current composition threshold was narrow to the current composition threshold are relevant.

The formulas for the exact relevance value for the *Closeness To Threshold* strategy (CTT) are depicted in Formula 4.1 to 4.5.

$$\begin{aligned}
comp &:= \text{the current component, contains a set of classes } \mathbf{Classes} \\
FirstIts &:= \text{the iterations with the lowest current merge threshold} \\
LastIts &:= \text{the iterations with the lowest current compose threshold} \\
CCands_i &:= \text{the set of component candidates in iteration } i \\
cc \in CCands_i &:= (ClassesA, ClassesB)
\end{aligned} \tag{4.1}$$

$$\begin{aligned}
rMerge_{cc} &:= \begin{cases} 1 & \text{if } |Merge - mergeThreshold| < \varepsilon \\ 0 & \text{else} \end{cases} \\
rCompose_{cc} &:= \begin{cases} 1 & \text{if } |Compose - composeThreshold| < \varepsilon \\ 0 & \text{else} \end{cases}
\end{aligned} \tag{4.2}$$

$$v_{comp,cc} := \begin{cases} 0 & \text{if } \#(cc.ClassesA \cap comp.Classes) = 0 \wedge \\ & \#(cc.ClassesB \cap comp.Classes) = 0 \\ 1 & \text{if } \#(cc.ClassesA \cap comp.Classes) \geq 1 \oplus \\ & \#(cc.ClassesB \cap comp.Classes) \geq 1 \\ 2 & \text{if } \#(cc.ClassesA \cap comp.Classes) \geq 1 \wedge \\ & \#(cc.ClassesB \cap comp.Classes) \geq 1 \end{cases} \tag{4.3}$$

$$\begin{aligned}
CTT(comp) &:= \sum_{i \in FirstIts} \left( \sum_{cc \in CCands_i} v_{comp,cc} \cdot rMerge_{cc} \right) \\
&+ \sum_{i \in LastIts} \left( \sum_{cc \in CCands_i} v_{comp,cc} \cdot rCompose_{cc} \right)
\end{aligned} \tag{4.4}$$

$$AllCCands := \sum_{i \in FirstIts \cup LastIts} \#CCands \tag{4.5}$$

$$CTT_{norm}(comp) := \frac{CTT(comp)}{AllCCands \cdot 2}$$

The calculation of the relevance value takes five steps:

1. As depicted in Formula 4.1, *comp* is defined as the current component for that the relevance is calculated. As explained above, the first iterations (*FirstIts*) and the last iterations (*LastIts*) are considered. *CCands<sub>i</sub>* is the set of component candidates contained in the architec-

ture model of the current iteration. Each of these candidates consist of two sets of classes: *ClassesA* and *ClassesB*.

2. *rMerge* and *rCompose* indicate if a component candidate is relevant and included in the rating (see Formula 4.2). They are determined by calculating the deviation of the threshold from the merge value or the compose value, respectively. If the deviation is greater than a chosen bound  $\varepsilon$ , *rMerge* or *rCompose* are set to 1, otherwise the value is 0.
3.  $v_{comp,cc}$  represents the rating value for a (*component*, *componentcandidate*)-tuple. It is defined as illustrated in Formula 4.3: The rating value is 0 if the two components of the component candidate *cc* and the component *comp* have no classes in common, 1 if one of the two components of *cc* and *comp* have at least one class in common, and 2 if both components of *cc* share at least one class with *comp*.
4. For the result of the relevance strategy *CTT*, the rating value then is multiplied with the *rMerge* value or the *rCompose* value, respectively, as shown in Formula 4.4. This is done for each component candidate in the first iterations or the last iterations, respectively.
5. To make the relevance values of different components comparable, they are normalized. This is done by dividing the value of *CTT* by  $AllCCands \cdot 2$  (Formula 4.5). *AllCCands* represents the sum of the number of component candidates from all regarded iterations, i.e., the iterations with the lowest merge value and the iterations with the lowest compose value. The factor 2 is needed because a component candidate consists of two components which can increase the rating value by 2.

**Component Complexity** Complex components consist of many classes, attributes, methods and interfaces. Because of this, they are unclear and confusing, difficult to maintain and to adapt, and this situation worsens more and more over the time. Thus, the risk of accidentally embedding design deficiencies increases. This leads to the assumption that, the more complex the component, the more likely it is to contain bad smell occurrences. This makes the complexity of a component significant to rate the relevance of a component for the bad smell detection.

In this thesis, the complexity is calculated by using a simplified version of the formula for the *Plain Component Complexity* as described by Cho et al. [CKK01]. There, the sum of classes, interfaces, and methods and the complexity of classes and methods is calculated. In this thesis, the formula depicted in Formula 4.6 is used.

$$\begin{aligned}
Complexity(comp) := & \#classes(comp) \\
& + \#interfaces(comp) \\
& + \#methods(comp) \\
& + \sum_{c \in Classes(comp)} \#attributes(c) \\
& + \sum_{m \in Methods(comp)} \#arguments(m)
\end{aligned}$$

$$\begin{aligned}
MaxSum := & Overall\#Classes + Overall\#Interfaces \\
& + Overall\#Methods + Overall\#Attributes \\
& + Overall\#Arguments
\end{aligned}$$

$$Complexity_{norm}(comp) := \frac{Complexity(comp)}{MaxSum} \quad (4.6)$$

The sum of the classes, interfaces, methods, attributes and arguments of a component is divided by the sum of the classes, interfaces, methods, attributes and arguments of all components, to normalize the complexity value.

All relevance strategies result in a value between zero and one.

How these relevance values are processed further is described in the following section.

#### 4.1.4 Rating Result

To identify relevant components from the rating values the different strategies provide, further calculations have to be done.

For this purpose, the components that are pareto optimal with respect to the relevance strategies are highlighted in the visualization of the analysis results.

The *pareto optimal* components are assumed to be good subjects for a bad smell detection because they represent the best available combination of relevance values, i.e., they are built from component candidates that are close to the merge or composition thresholds and in addition, they are among the most complex components of the system. Thus, the reengineer can directly focus on these candidates and thereby she can more easily and quickly continue with the reengineering process.

The pareto optimal set contains solutions that represent the best possible trade-off among the objectives [CDJ10]. A solution is called pareto optimal if and only if there is no solution that dominates this solution. Here, we use the dominates

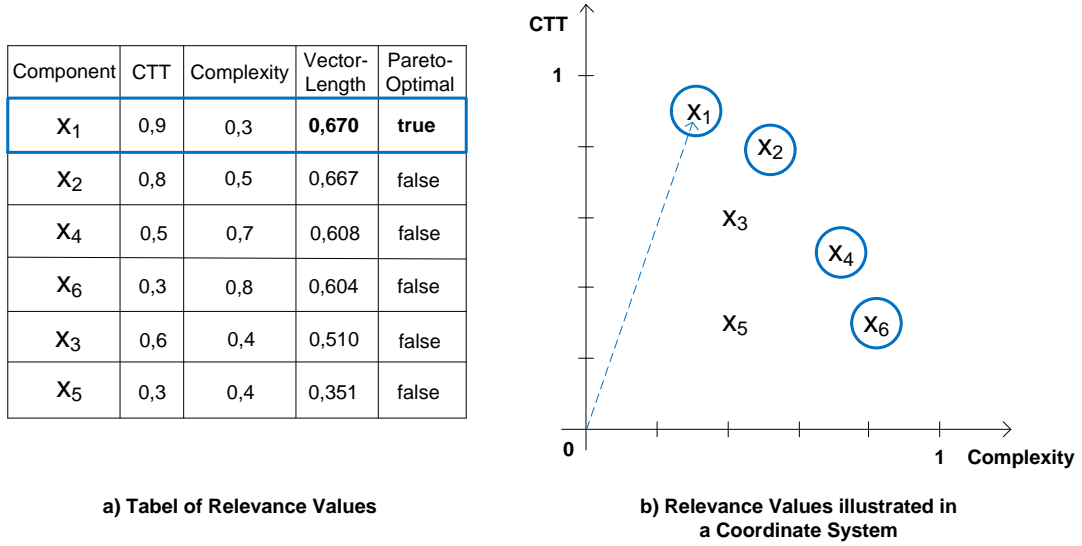


Figure 4.2: Example for the calculation of the relevance values result

relation in a maximization context: A solution  $y$  dominates a solution  $z$  iff  $\forall i \in [1...n], f_i(y) \geq f_i(z)$  and  $\exists i \in [1...n]$  such that  $f_i(y) > f_i(z)$ .

In the cases where several pareto optimal solutions exist, a further criterion is required to determine the most relevant component. Because of this, in addition to determining the pareto optimality, the length of the vector from the origin to the point constituted by the relevance values in a multi dimensional space, is calculated. The higher the geometric distance to the origin, the more relevant is the corresponding component. The resulting values are normalized to a value between zero and one to simplify the comparison. The resulting formula is depicted in Formula 4.7.

$$Relevance(C) := \frac{\sqrt{\sum_{i=1}^n v_i^2}}{\sqrt{n}} \quad |v_i \in [0; 1] \quad (4.7)$$

with  $v$  as the relevance value of the strategy  $i$  and  $n$  being the total number of strategies.

Figure 4.2 visualizes a set of example relevance values as table (a) and as graph (b). As illustrated in the graph, each relevance strategy defines a dimension: the Complexity strategy represents the x-axis and the Closeness To Threshold strategy represents the y-axis.  $x_1$  to  $x_6$  represent the components. The pareto optimal components  $x_1, x_2, x_4$  and  $x_6$  are marked with a blue frame. They build up a *pareto front*. Each candidate below the pareto front is dominated by the other candidates, hence, it is not pareto optimal (here:  $x_3$  and  $x_5$ ) and therefore less interesting. The pareto optimal candidates differ in their distance from the origin.  $x_1$  is the component with the largest origin vector, which is marked with a blue arrow. Note that the length of the vectors from the origin has been normalized.

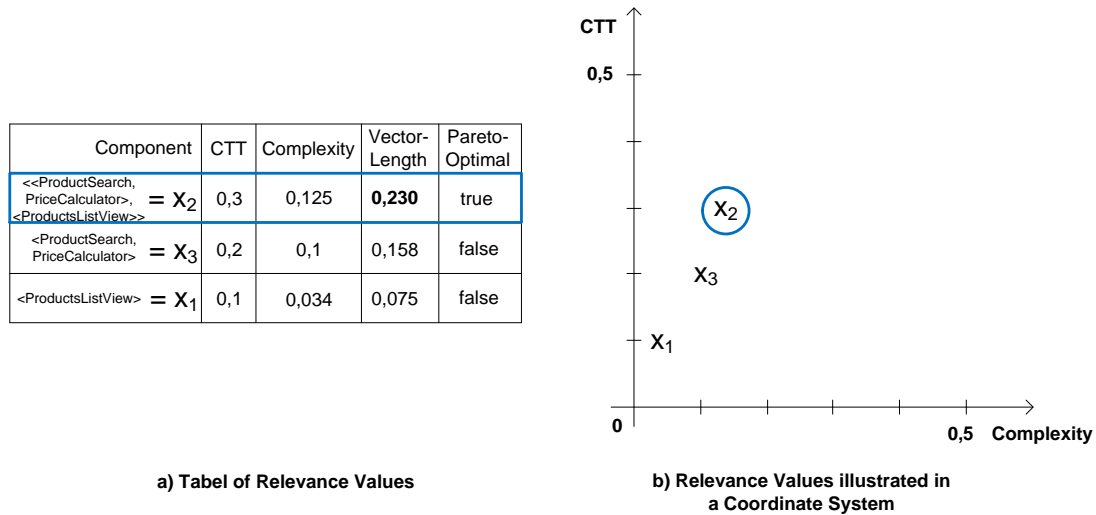


Figure 4.3: Relevance values results for the running example

The resulting graph for the running example is depicted in Figure 4.3. In this example, only one pareto optimal component exists:  $x_2$ . It represents the composite component that consists of the two other components in this example, which leads to this clear result.

This approach to calculate an overall result for the relevance is easily extendable. Since the pareto optimality as well as the length of the vector to the origin are computable for arbitrarily many dimensions. Any number of strategies can be added to rate the relevance of a component.

## 4.2 Rating Concept for Relevant Bad Smell Occurrences

In the following sections, the concept for the *bad smell relevance analysis* is explained. Here, the relevance of a bad smell occurrence is determined on the basis of the metric values of the clustering analysis.

The bad smell relevance analysis is executed after the bad smell detection and before the decision for a reengineering strategy is made.

### 4.2.1 Motivation

The example store system as introduced in Section 2.5 contains two occurrences of the bad smell *Interface Violation*: one between the classes `ProductsListView` and `ProductSearch` and another one between the classes `PriceCalculator` and `ProductSearch`, as depicted in Figure 4.4.

In component-based software architectures, the communication between components is strictly defined. The communication within a component can be

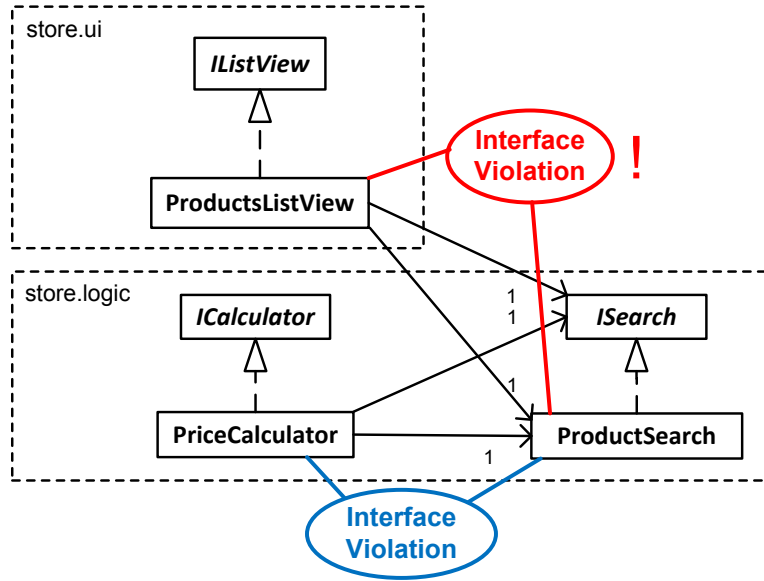


Figure 4.4: Example system with one relevant and one less relevant bad smell

handled more easily, for example for a better efficiency. As a consequence, design deficiencies regarding the communication between components can be distinguished by heuristics using this knowledge. For example, regarding the fact that `ProductsListView` and `ProductSearch` belong to different conceptual components, the interface violation between those classes probably is a design deficiency and should be removed. In contrast, the interface violation between the classes `PriceCalculator` and `ProductSearch` which belong to the same part of the system, may be intended and is not necessarily a deficiency of the architecture.

To conclude, not all bad smell occurrences are equal. Heuristics can be used to distinguish bad smell occurrences that are really problematic from occurrences that can be tolerated in a component-based software architecture. Heuristics about the design of the system under analysis are already available by the metrics used in the clustering and partly reused in the component relevance analysis. In the bad smell relevance analysis, the metric values are used again, but this time to rate the relevance of bad smell occurrences.

In the simple example used above, the relevance can be determined by regarding the locations of the classes, i.e. the membership to a package in java. The metric *Package Mapping* used in the clustering (see Section 2.3.1) represents this heuristic, so this can be used here, to indicate the relevance of these interface violation occurrences.

The remainder of this chapter details on how the package mapping metric is used for the relevance analysis and which other metric values from the clustering can be used to evaluate the relevance of this and other bad smells.

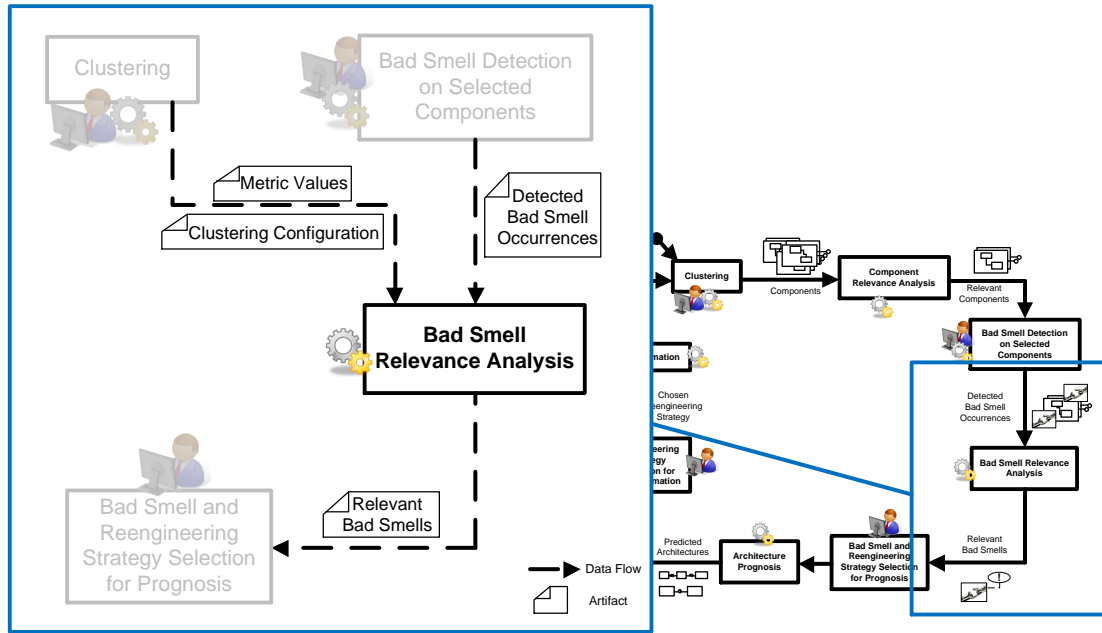


Figure 4.5: The bad smell relevance analysis in the reengineering process

### 4.2.2 Integration in the Reengineering Process

In the reengineering process, the bad smell relevance analysis is the step after the bad smell detection, as depicted in Figure 4.5. It takes the detection results, i.e. the detected bad smell occurrences, as input, as well as the metric values and the clustering configuration. The relevant bad smell analysis results in a set of relevant bad smells.

### 4.2.3 Rating Strategies

Similar to the components relevance analysis, several rating strategies are used to determine the relevance value of a bad smell occurrence. The applicability of the strategies depends on the bad smell types.

The relevance strategies are described below.

**Class Locations** The idea behind this strategy is that classes that reside in the same part of the system (i.e. are in the same branch in the package tree or even belong to the same package), are intended to collaborate with each other. Consequently, an occurrence of a bad smell like *Interface Violation* between classes that are located far away from each other, is a more serious design problem, than an occurrence between classes in the same package. For this strategy, the value of the *PackageMapping* metric is used. The



exact formula is depicted in Formula 4.8.

$CC_{BS}$  := Component Candidate that corresponds to the Bad  
Smell Occurrence  $BS$

$$Relevance_{CL}(BS) := 1 - PackageMapping(CC_{BS}) \quad (4.8)$$

Here,  $BS$  is the bad smell occurrence and  $CC_{BS}$  represents the component candidate that contains the classes that are involved in the bad smell occurrence  $BS$ . The higher the *PackageMapping* value, the less relevant the occurrence is rated.

This strategy is applicable for the bad smells *Interface Violation* and *Communication via Non-Transfer Objects*.

For non-java-based systems, the same strategy can be used with the *DirectoryMapping* metric [Kro10] instead of the *PackageMapping* metric.

**Number of External Accesses** In order to achieve high reusability, different components of a system ought to be loosely coupled [Mye75, LTC02]. In a clustering algorithm, high coupling between components is an indicator for a component merge [CKK08, Kro10]. As pointed out in Section 2.3.1 the metric *Coupling* is defined as the ratio of internal accesses and external accesses. The bad smell *Interface Violation* increases the numbers of internal accesses and external accesses for a component candidate by two for each. This implies that *Interface Violation* occurrences adulterate the component design especially for components with few external accesses. In contrast, an *Interface Violation* occurrence in a component with many external accesses is not as problematic. Because of this, bad smell occurrences in a component candidate with a high *External Accesses* metric value are rated as less relevant than occurrences in a candidate with a lower *External Accesses* value, as illustrated in Formula 4.9.

$$Relevance_{EA}(BS) := 1 - ExternalAccesses(CC_{BS}) \quad (4.9)$$

This strategy is applicable for the bad smell *Interface Violation*.

**Higher Interface Adherence** The *Higher Interface Adherence* strategy does a prediction for the reengineered system in which the regarded bad smell has been removed. For this estimation, the fact that the value for the metric *InterfaceAdherence* increases, if an *Interface Violation* occurrence was removed, is used.

The *InterfaceAdherence* metric value only takes part in the calculations for the overall metric values, if the metric value for *Coupling* is greater or equal

to  $\varepsilon$  (see Section 2.3.2). Because of this, the Higher Interface Adherence strategy returns zero for component candidates whose coupling is less than  $\varepsilon$  (see Formula 4.10). Otherwise, the following steps are processed to calculate a relevance value for this strategy:

First, a higher value *InterfaceAdherence* value has to be chosen. Currently, the maximum value (1) is used for this. In the future a better heuristic can be applied to search a more adequate value.

Then a new value for the overall *Merge* metric is calculated for the component candidate that corresponds to the bad smell occurrence, as described above. The new value is based on the new *InterfaceAdherence* value, while the values of the other basic metrics remain unchanged.

Next, the relation of the new *Merge* value to the *CurrentMergeThreshold* is compared to the relation of the original *Merge* value to the threshold. Only the cases in which the relation has changed are of interest:

1. the newly calculated *Merge* value is lower than the current merge threshold, but the old value was higher, or
2. the new *Merge* value is higher than the threshold, but the old value was lower.

If one of these cases is true, the result is the deviation of the value to the threshold. Otherwise, the result is zero. Formula 4.11 shows the exact formula.

$$Relevance_{HIA}(BS) := \begin{cases} 0 & \text{if } (coupling(CC_{BS}) < \varepsilon) \\ Dev(CC_{BS}) & \text{else} \end{cases} \quad (4.10)$$

$$Dev(CC_{BS}) := \begin{cases} |t_{Merge} - Merge_{new}| & \text{if } ((Merge_{old} < t_{Merge} \\ & \wedge Merge_{new} \geq t_{Merge}) \\ & \vee (Merge_{old} \geq t_{Merge} \\ & \wedge Merge_{new} < t_{Merge})) \\ 0 & \text{else} \end{cases} \quad (4.11)$$

Here,  $t_{Merge}$  represents the current merge threshold,  $Merge_{new}$  represents the new calculated value for the overall *merge* value regarding an *InterfaceAdherence* value of 1, and  $Merge_{old}$  represents the old overall *Merge* value with the original *InterfaceAdherence* value.

In the future, this strategy could be extended to also consider the *Compose* metric value.

This strategy is applicable for the bad smell *Interface Violation*.

**Communication via Data Classes** As explained in Section 2.1, transfer objects serve as data containers for messages between components. As a consequence, transfer object classes are simple data classes that do not contain any methods that implement the application logic. In a good component-oriented design, transfer object classes are marked as such.

In the clustering with SoMoX, transfer objects are recognized and not assigned to any components. Because of this, a *Communication via Non-Transfer Objects* occurrence is less relevant for reengineering, if the non-transfer object is a data class, which indicates that an incorrectly marked transfer object is used. The closer the non-transfer object class comes to being a data class, the more the relevance value decreases (see Formula 4.12).

$$Relevance_{DC}(BS) := 1 - IsDataClass(BS.dataClass) \quad (4.12)$$

$$IsDataClass(c) := \begin{cases} 0 & \text{if } \#Fields(c) = 0 \\ 1 - \left( \frac{\#AllMethods(c)}{\#NonAccessors(c) + MissingAccessors(c)} \right) & \text{else} \end{cases} \quad (4.13)$$

$$MissingAccessors(c) := |2 \cdot \#Fields(c) - \#Setters(c) - \#Getters(c)| \quad (4.14)$$

`BS.dataClass` represents the non-transfer object class.

Formula 4.13 depicts how a the similarity of the non-transfer object class to a data class is calculated. A regular data class has a two accessor methods for each field: one getter and one setter. The *MissingAccessors* formula (see Formula 4.14) is calculated by counting the fields and subtracting the number of getters and setters. By this a wrong number of accessor methods is detected. The more the number of accessors deviates from the regular case, the higher is the *MissingAccessors* value.

In the *IsDataClass(c)* formula, the *MissingAccessors* value is added to the number of methods that are not getters or setters. Then the number of all methods in the class is divided by the sum. The result is then subtracted from one. If the class contains no fields (it is not a data class) *IsDataClass* returns 0 and the relevance strategy returns 1, i.e., the bad smell occurrence is very relevant.

Data classes can be identified during different steps in the process. For classes that have already been identified as data class in the clustering, this strategy returns 0.

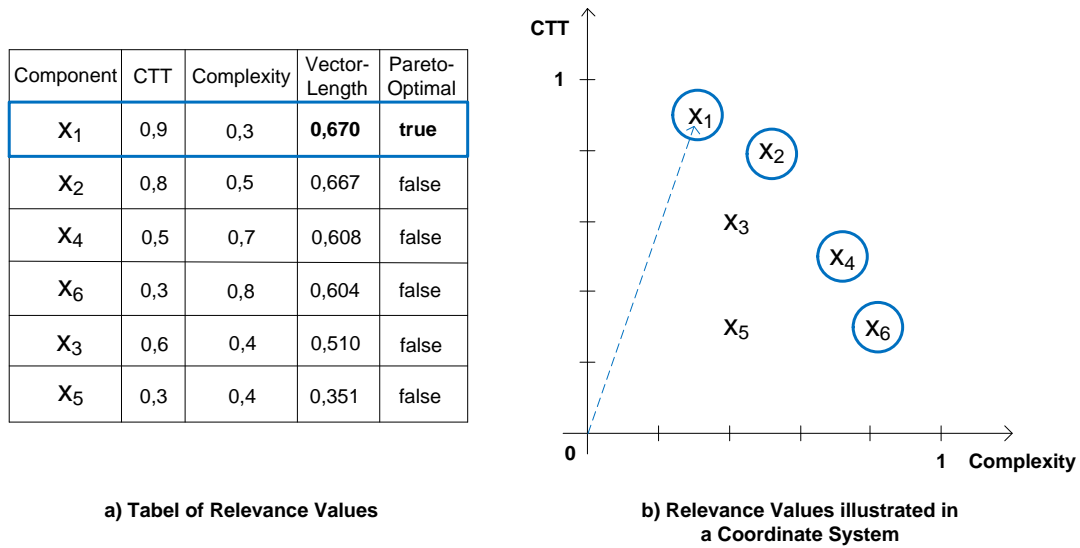


Figure 4.6: Results of the bad smell relevance analysis for the running example

This strategy is only applicable to rate the relevance of the bad smell *Communication via Non-Transfer Objects*.

#### 4.2.4 Rating Result

The overall rating result for the bad smell occurrences is determined in the same way as in the relevance analysis for components (see Paragraph 4.1.4). The only further restriction is that it has to be taken into account if a certain strategy is applicable for the current bad smell, or not.

The results for the two Interface Violation occurrences in the running example are depicted in Figure 4.6. Since this thesis presents three relevance strategies for the Bad Smell Relevance Analysis for Interface Violation occurrences, three dimensions are taken into account this time: Class Locations (CL), Number of External Accesses (NEA) and Higher Interface Adherence (HIA). Both occurrences are pareto optimal, but considering the distance to the origin, the interface violation in the class `ProductsListView` is more relevant, than the interface violation in `PriceCalculator`.

A high relevance value indicates a high probability that the current bad smell occurrence is a good subject to reengineering and that the reengineering would change the system's recovered architecture. However, it is not guaranteed that the recovered architecture is significantly influenced by the removal of the bad smell occurrence.

## 5 Reengineering Strategies

After the Reengineer, supported by the relevance analysis, has decided, which bad smell occurrence should be removed, she has to find a way to accomplish this. This currently has to be done manually by first identifying appropriate *Reengineering Strategies* by consulting design experts or adequate literature, if necessary.

In many cases there are different reengineering strategies to accomplish the removal of a bad smell. Then, the reengineer has to decide which strategy fits her requirements best. To determine the best reengineering strategy for the removal of a bad smell occurrence, the consequences on the system's architecture are an important criterion. If this has to be done manually, the task of deciding on an appropriate strategy becomes time-consuming.

To support this process, different reengineering strategies for specified bad smells can be specified. For a detected bad smell occurrence, the appropriate reengineering strategies are then presented to the reengineer. Then, the reengineer can select between the proposed reengineering strategies in order to perform an architecture prognosis (see Chapter 6) that shows the impact of the application of the reengineering strategy on the system's architecture. Thus, the reengineer can easily make a more informed decision for her reengineering, to get the expected results.

For the selected interface violation occurrence from the running example (see Chapters 2 and 4), the reengineer has several possibilities to correct this design deficiency. Here, two strategies are explained exemplarily.

First, she could simply remove the call and the cast as illustrated in the activity diagram in Figure 5.1. In consequence, the behavior of the system is modified because a part of the method's functionality gets lost.

The other possibility is to extend the interface by adding a new method declaration to the interface (see Figure 5.2). Then the method of the interface can be called instead of the method of the concrete subclass. Furthermore, the cast can be removed. The modification of the interface has the consequence that other classes that implement that interface have to implement the new method, too.

In Figure 5.3, an extract of the original source code (a) and of the system after the application of the both strategies (b+c) is depicted. In this source code

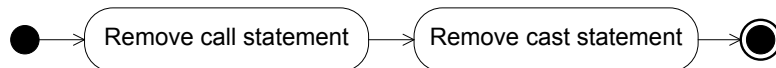


Figure 5.1: Reengineering strategy that removes the call as activity diagram

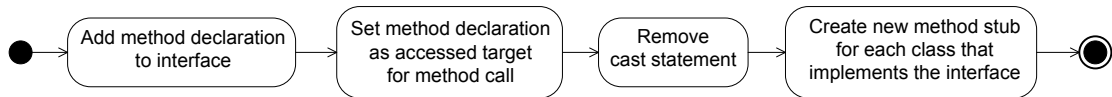


Figure 5.2: Reengineering strategy that extends the interface as activity diagram



Figure 5.3: Source code example for interface violation and reengineered systems

extract, the `@Override` annotations are used to illustrate the differences in the system that are introduced with the adaptation of the interface.

The lines responsible for the interface violation occurrence are marked red. They are contained in the `printList()` method of the `ProductsListView` class and include the downcast of the object `search` to the concrete type `ProductSearch` as well as the call of the method `searchProducer()`.

The changes done by the reengineering strategies are marked in blue. The result of the application of the reengineering strategy that removes the call is depicted in part b. The only part of the system that changes is the method `printList()`.

The second reengineering strategy is more complex. A method declaration for the method `searchProducer()` is added to the interface `ISearch` and the `searchProducer` method in the concrete class `ProductSearch` now implements the method from the interface. This leads to the fact that the `searchProducer()` call in `printList()` can be done on the object `search` of the type `ISearch` and because of this, the line with the cast statement can be deleted. As stated

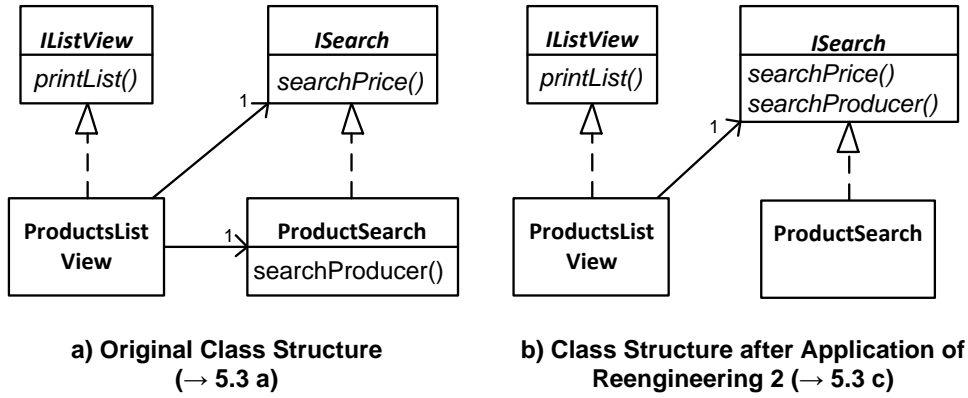


Figure 5.4: Class diagrams for the original and the reengineered system

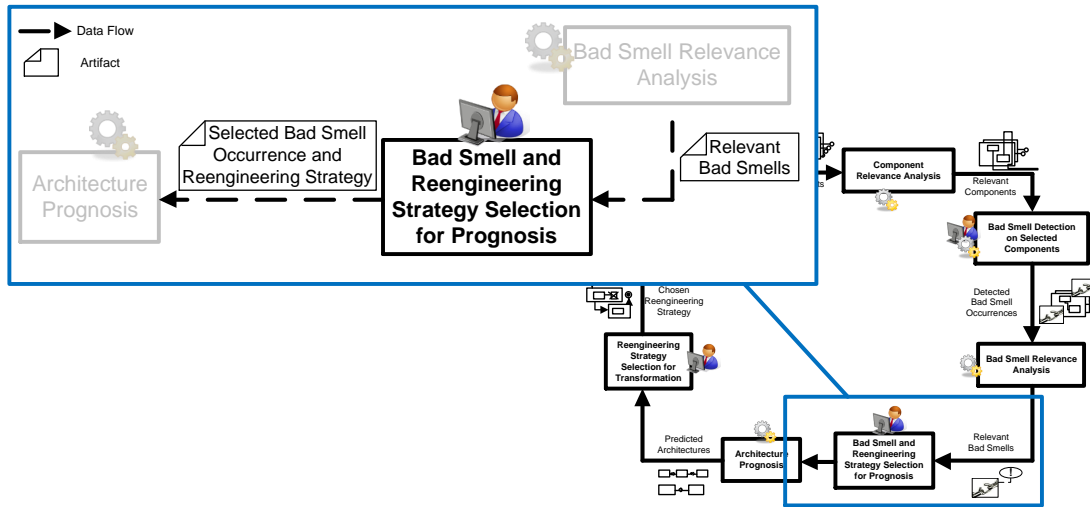


Figure 5.5: The reengineering strategy selection in the reengineering process

above, this has the consequence that other classes that implement the interface `ISearch` also have to be adapted, i.e. they have to implement the new method `searchProducer()`. The implementation of those methods is a task that has to be done manually by the reengineer. All other modifications of this strategy can be executed automatically.

A comparison of the original system to the reengineered system in form of a class structure is depicted in Figure 5.4. In part a, the original system is shown as class diagram, in part b, the system after the application of the second reengineering strategy, as described above, is shown. In contrast to the original system, in the new class structure the class `ProductsListView` has no longer a reference to the concrete class `ProductSearch`. Instead, the interface `ISearch` has been extended.

After the reengineer selected a bad smell occurrence and a reengineering strategy to accomplish this, the architecture prognosis can be started. The relevant process extract is shown in Figure 5.5.

Note that there also might be cases in which the removal of a bad smell cannot be accomplished fully automatically. Then the reengineer has to intervene and to remove a bad smell partly or completely by himself.



# 6 Architecture Prognosis

The following chapter details on the concept for the architecture prognosis. First, the idea for an architecture prognosis is motivated. Then, the comparison criteria and the actual prognosis calculation are explained.

## 6.1 Motivation

To support the decision which reengineering strategy should be applied to her system, the reengineer has to find out which strategy meets his requirements best. For this, one important decision criterion is the consequence of the application of a strategy to the system under analysis.

For example, the two reengineering strategies for interface violation occurrences, as presented in Chapter 5, have different consequences. If the reengineer decides to remove the bad smell by deleting the call, the behavior of the system is changed. In contrast, if the reengineer selects the reengineering strategy that extends the interface, the behavior remains unchanged but more parts of the system have to be adapted. To make a decision to apply one of the strategies, the reengineer has to know the consequences in both cases and she should have an overview about the impact on the system's architecture.

The different reengineering strategies effect different modifications in the class structure as well as in the metric values. Because of this, the resulting component structure of the clustering after the reengineering can differ in both cases. Two possible resulting architectures for a part of the example system and the reengineering strategies presented in Chapter 5 are depicted in Figure 6.1. In the first possible architecture (a), the two classes `ProductsListView` and `ProductSearch` are merged into one component with the interfaces `IListView` and `ISearch`. This

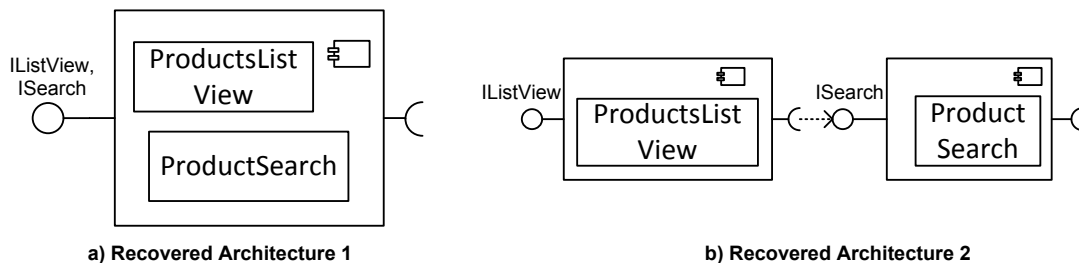


Figure 6.1: Recovered example architectures

is the same result as from the initial clustering (see Section 2.5), but one class is left out to preserve clarity. The reason for the merge in one component is that the coupling between the classes `ProductsListView` and `ProductSearch` is still tight. In the second possible architecture (b), two components are recovered: one component that contains the class `ProductsListView` and has the interface `IListView` and one component that contains the class `ProductSearch` and the interface `ISearch`. The both components are connected via the `ISearch` interface. In this case, the removal of the interface violation between `ProductsListView` and `ProductSearch` caused these classes to be loosely coupled, so that they are clustered into different components.

For the presented example situation, the first recovered architecture possibility (a) is predicted. How the architecture changes depends on how the metric values are influenced by the application of a reengineering strategy. This, in turn, depends on the strategy, as well as on the original system, i.e., the classes that are involved and the selected bad smell occurrence. In most cases, the consequences of the application of the different reengineering strategies on the system's architecture are not obvious. For this reason, I propose a prognosis, in which the consequences of a reengineering strategy are calculated and presented to the reengineer.

In this *Architecture Prognosis*, the concrete architecture that will be created by the reengineering is calculated and presented to the user. This helps the reengineer to decide, how to accomplish the removal of bad smells, so that the target architecture meets his requirements best.

For this purpose, the architecture resulting from the clustering (referred to as *original architecture*) is compared with the anticipated architecture from the prognosis (referred to as *predicted architecture*).

## 6.2 Integration in the Reengineering Process

The architecture prognosis can be started after the step in which the reengineer selects a bad smell occurrence to remove and a reengineering strategy to accomplish the removal (see Figure 6.2). In addition to the selected bad smell occurrence and the reengineering strategy, the architecture prognosis takes the current architecture model, created in the clustering, as input. It results in a predicted architecture to the bad smell occurrence and reengineering strategy tuple.

If the reengineer plans to remove several bad smell occurrences successively, she can reuse the predicted architecture for the next prognosis.

## 6.3 Comparison Criteria

First, it has to be determined which information should be included in the prognosis.

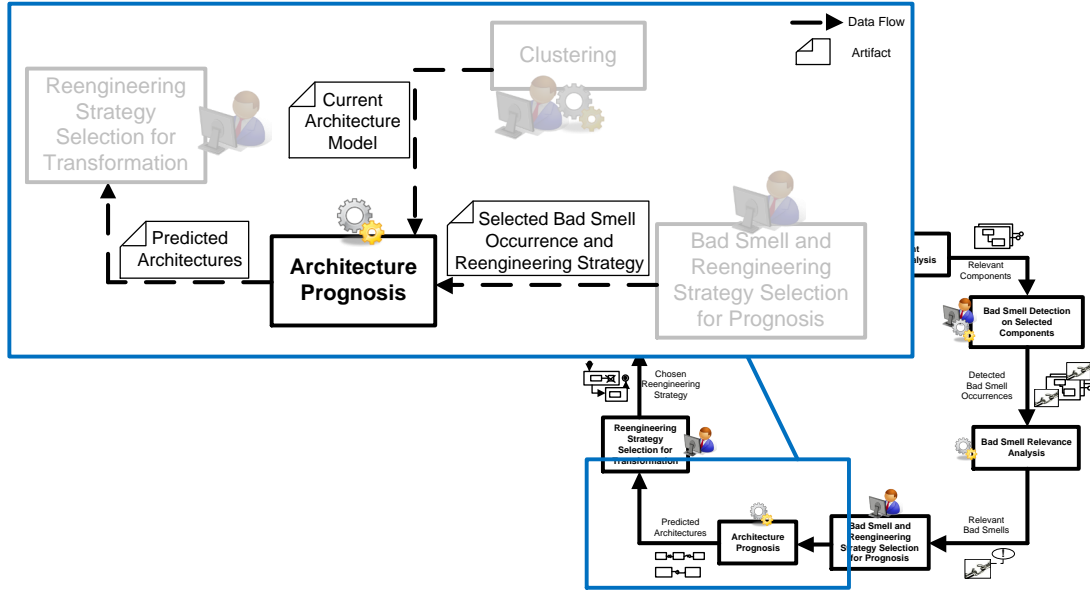


Figure 6.2: The architecture prognosis in the reengineering process

There are several levels of detail at which the two architectures can be compared. On a very abstract level, there is the comparison of the number of existing components. In addition, the number of primitive components as well as the number of composite components can be regarded. This level of detail is sufficient if only a rough overview of the changes between the two architectures is of interest to the user.

The next step is to compare the total number of interfaces and the number of messages between components. These values are needed if the communication and the collaboration between the components are of interest.

On a more detailed level, the size of the particular components and their concrete composition becomes relevant. This comprises the sub components for composite components and the implementing classes for primitive components. This data could concern users that need a more detailed view of the predicted architecture. One use case could be that further analyses have to be executed on the predicted architecture, for example to evaluate certain characteristics of single components.

Other details that could be considered are the connectors between interfaces and how the interfaces are used, i.e., the concrete sequences of messages that are sent. This information is of interest, e.g., if a subsequent behavioral analysis on the predicted architecture is intended. Also for a performance analysis this could be useful.

For the first version of an architecture prognosis in the presented reengineering process, this thesis focuses on the more abstract levels of detail. According to this, the following criteria of the original architecture and the predicted architecture

are compared:

- Total number of existing components
- Number of primitive components
- Number of composite components
- Total number of interfaces
- Total number of messages
- The size and the composition of components

## 6.4 Prognosis Calculation

To create the prognosis, a component model of the predicted architecture has to be calculated. The simplest possibility to obtain the predicted architecture is to executing a reengineering strategy on a copy of the system and executing a new clustering on the reengineered copy.

For this, the same configuration for the clustering has to be used as in the initial clustering. Using the same configuration is important because otherwise the results are not comparable because the clustering could proceed differently for the same input. For this purpose, the configuration is stored in the metric values model during the clustering which makes the configuration from the initial clustering accessible to the architecture prognosis.

According to this, the required inputs for the architecture prognosis are: The metric values model of the original architecture, a bad smell occurrence to be removed, an appropriate reengineering strategy to accomplish the removal, and component models of the original and of the predicted architecture.

In the future a more efficient way to create the predicted architecture without performing new clustering on the whole system could be investigated. But because the clustering is a complex process that includes several iterations which are based on each other, it is a difficult task to manipulate the results only for the modified part of the system.

To simplify the comparison for the user, the differences between the original architecture and the predicted architecture are highlighted. Furthermore, for classes that are assigned to another component in the predicted architecture, than before, it is displayed, where these classes were located in the original architecture.

Figure 6.3 depicts a visualization of an architecture prognosis for the running example. Part a shows the original architecture and Part b shows the predicted architecture. The original architecture in this example only consists of one component, which is here named `comp 1`. The predicted architecture consists of two components, `comp 1` and `comp 2`. Modified components are visualized with a yellow border in this figure.

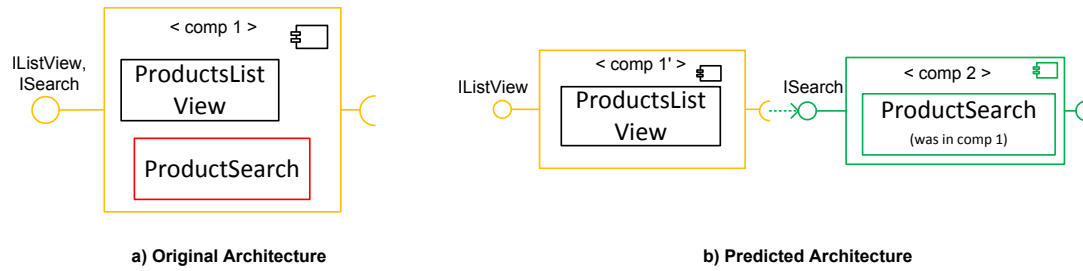


Figure 6.3: An architecture prognosis for the example system

The reason for the new component structure is that the class `ProductSearch` is assigned to another component after the reengineering. In this figure, this is marked in the original architecture by a red border. Objects that in the predicted architecture are new in comparison to the original architecture are colored green. In this case this applies to the component `comp 2`. For the class `ProductSearch` in the predicted architecture, the label “was in comp 1” indicates the former location of the class.



# 7 Realization

This chapter describes how the approach presented in the chapters 3 to 6 was realized. This includes the storage of the metric values from the clustering as they are required for the relevance analysis. Furthermore, the implementation of the relevance analysis and the architecture prognosis are explained and a short overview of the user interface is given.

## 7.1 Overview

The relevance analysis and the architecture prognosis are realized as Eclipse plug-ins. Figure 7.1 shows the components involved and the dependencies between them. Dependencies between subcomponents of a composite component are omitted for a better readability.

The components that were developed within the scope of this work are marked blue. The **Relevance Analysis** component as well as the **Architecture Prognosis** component include two plug-ins: one that contains the logical part of the realization and one plug-in for the user interface.

As depicted in the figure, the **Relevance Analysis** and the **Architecture Prognosis** require other components from **SISSy**, **SoMoX**, **Reclipse** and **Fujaba**. From **SISSy**, the **GAST Meta Model** is used. This model specifies the parsed abstract syntax tree of the system under analysis. The used subcomponents of **SoMoX** are the **SoMoX Core**, the **Source Code Decorator Meta Model** and the **Metric Values Meta Model**. The **SoMoX Core** is responsible for configuring and starting of the clustering process. The **Source Code Decorator Meta Model** specifies the correspondence of elements from the architecture model (SAMB) to the model elements from the GAST. Therefore, the **Source Code Decorator Meta Model** can be used to access both, the components and their implementing classes. The **Metric Values Meta Model** was developed within the scope of this work and is used for the storage of the metric values from the clustering. This model is explained in detail in Section 7.2.1. From the **Reclipse** component, the **Reclipse Structure Specification** subcomponent is used, as well as the **Reclipse Inference**. **Reclipse Structure Specification** contains the meta model for the pattern specification language used in **Reclipse**. Classes associated with the pattern detection are contained in the **Reclipse Inference** component. The **Fujaba** component holds the **Story Diagram Meta Model** and a **Story Diagram Interpreter** which can execute story diagrams specified with that meta model. The story diagrams are used to specify the reengineering strate-

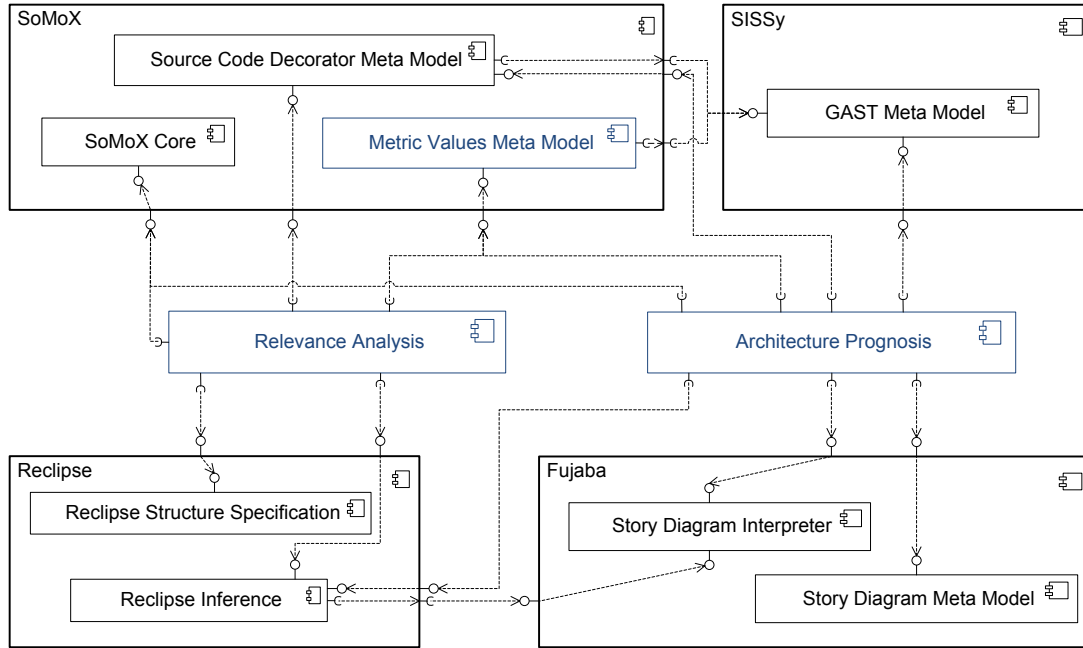


Figure 7.1: Component architecture of the developed tools and their environment

gies.

The **Relevance Analysis** uses the **SoMoX Core** because the core contains the SoMoX configuration. Furthermore, it depends on the **Source Code Decorator Meta Model** and the **Metric Values Meta Model**. The **Relevance Analysis** also has dependencies to the **Reclipse Inference** and to the **Reclipse Structure Specification** because of the required model elements. The architecture prognosis needs to start a SoMoX clustering as well as the story diagram interpreter. Because of that, the **Architecture Prognosis** component has more dependencies than the **Relevance Analysis**. It uses the **SoMoX Core**, the **Source Code Decorator Meta Model** and the **Metric Values Meta Model** from SoMoX and the **GAST Meta Model** from SISSy. Furthermore, the **Reclipse Inference** is used and the **Story Diagram Interpreter** and the **Story Diagram Meta Model** that belong to the Fujaba Tool-Suite.

## 7.2 Storage of Metric Values

The relevance analysis uses the metric values from the clustering.

They are calculated during the clustering by SoMoX and have to be stored in a way that allows further processing. An appropriate meta model was specified using Ecore [SBPM08].



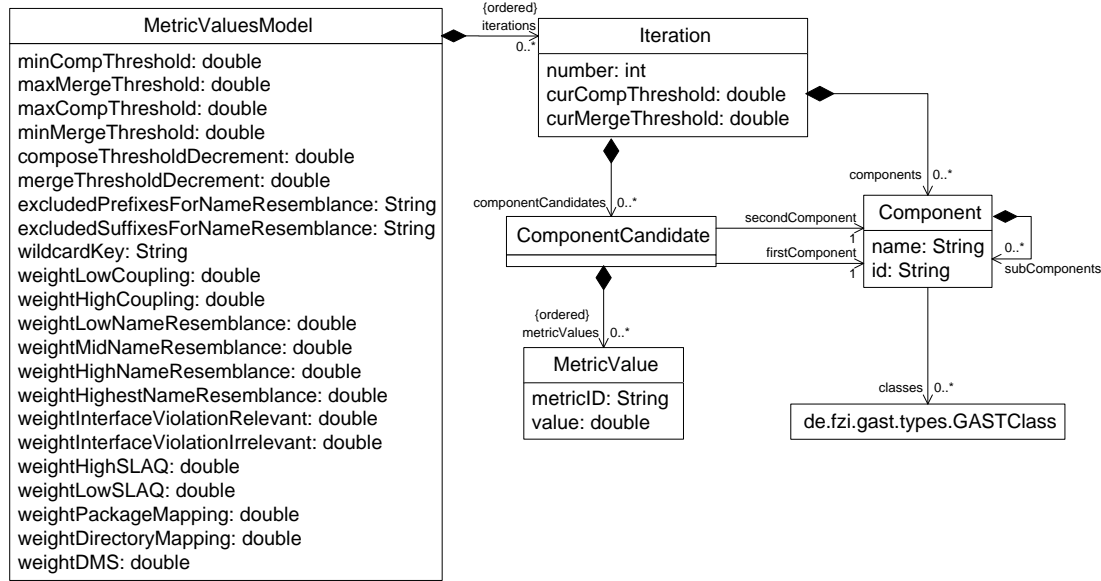


Figure 7.2: Meta model for storing the metric values

### 7.2.1 Metric Values Model

The model used to save the metric values is depicted in Figure 7.2. The root element **MetricValuesModel** contains properties of the SoMoX configuration, like the attributes **minCompThreshold** for the minimal composition threshold and **maxMergeThreshold** for the maximal merge threshold. Furthermore it stores the metric weights. A **MetricValuesModel** consists of several iterations. The element **Iteration** has a number to identify its related clustering step. In addition, it stores the composition and the merge threshold used in this iteration (**curCompThreshold**, **curMergeThreshold**) and a boolean value **isMergeIteration** that indicates if the iteration is used to merge component candidates, or to create composite components. In contrast to the other thresholds, the current composition threshold and the current merge threshold have to be stored in the iteration because they are modified during the process, as described in Section 2.3. An **Iteration** contains **componentCandidates** and **components**. A **ComponentCandidate** references two **Components**. **Components** can have arbitrarily many **subComponents**. If a **Component** has at least one sub component, it is a composite component, otherwise it is a primitive component. Furthermore, the **Component** class has a reference to **GASTClass** from the GAST meta model because components can consist of several classes. A **ComponentCandidate** has **metricValues** which are assigned to it by the clustering. The **MetricValue** element has a **metricID** and a **value** which stores the actual metric values, determined in the clustering for the component candidates.

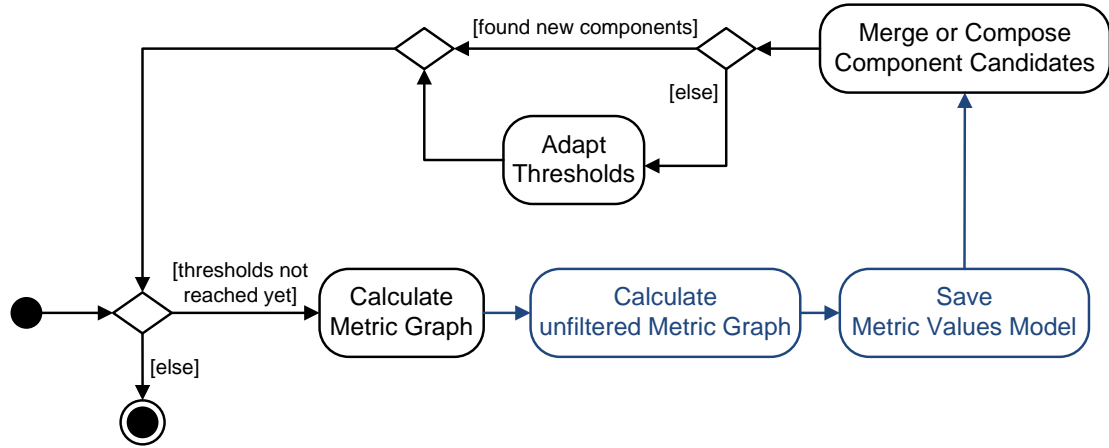


Figure 7.3: Simplified illustration of the method that is responsible for the recovery of components in the clustering

### 7.2.2 Integration of the Metric Values Model in the Clustering Process

To store the metric values computed in each iteration of the clustering process, the class from SoMoX that is responsible for performing the clustering iterations had to be modified. The method that implements the main clustering process is illustrated in the activity diagram in Figure 7.3. The blue parts of the diagram were added to save the metric values. The first step that is taken in every clustering iteration is the calculation of the metrics graph (see Chapter 2.3). There, for each component candidate, the metric values are calculated. During this step, SoMoX also filters the component candidates, so that only candidates that pass the minimum merge threshold (for merge iterations) or the minimum composition threshold (for compose iterations) will be processed further. However, for the metric values model, all component candidates have to be regarded because, e.g., those that are slightly below a threshold, are still significant in the relevance analysis. Because of this, an additional step **Calculate unfiltered Metric Graph** is added in which a graph is created, that contains the metric values for each component candidate. This graph is used in the **Save Metric Values Model** operation.

The **Save Metric Values Model** operation saves all data from the clustering process that will be required later. This includes configuration values and information about each iteration and particularly the metric values. While configuration values are only stored in the first iteration, the metric values are saved in each iteration for the current set of component candidates. The operation takes the unfiltered metrics graph as input, together with the set of current component candidates. Furthermore, the current SoMoX configuration is required, in addition to the number of the current iteration and the current merge and compose thresh-

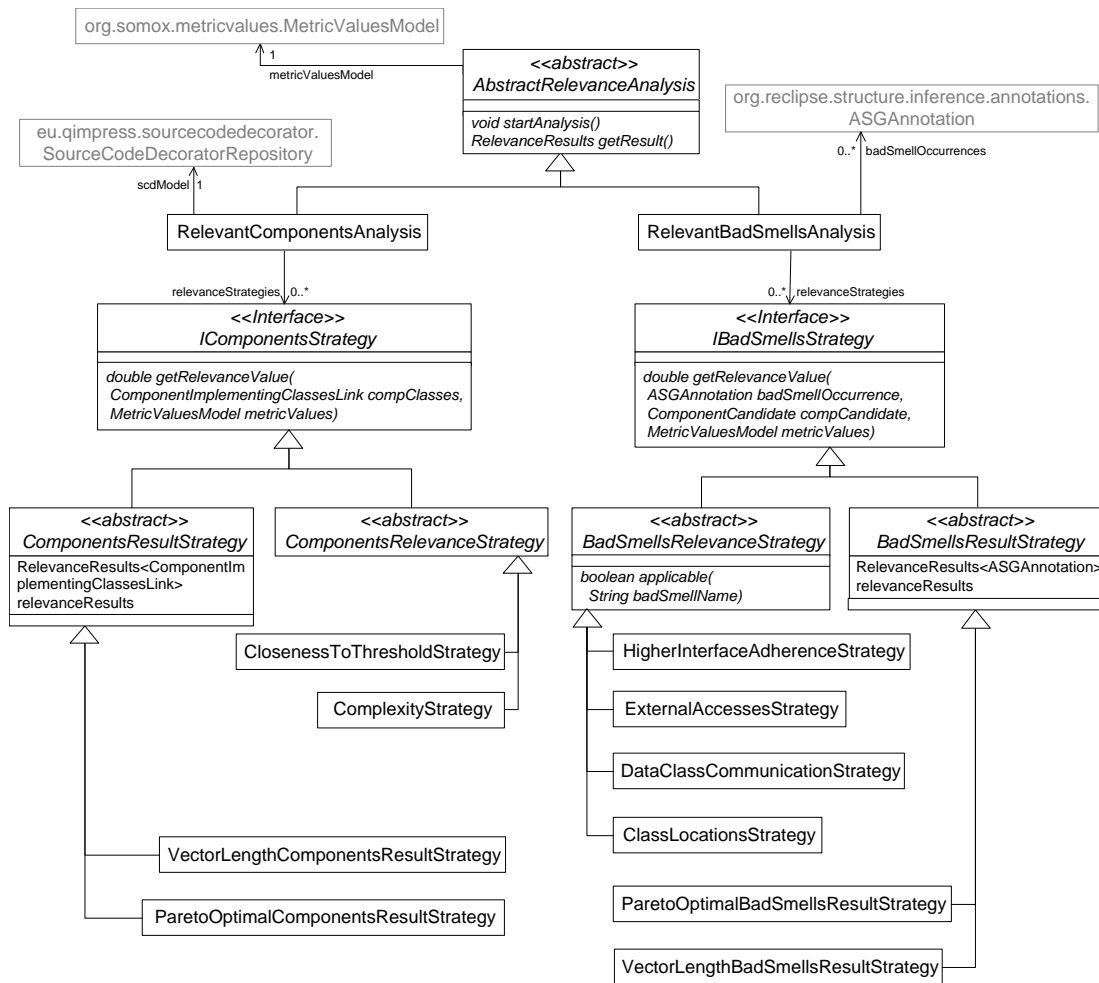


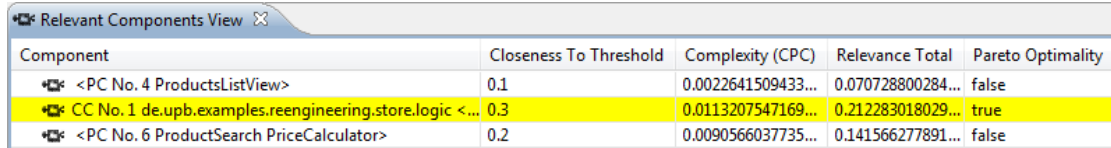
Figure 7.4: The classes used for the relevance analysis

olds. The data then is store using the Metric Values Model described in Section 7.2.1.

## 7.3 Relevance Analysis

The relevance analysis is split into the relevance analysis for components and the relevance analysis for bad smells.

Figure 7.4 shows the class structure of the relevance analysis implementation. The core is formed by the abstract class **AbstractRelevanceAnalysis** and its sub-classes **RelevantComponentsAnalysis** and **RelevantBadSmellsAnalysis**. The concrete analysis classes implement a method **startAnalysis** to start the calculation process and a method **getResult** that returns the analysis results. **AbstractRelevanceAnalysis** references the **MetricValuesModel** to access the metric values from the clustering, while **RelevantComponentsAnalysis** has a ref-



Component	Closeness To Threshold	Complexity (CPC)	Relevance Total	Pareto Optimality
<PC No. 4 ProductsListView>	0.1	0.0022641509433...	0.070728800284...	false
CC No. 1 de.upb.examples.reengineering.store.logic <...	0.3	0.0113207547169...	0.212283018029...	true
<PC No. 6 ProductSearch PriceCalculator>	0.2	0.0090566037735...	0.141566277891...	false

Figure 7.5: Relevant Components View

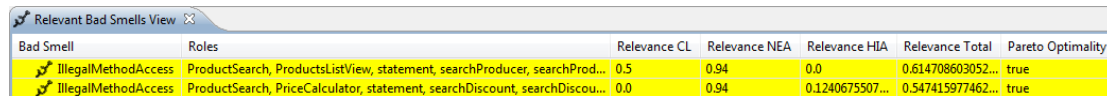
erence to the `SourceCodeDecoratorRepository`, which is the root class of the source code decorator model from SoMoX, to access the architecture created in the clustering. `RelevantBadSmellsAnalysis` references the `ASGAnnotation` class from Reclipse to access the detected bad smell occurrences. Both analysis classes hold sets of relevance strategies. To simplify the process of extending or adapting the relevance analysis, the relevance strategies are loosely coupled to the analysis algorithm by the strategy design pattern [GHJV95] with the analysis classes in the role of the contexts. Strategies belonging to the component relevance analysis implement the `IComponentsStrategy` interface and strategies for the bad smell relevance analysis implement the `IBadSmellsStrategy` interface. In both analysis parts, it is distinguished between relevance strategies and result strategies. Relevance strategies in the component relevance analysis extend the abstract class `ComponentsRelevanceStrategy`. Result strategies used in the component relevance analysis extend the abstract class `ComponentsResultStrategy`. The `ComponentsResultStrategy` has a list of maps that hold the relevance values for all component/strategy pairs. Relevance strategies in the bad smell relevance analysis extend the abstract class `BadSmellsRelevanceStrategy`. This class provides the abstract method `applicable`. This returns a boolean value that determines if a strategy is applicable for a given bad smell type. Result strategies in the bad smell relevance analysis extend the abstract class `BadSmellsResultStrategy`. It has a list of maps that hold the relevance values for the the bad smell occurrence/strategy pairs. Each strategy implements a `getRelevanceValue` method that returns a double value that represents the result for that strategy.

### 7.3.1 User Interface

The results of both relevance analyses are visualized in two views: the **Relevant Components View** and the **Relevant Bad Smells View**. Both views show the analysis results in tabular form.

Figure 7.5 shows the Relevant Components View for the store example.

Each line presents one component. The column **Component** shows the classes the component consists of. The second and third columns **Closeness To Threshold** and **Complexity (CPC)** show the values of the two relevance strategies (see 4.1.3). The **Relevance Total** column shows the normalized vector length and the column **Pareto Optimality** tells if the candidate is pareto optimal, as described in Section 4.1.4. Pareto optimal candidates are highlighted with a yellow background in the whole line. If the candidate with the highest vector length is not pareto



Bad Smell	Roles	Relevance CL	Relevance NEA	Relevance HIA	Relevance Total	Pareto Optimality
IllegalMethodAccess	ProductSearch, ProductsListView, statement, searchProducer, searchProd...	0.5	0.94	0.0	0.614708603052...	true
IllegalMethodAccess	ProductSearch, PriceCalculator, statement, searchDiscount, searchDiscou...	0.0	0.94	0.1240675507...	0.547415977462...	true

Figure 7.6: Relevant Bad Smells View

optimal, the **Relevance Total** field for this candidate is highlighted as well.

Figure 7.6 shows the Relevant Bad Smells View for interface violation occurrences of the store example. Each line presents one bad smell occurrence. The first column **Bad Smell** shows the name of the bad smell specification. The second column **Roles** shows the roles of the pattern specification and the names of the objects that play these roles in the concrete pattern candidate. The next columns show the relevance values for the different strategies that were explained in Section 4.2.3: **Relevance CL** presents the value for the relevance strategy *Class Locations*; **Relevance NEA** shows the value for the relevance strategy *NumberOfExternalAccesses*; the value for the *Higher Interface Adherence* relevance strategy is presented in the column **Relevance HIA**; **Relevance DCC** shows the value for the *Communication via Data Classes* relevance strategy. The last two columns are the same as in the Relevant Components View. They show the overall relevance and if a candidate is pareto optimal. Pareto optimal candidates are highlighted as well as the candidate with the highest overall relevance.

## 7.4 Reengineering Strategies

The reengineering strategies are specified by story diagrams, which are graphical in-place model-to-model transformations [FNTZ00, Z01].

The reengineering strategies fit to the bad smell specifications described in Section 2.4. These strategies take objects with the types of the annotated elements from the pattern specification as parameters. The object variables names in the story diagrams that accord to an element in the specification (and are bound per parameter expression because of this), have the same names as in the pattern specifications.

To add a short description that helps to clarify the intent of a reengineering strategy, **EAnnotation** objects with the key `http://reclipse.reengineering.org/strategydescription` are added to the story diagrams.

The concrete story diagrams to the strategies described in Section 5 are illustrated in the Appendix A.2.

## 7.5 Architecture Prognosis

The input that is required for the execution of the architecture prognosis is:

- The metric values model of the original architecture: This model has been

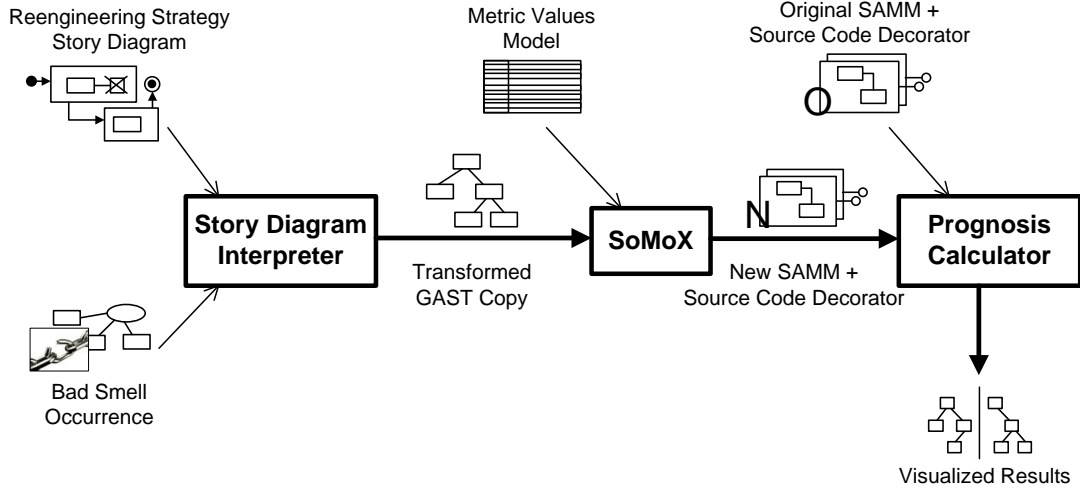


Figure 7.7: Realization of the architecture prognosis

saved in the initial clustering and is then used to start the clustering after the application of the reengineering strategy with the same parameters as the original clustering.

- The bad smell occurrence to be removed: This is selected by the user and provides references to the concrete objects from the GAST that has to be transformed. This information is used when executing the transformation.
- The selected reengineering strategy to accomplish the removal of the bad smell occurrence: This is also selected by the user. The transformation is done by executing this strategy.
- The *SAMM* of the original and of the predicted architecture: The SAMMs are required to get the data to compare both architectures. The SAMM of the predicted architecture is created during the clustering on the reengineered copy of the system.
- The *Source Code Decorator Model* of the original and the predicted architecture: Those models provide additional data to compare the particular components of the both architectures. The Source Code Decorator of the predicted architecture is created during the clustering on the reengineered copy of the system, like its SAMM.

To execute the architecture prognosis, the bad smell occurrence to remove and the reengineering strategy have to be selected by the user. Furthermore, the metric values of the initial clustering have to be provided. Other required inputs can be derived from the bad smell occurrence, provided that the SAMM and the Source Code Decorator files from one clustering run are stored in the same folder, which is the default setting in SoMoX.

To calculate the prognosis results, several steps are needed as depicted in Figure 7.7. First, the transformation has to be executed by starting the **Story Diagram Interpreter** with the story diagram that represents the chosen reengineering strategy and bad smell occurrence. The result is a transformed GAST copy that represents the reengineered system. Then, **SoMoX** has to be started to execute the clustering on this transformed GAST. During the clustering, a new SAMM and a new Source Code Decorator Model are created, which specify the new architecture model. This new architecture model as well as the original model that was input for the initial clustering run, are given to the **Prognosis Calculator**. There, the architecture prognosis results are calculated, analyzed and visualized.

### 7.5.1 Executing the Reengineering Strategy

To start the story diagram interpreter, a story diagram and objects from the host graph as context are required. The story diagram is the reengineering strategy that is selected by the user. The architecture prognosis is performed on a copy of the GAST that was an input for the initial clustering. The host graph is this copy. The objects that are given as argument are taken from the selected bad smell occurrence which is represented by an annotation from the Reclipse annotations model. From that annotation, each annotated element is transferred. With this data, the interpreter can execute the given reengineering strategy on the GAST copy. After that, the transformed copy is stored in a new Ecore resource.

### 7.5.2 Starting a Clustering with SoMoX

Typically, when starting SoMoX, the user creates a clustering configuration. By this a set of values is specified, e.g., metric weights, merge and composition thresholds, and a blacklist of files that are to be ignored in the clustering. When executing a new clustering for the architecture prognosis, it is important that the same configuration as in the initial clustering is used (see Chapter 6). The configuration from the initial clustering is restored by taking the stored configuration values from the metric values model and using them to create a new instance of the class **SoMoXConfiguration**. The only values of the configuration that are modified, are the input file, which then contains the transformed GAST instead of the original GAST, and the output folder to prevent the overwriting of the original clustering results, like the SAMM or the Source Code Decorator Model. The resulting configuration is then used when starting SoMoX.

### 7.5.3 Calculating the Prognosis Results

The calculation process for the prognosis results uses the SAMM and the Source Code Decorator Model from the initial clustering (original architecture) and from the clustering executed on the transformed GAST (predicted architecture).

	Original Architecture	Predicted Architecture
Total Number of Components	6	4
Number of Primitive Components	4	3
Number of Composite Components	2	1
Total Number of Interfaces	24	24
Total Number of Messages	29	29

Figure 7.8: The architecture prognosis view

Some of the proposed comparison criteria can be derived directly from those models. These are the total number of components, the number of primitive components, the number of composite components, the total number of interfaces and the total number of messages (see Chapter 6).

To compare the composition of the particular components, for the original architecture as well as for the predicted architecture, component trees are constructed. Those component trees represent the component structure with sub components as children of composite components and classes as children of primitive components. The differences in both architectures have to be highlighted. The assignment of components in the original architecture model and components in the new architecture model is done by comparing the component names.

The collected and calculated values are visualized to the user as a view. Details are described in the following paragraph.

#### 7.5.4 User Interface

The **Architecture Prognosis View** presents the results from the prognosis in form of a comparison between original architecture and predicted architecture. Figure 7.8 shows the comparison for an extended version of the store example (see Section 8.1). There, the prognosis is shown for the removal of an interface violation occurrence by extending the interface.

On the top part of the view, a table juxtaposes the original architecture to the predicted architecture. The lines of this table show the total number of components, the number of primitive components, the number of composite components, the number of interfaces and the number of messages for each architecture.

Below the table, the elements of the both architectures are shown by two tree viewers: the original architecture on the left, the predicted architecture on the right. In the first lines only the top level elements are shown: the components. Primitive and composite components can be distinguished by different icons and by the name of the component. For primitive components, a label between the component name shows how many classes it consist of. For composite components, the number of sub components is presented in this label. The lines can be expanded so that the implementing classes or sub components can be inspected.



To simplify the comparison of the architectures, lines that differ are highlighted with a yellow background. Components that are missing in the predicted architecture are highlighted with a red background on the original architecture side, while components that are new in the predicted architecture are highlighted with a green background on the predicted architecture side.



## 8 Evaluation

This chapter deals with the evaluation of the concept presented in Chapters 3 to 6. To validate the concept, the whole reengineering process illustrated in Chapter 3 is applied to an artificial fabricated example system and to two existing software systems: CoCoME and Palladio FileShare. For the existing software systems, first the procedure of the evaluation is described and then the results are illustrated. Section 8.4 discusses the evaluation results.

### 8.1 Store Example

For a first validation step the proposed approach is tested on a more complex version of the store example presented in Section 2.5. The system contains 8 classes containing the logic part of the system, 15 model classes and 9 classes concerned with the user interface of the system. The conceptual architecture is depicted in Figure 8.1.

Several bad smell occurrences were intentionally inserted into the system.

The clustering resulted in 5 components: 3 primitive components and 2 composite components, as depicted in the left part of Figure 8.2. In the recovered components, the model classes were all assigned to the same primitive component (PC No.60) but the classes that belong to the logic component of the system were incorrectly merged into the same component with some of the UI classes (PC No.58). The third primitive component (PC No.64) contains the remaining two classes of that belong to the UI part. The used clustering configuration and a detailed list of clustering results are shown in the appendix (Section B.1).

Figure 8.2 also shows the results of the component relevance analysis.

The relevance ratings suggest a composite component for the bad smell detection. The suggested component is also the largest component in the system. It contains all the bad smell occurrences named above.

Ten interface violation occurrences and two non-transfer object communication occurrences were detected in this system. Figure 8.3 depicts these occurrences in diagram that contains the involved classes and in a list.

Figure 8.3 also shows the overall relevance values for each bad smell occurrence. The bad smell relevance analysis identified the interface violation occurrences between `PriceCalculator` and `ProductSearch` and between `ProductSearch` and `ProductsListView` as pareto optimal. The occurrence between `ProductSearch` and `ProductsListView` is the only one between two conceptual components and consequently it correctly received the highest relevance value among all occur-

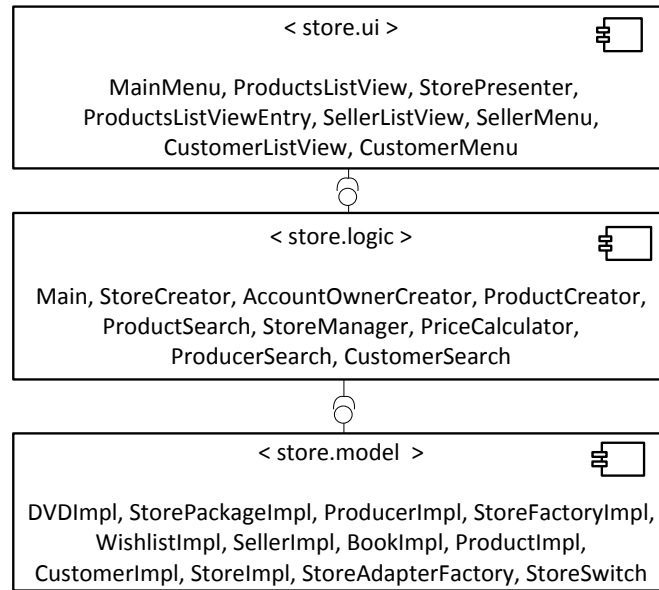
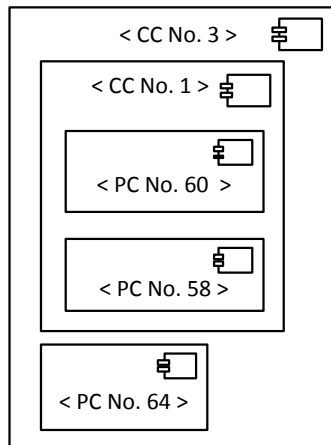


Figure 8.1: Conceptual architecture of the extended store example



Component	CTT	Complexity	Total Relevance	Pareto-Optimal
CC No. 3	0,255	0,336	<b>0,298</b>	<b>true</b>
CC No. 1	0,24	0,327	0,287	false
PC No. 58	0,154	0,072	0,121	false
PC No. 60	0,085	0,254	0,19	false
PC No. 64	0,015	0,011	0,013	false

Figure 8.2: Detected component structure and component relevance ratings

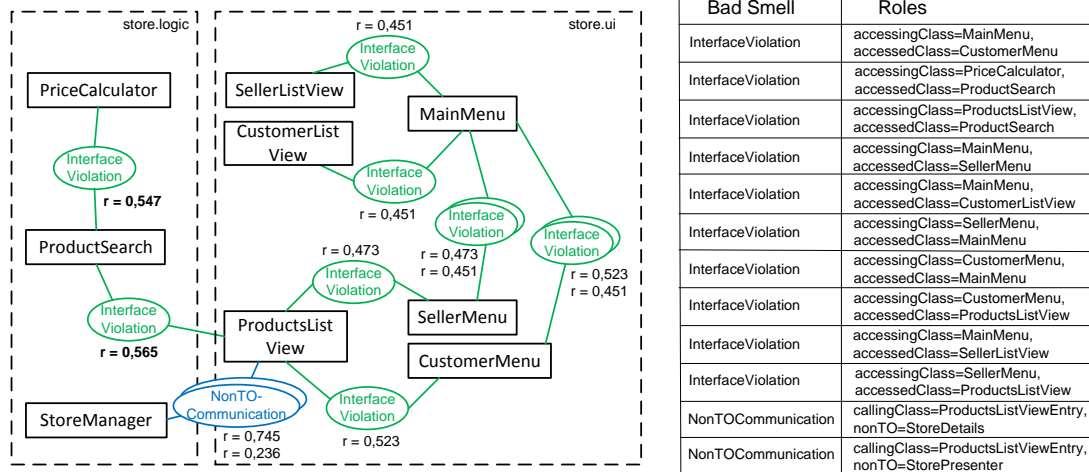


Figure 8.3: The bad smell occurrences in the store example

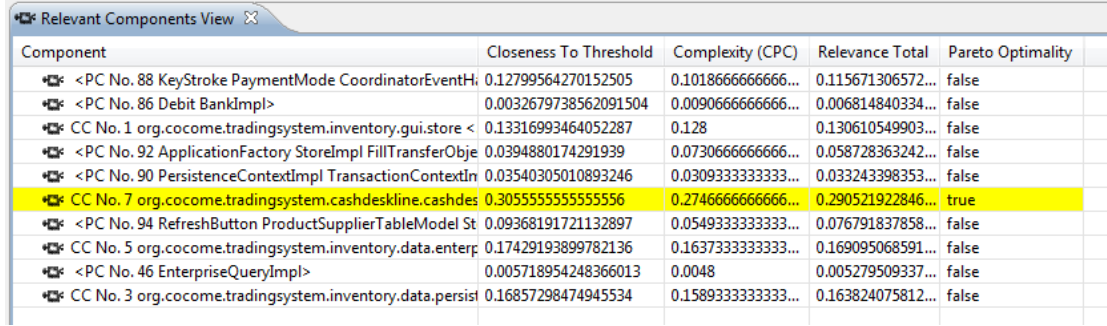
rences. Among the non-transfer object communication occurrences, one achieved a very high rating because of the correctly identified data class *StoreDetails*, while the other one received a very low rating.

The Architecture Prognosis showed that the removal of two of the interface violations (the occurrence between *MainMenu* and *CustomerMenu* with a relevance value of 0,523 and the occurrence between *ProductsListView* and *CustomerMenu* with a relevance value of 0,523) would lead to an architecture that consists of only one composite component instead of two. The removal of the other interface violations would lead to an architecture that is equal to the original architecture. The interface violation occurrences for whose removal a modified architecture was predicted are not pareto optimal with respect to their relevance. But nevertheless, they achieved a high rating compared to most of the other interface violations and they are the most relevance occurrences within the UI component. The application of each of the two reengineering strategies described in Chapter 5 lead to the same results for the removal of each interface violation.

Even after all interface violations have been remove automatically and the non-transfer object communication occurrences has been removed manually, the predicted architecture did not result in the conceptual architecture, where logic classes and user interface classes are separated. This issue is discussed later in this chapter (Section 8.4).

## 8.2 CoCoME

To further validate the approach this thesis presents, it is applied to the reference implementation of the Common Component Modeling Example CoCoME [HKW<sup>+</sup>08]. CoCoME represents a trading system. Its architecture is component-



Component	Closeness To Threshold	Complexity (CPC)	Relevance Total	Pareto Optimality
<PC No. 88 KeyStroke PaymentMode CoordinatorEventHi>	0.12799564270152505	0.10186666666666...	0.115671306572...	false
<PC No. 86 Debit BankImpl>	0.0032679738562091504	0.00906666666666...	0.006814840334...	false
CC No. 1 org.cocome.tradingsystem.inventory.gui.store <	0.13316993464052287	0.128	0.130610549903...	false
<PC No. 92 ApplicationFactory StoreImpl FillTransferObje	0.0394880174291939	0.07306666666666...	0.058728363242...	false
<PC No. 90 PersistenceContextImpl TransactionContextIn	0.03540305010893246	0.03093333333333...	0.033243398353...	false
CC No. 7 org.cocome.tradingsystem.cashdeskline.cashdes	0.3055555555555556	0.27466666666666...	0.290521922846...	true
<PC No. 94 RefreshButton ProductSupplierTableModel St	0.09368191721132897	0.05493333333333...	0.076791837858...	false
CC No. 5 org.cocome.tradingsystem.inventory.data.enterp	0.17429193899782136	0.16373333333333...	0.169095068591...	false
<PC No. 46 EnterpriseQueryImpl>	0.005718954248366013	0.0048	0.005279509337...	false
CC No. 3 org.cocome.tradingsystem.inventory.data.persis	0.16857298474945534	0.15893333333333...	0.163824075812...	false

Figure 8.4: The components from the clustering on CoCoME and their relevance

based and is intended to illustrate good component-oriented design. Another reason to choose CoCoME as an example software system in this thesis is that its conceptual architecture is well-documented. Furthermore, a reference implementation exists which was created manually and contains several design deficiencies [vDB11]. It consists of 127 classes with over 5000 lines of code. CoCoME has also been used to gain practical experiences with the bad smell detection [Tra11] and as case study for the clustering [Kro10].

### 8.2.1 Procedure

The application of the approach on CoCoME consists of the steps from the proposed process as presented in Chapter 3:

1. Initial clustering
2. Component relevance analysis
3. Bad smell detection
4. Bad smell relevance analysis
5. Selection of bad smell occurrence and reengineering strategy
6. Architecture Prognosis

Table 8.1 depicts the configuration used for the clustering.

### 8.2.2 Results

1. Initial clustering:

The initial clustering performed with the metric values from Table 8.1 results in a component structure of 6 primitive components and 4 composite components. The precise assignment of the classes to the components is listed in the appendix (Section B.2). The clustering was done in 16 iterations.

Metric	Weight
Package Mapping	60
Directory Mapping	0
DMS	5
Low Coupling	0
High Coupling	15
Low Name Resemblance	5
Mid Name Resemblance	15
High Name Resemblance	30
Highest Name Resemblance	45
Low SLAQ	0
High SLAQ	15
Composition: Interface Adherence	40
Clustering Composition Threshold Max Value	100
Clustering Composition Threshold Min Value	25
Clustering Composition Threshold Decrement	10
Merge: Interface Violation	10
Clustering Merge Threshold Max Value	100
Clustering Merge Threshold Min Value	45
Clustering Merge Threshold Increment	10
Blacklist	everything but org.cocome.*
Additional filter	.*TO .*Event

Table 8.1: Configuration used for the clustering on CoCoME

## 2. Component relevance analysis:

Figure 8.4 shows the relevance rating of the components from CoCoME. The most relevant component by far is the composite component **CC No.7**. It dominates regarding the value from the complexity strategy ( $\approx 0,2747$ ) and regarding the value from the Closeness To Threshold strategy ( $\approx 0.306$ ). The relevance of a component is not equivalent to the number of bad smell occurrences contained in that component. The component relevance analysis evaluates more than that (see Closeness To Threshold strategy, Section 4.1.3). Nevertheless, to evaluate this aspect of the component relevance analysis, a bad smell detection was carried out on each of the components and the number of detected bad smells per component was compared to the relevance hierarchy. Table 8.2 shows the results. Each line represents the values for a given component. The second and third column show the number of occurrences for the bad smells Interface Violation and Communication via Non-Transfer-Objects that were detected within the selected component. The column **Relevance** shows the rating for the overall relevance indicated

Selected Components	Bad Smells		Component Relevance Ratings					
	Interface Violation	Non-TO-Comm.	Overall Relevance	Overall Relevance (rank)	CTT	CTT (rank)	Compl.	Compl. (rank)
PC No. 46	2	0	0,0053	<b>10</b>	0,0057	<b>9</b>	0,0048	<b>10</b>
PC No. 86	0	0	0,0068	<b>9</b>	0,0033	<b>10</b>	0,0091	<b>9</b>
PC No. 88	0	0	0,1157	<b>5</b>	0,128	<b>5</b>	0,1019	<b>5</b>
PC No. 90	9	0	0,0332	<b>8</b>	0,0354	<b>8</b>	0,0309	<b>8</b>
PC No. 92	0	0	0,0587	<b>7</b>	0,0395	<b>7</b>	0,0731	<b>6</b>
PC No. 94	0	0	0,0768	<b>6</b>	0,0937	<b>6</b>	0,0549	<b>7</b>
CC No. 1	0	0	0,1306	<b>4</b>	0,1332	<b>4</b>	0,128	<b>4</b>
CC No. 3	9	0	0,1638	<b>3</b>	0,1686	<b>3</b>	0,1589	<b>3</b>
CC No. 5	11	0	0,1691	<b>2</b>	0,174	<b>2</b>	0,1637	<b>2</b>
CC No. 7	2	0	0,2905	<b>1</b>	0,3056	<b>1</b>	0,2747	<b>1</b>
All	13	31						
All PCs	13	0						
All CCs	11	21						

Table 8.2: The components detected in CoCoME, the detected bad smells per component and relevance ratings

by the length of the vector from the origin, as described in Section 4.1.4, while the columns **CTT** and **Compl.** show the values for the two relevance strategies. The **(rank)** columns show a ranking of the components regarding the relevance, e.g. the component with rank 1 is the most relevant and the component with rank 10 is the one with the lowest relevance value.

As depicted in the table, the components in which bad smells are detected are PC No.46, PC No.90, CC No.3, CC No.5 and CC No.7. All relevance values correctly identified the three composite components (No.3, No.5, and No.7) as relevant for the bad smells search: they got the first three ranks in the overall relevance value as well in both relevance strategies. However, the two primitive components No.46 and No.90 got a very low rating, although the search within them revealed bad smell occurrences.

As a second test, detection runs on all components together, on all primitive components together, and on all composite components together were executed. The results are depicted in the three rows at the bottom of the table. It is noticeable, that the Non-Transfer Object occurrences were only detected when searching in more than one component. Section 8.4 discusses this issue in detail.

To perform a comprehensive evaluation of the next process steps independently from the results of the component relevance analysis, I selected the set of all components to be the input for the bad smell detection in the next step.

### 3. Bad smell detection:

The results from the bad smell detection are depicted in Figure 8.5. 13



Annotation	Rating	Annotated Elements
DirectComposition (4 annotations)		
IllegalMethodAccess (11 annotations)		
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,23%	implementedInterface=PersistenceContext, accessingClass=EnterpriseQueryImpl, functionAccess=getEntityManag...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,23%	implementedInterface=PersistenceContext, accessingClass=EnterpriseQueryImpl, functionAccess=getEntityManag...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccess	98,95%	implementedInterface=PersistenceContext, accessingClass=StoreQueryImpl, functionAccess=getEntityManager, ac...
IllegalMethodAccessBetweenComponents (2 annotations)		
IllegalVariableAccessBetweenComponents (3 annotations)		
IndirectComponentClasses (4 annotations)		
IndirectComposition (3 annotations)		
NonTOCommunication (31 annotations)		

Figure 8.5: Detected bad smells in the selected component of CoCoME

Bad Smell	Roles	Relevance CL	Relevance NEA	Relevance HIA	Relevance Total	Pareto Opt.
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, EnterpriseQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.92	0.0	0.556187214602...	true
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, EnterpriseQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.92	0.0	0.556187214602...	true
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccess	PersistenceContextImpl, StoreQueryImpl, statement, getEntityManager, getEntityManager, ...	0.285714285...	0.64	0.0	0.404653206693...	false
IllegalMethodAccessBetweenComp	EnterpriseQueryImpl, PersistenceContextImpl, getEntityManager, getMeanTimeToDelivery, ...	0.285714285...	0.92	0.0	0.556187214602...	true
IllegalMethodAccessBetweenComp	EnterpriseQueryImpl, PersistenceContextImpl, getEntityManager, queryEnterpriseById, getE...	0.285714285...	0.92	0.0	0.556187214602...	true

Figure 8.6: Interface violation occurrences in CoCoME, rated by their relevance

occurrences of the bad smell Interface Violation were found: 11 times the `IllegalMethodAccess` pattern and two times the `IllegalMethodAccessBetweenComponents` pattern. It is notable that all `IllegalMethodAccess` occurrences concern the interface `PersistenceContext` and a method named `getEntityManager` which is located in the class `PersistenceContextImpl`. The accessing class is either `StoreQueryImpl` or `EnterpriseQueryImpl`.

Furthermore, 31 occurrences of `NonTOCommunication` were detected. After manually inspecting the results, I identified 15 of these candidates as false positives because the called class in these cases was a class named `FillTransferObjects`. This class is used to create transfer objects and hence, passing non-transfer objects can be tolerated in this case. Consequently, this class should probably have been excluded from the clustering. Eight other `NonTOCommunication` occurrences among the 31 candidates were related to classes with the suffix `Event` as non-transfer object. These can also be viewed as false positives because in CoCoME the event classes are not part of the architecture. To filter such cases, the used bad smell specification should be adapted.

#### 4. Bad smell relevance analysis:

Figure 8.6 depicts the results of the bad smell relevance analysis for the

Bad Smell	Roles	Relevance CL	Relevance DCC	Relevance Total	Pareto Opt...
NonTOCommunication	GUIEventHandlerImpl, onInvalidCreditCard, InvalidCreditCardEvent, CashDeskGUI, onInvalidCreditCard,	0.0	1.0	0.707106781186...	true
NonTOCommunication	StoreImpl, fillProductWithStockItemTO, StockItem, FillTransferObjects, fillProductWithStockItemTO,	0.0	0.0	0.0	false
NonTOCommunication	GUIEventHandlerImpl, onExpressModeDisabledEvent, ExpressModeDisabledEvent, CashDeskGUI, onExpressMod...	0.0	1.0	0.707106781186...	true
NonTOCommunication	Store, addElem, Refreshable, RefreshButton, addElem,	0.0	1.0	0.707106781186...	true
NonTOCommunication	StoreImpl, fillProductTO, Product, FillTransferObjects, fillProductTO,	0.0	0.0	0.0	false
NonTOCommunication	GUIEventHandlerImpl, onStarted, SaleStartedEvent, CashDeskGUI, onStarted,	0.0	1.0	0.707106781186...	true
NonTOCommunication	StoreImpl, fillStoreWithEnterpriseTO, Store, FillTransferObjects, fillStoreWithEnterpriseTO,	0.0	0.0	0.0	false
NonTOCommunication	PrinterControllerEventHandlerImpl, updatePrintout, ChangeAmountCalculatedEvent, PrinterController, updateP...	0.0	1.0	0.707106781186...	true
NonTOCommunication	StoreImpl, fillProductWithSupplierAndStockItemTO, Product, FillTransferObjects, fillProductWithSupplierAndSt...	0.0	0.0	0.0	false
NonTOCommunication	LightDisplayControllerEventHandlerImpl, update, ExpressModeEnabledEvent, LightDisplayController, update,	0.0	1.0	0.707106781186...	true
NonTOCommunication	StoreImpl, fillProductWithSupplierAndStockItemTO, PersistenceContext, FillTransferObjects, fillProductWithSup...	0.0	1.0	0.707106781186...	true
NonTOCommunication	StoreImpl, fillProductWithSupplierTO, Product, FillTransferObjects, fillProductWithSupplierTO,	0.0	0.0	0.0	false
NonTOCommunication	StoreImpl, fillProductWithStockItemTO, StockItem, FillTransferObjects, fillProductWithStockItemTO,	0.0	0.0	0.0	false
NonTOCommunication	AmplStarter, fillProductTO, Product, FillTransferObjects, fillProductTO,	0.285714285...	0.0	0.202030508910...	true
NonTOCommunication	StoreImpl, fillEnterpriseTO, TradingEnterprise, FillTransferObjects, fillEnterpriseTO,	0.0	0.0	0.0	false
NonTOCommunication	StoreImpl, fillProductWithSupplierAndStockItemTO, StoreQueryIf, FillTransferObjects, fillProductWithSupplierA...	0.0	1.0	0.707106781186...	true
NonTOCommunication	AmplStarter, fillStoreTO, Store, FillTransferObjects, fillStoreTO,	0.285714285...	0.0	0.202030508910...	true
NonTOCommunication	StoreImpl, fillStoreTO, Store, FillTransferObjects, fillStoreTO,	0.0	0.0	0.0	false
NonTOCommunication	Store, addElem, Refreshable, RefreshButton, addElem,	0.0	1.0	0.707106781186...	true
NonTOCommunication	Store, addElem, Refreshable, RefreshButton, addElem,	0.0	1.0	0.707106781186...	true
NonTOCommunication	GUIEventHandlerImpl, onCreditCardScanFailed, CreditCardScanFailedEvent, CashDeskGUI, onCreditCardScanFai...	0.0	1.0	0.707106781186...	true
NonTOCommunication	AmplStarter, fillProductTO, Product, FillTransferObjects, fillProductTO,	0.285714285...	0.0	0.202030508910...	true
NonTOCommunication	GUIEventHandlerImpl, onFinished, SaleSuccessEvent, CashDeskGUI, onFinished,	0.0	1.0	0.707106781186...	true
NonTOCommunication	StoreImpl, fillComplexOrderTO, ProductOrder, FillTransferObjects, fillComplexOrderTO,	0.0	0.0	0.0	false
NonTOCommunication	StoreImpl, fillProductWithStockItemTO, StockItem, FillTransferObjects, fillProductWithStockItemTO,	0.0	0.0	0.0	false
NonTOCommunication	PrinterControllerEventHandlerImpl, updatePrintout, RunningTotalChangedEvent, PrinterController, updatePrint...	0.0	1.0	0.707106781186...	true
NonTOCommunication	Store, addElem, Refreshable, RefreshButton, addElem,	0.0	1.0	0.707106781186...	true
NonTOCommunication	StoreImpl, fillComplexOrderTO, ProductOrder, FillTransferObjects, fillComplexOrderTO,	0.0	0.0	0.0	false
NonTOCommunication	GUIEventHandlerImpl, onExpressModeEnabledEvent, ExpressModeEnabledEvent, CashDeskGUI, onExpressMode...	0.0	1.0	0.707106781186...	true
NonTOCommunication	Store, addElem, Refreshable, RefreshButton, addElem,	0.0	1.0	0.707106781186...	true
NonTOCommunication	LightDisplayControllerEventHandlerImpl, update, ExpressModeDisabledEvent, LightDisplayController, update,	0.0	1.0	0.707106781186...	true

Figure 8.7: Communication via non-transfer object occurrences in CoCoME, rated by their relevance

detected interface violations. The `IllegalMethodAccess` occurrences in the class `EnterpriseQueryImpl` are rated higher than the ones in `StoreQueryImpl`. The difference is due to the result for the relevance strategy *Number of External Accesses*.

The ratings for the different bad smell occurrences are all very similar because in most cases the same classes are involved. The impact of such situations is discussed in Section 8.4.

Figure 8.7 shows the ratings of the relevance analysis for the detected Communication via Non-Transfer Object occurrences. Most of occurrences related to the class `FillTransferObjects` are rated as not very relevant. This result corresponds to my observations that these candidates are false positives and should be ignored as pointed out above. To conclude, in such situations, the rating received from the bad smell relevance analysis seems to be useful to the reengineer.

##### 5. Selection of bad smell occurrence and reengineering strategy:

I selected one of the two most relevant `IllegalMethodAccess` occurrences to be removed. The method inside which the interface violation takes place is `getMeanTimeToDelivery`, in the class `EnterpriseQueryImpl`.

To accomplish the removal, the two reengineering strategies illustrated in Chapter 5 were proposed, as shown in the screenshot in Figure 8.8.

I decided to execute the architecture prognosis for the application of the strategy that extends the interface first for two reasons: 1. I did not want

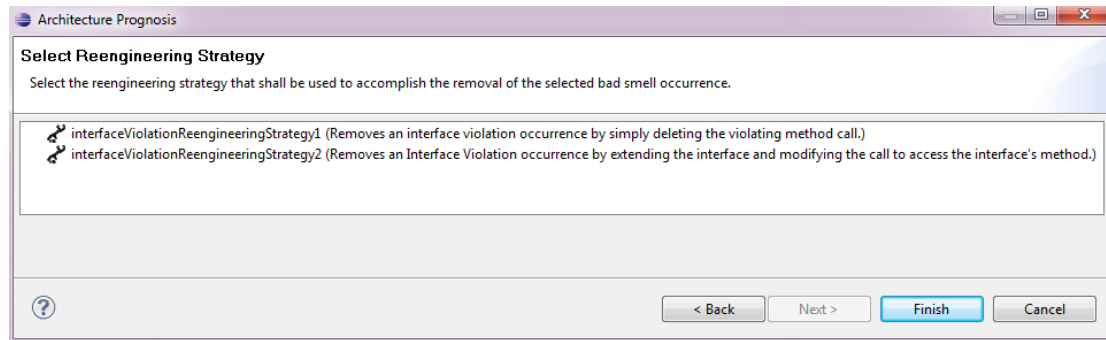


Figure 8.8: The reengineering strategies selection page from the Architecture Prognosis Wizard

to loose a part of the system’s behavior by deleting a method call. 2. From the results of the Bad Smell Relevance Analysis, I knew that there were several relevant interface violation occurrences that concern the same interface as the occurrence that I wanted to remove. Because of this, it seemed worthwhile to extend the interface, in order to improve the whole system’s quality.

#### 6. Architecture prognosis:

Figure 8.9 shows a screenshot of the Architecture Prognosis View for the removal of the selected bad smell occurrence. As depicted there, the architecture created in the original clustering consists of 11 components: 7 primitive components and 4 composite components. In contrast, the predicted architecture consists of only 10 components: 7 primitive components and 3 composite components. The component trees show that the composite component CC No.5 is missing in the predicted architecture. In addition, the component CC No.3 changed: in the predicted architecture, it contains one component more than in the original architecture.

Figure 8.10 shows an abstract illustration of the component structure created in the initial clustering and the predicted component structure for the selected combination of a bad smell occurrence and reengineering strategy. The notation is similar to UML component diagrams but additionally for the interesting components, the contained classes are visualized and interfaces and connectors are left out due to readability reasons. In addition to the component name that was given by the clustering, a second label shows the name of the corresponding conceptual component as documented.

In the original architecture, the component `inventory.data` is fragmented into the primitive components PC No.90 which is located in the composite component CC No.3 and PC No.46 which is located in CC No.5. CC No.5 also contains CC No.3. In contrast, in the predicted architecture, the primitive components PC No.90 and PC No.46 that make up the data component,



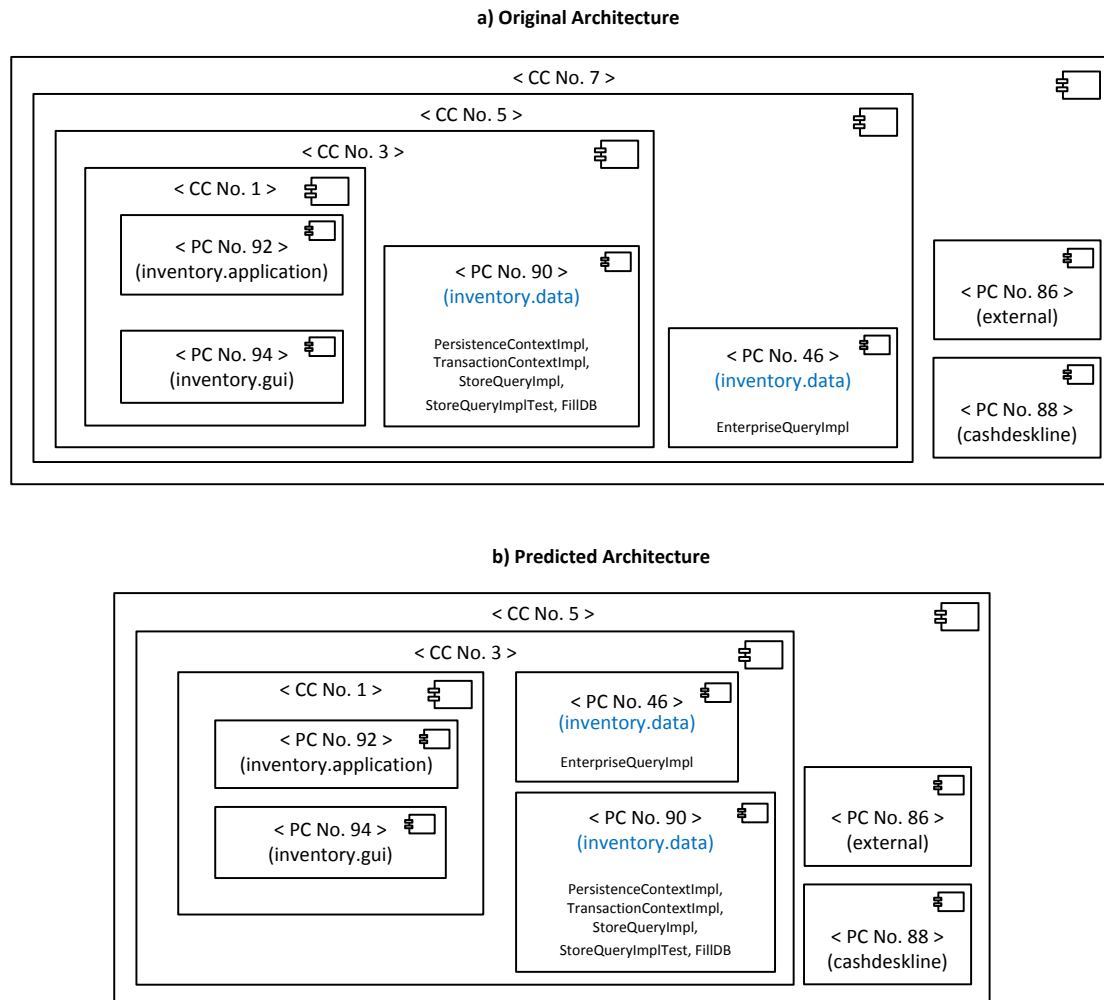


Figure 8.10: The original and a predicted components in CoCoME

are both assigned to CC No.3.

To conclude, the architecture after the removal of the selected bad smell occurrence is closer to the conceptual architecture than before, which supports the assumptions of this thesis. However, the predicted architecture still differs from the conceptual architecture, e.g. the conceptual component `inventory.data` still consists of two parts. After performing several iterations of the reengineering process, in which all other interface violation occurrences were removed one after another, the predicted architecture did not change again.

In a next evaluation step, I executed the architecture prognosis for one of the less relevant bad smell occurrences: The method containing the interface violation is named `queryStoreById` and located in the class `StoreQueryImpl`. Again I chose the reengineering strategy that extends the interface. This time the predicted architecture remained equal to the original architecture. This means that the bad smell relevance analysis correctly identified bad smell occurrences whose removal lead to an architecture that is closer than the conceptual architecture than the original architecture, while bad smell occurrences whose removal did not change the architecture received a lower rating. To conclude, the bad smell relevance analysis was an actual support to the reengineer in this situation.

## 8.3 Palladio FileShare

Palladio FileShare realizes a server-based file sharing platform. It is written in Java and represents a typical business information system. The system's architecture is well-documented and has already been used as case study for the clustering with SoMoX [Kro10, KKR10].

### 8.3.1 Procedure

For the application of the approach on Palladio FileShare, the performed steps used above have been slightly adapted, in order to learn more about the reasons for the analysis results:

1. Initial clustering of the original system
2. Addition of bad smells and clustering of the adapted system
3. Clustering of the adapted system
4. Component relevance analysis
5. Bad smell relevance analysis
6. Architecture prognosis

Table 8.3 depicts the configuration used for the initial clustering and the clustering of the adapted system. This configuration is a slightly adapted version from the one used by Krogmann et al. [KKR10].

Metric	Weight
Package Mapping	100
Directory Mapping	0
DMS	7
Low Coupling	0
High Coupling	10
Low Name Resemblance	5
Mid Name Resemblance	15
High Name Resemblance	40
Highest Name Resemblance	90
Low SLAQ	0
High SLAQ	25
Composition: Interface Adherence	25
Clustering Composition Threshold Max Value	80
Clustering Composition Threshold Min Value	25
Clustering Composition Threshold Decrement	15
Merge: Interface Violation	10
Clustering Merge Threshold Max Value	100
Clustering Merge Threshold Min Value	41
Clustering Merge Threshold Increment	7
Blacklist	java, de.uka.ipd.sdq, de.uka.ipd.sdq.BySuite, de.uka.ipd.sdq.palladio- fileshare.testdriver

Table 8.3: Configuration used for the Clustering on Palladio FileShare

### 8.3.2 Results

#### 1. Initial clustering of the original system:

The initial clustering performed on the original Palladio FileShare System with the metric values from Table ?? resulted in a component structure with 12 primitive components and 3 composite components. The clustering was done in 16 iterations. A sketch of the composition of the components with reference to the conceptual components is depicted in Figure 8.11. The two composite components CC No.1 and CC No.5 contain the parts that are documented as the compression and hashing components. The

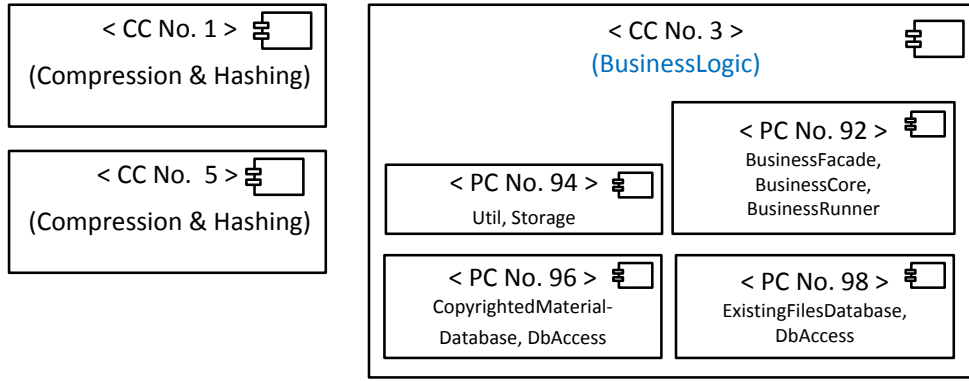


Figure 8.11: Discovered components in Palladio FileShare

figure abstracts from the detailed composition of these components, since the further evaluation focuses on the other composite component CC No.3: This component contains the business logic part of the system. It contains four primitive components.

The precise assignment of the classes to the components is listed in the appendix (Section B.3).

## 2. Addition of bad smells and clustering of the adapted system:

No interface violations that accord to the `IllegalMethodAccess` specification have been found in the original Palladio FileShare system. To investigate the detailed difference between a system that does not contain interface violations and a system that contains interface violations and to enable an evaluation of the relevance analyses, I added two interface violations to the system. Both are within the class `BusinessCore` in the business logic component and both bypass the interface `IExistingFilesDatabase`.

Nevertheless, a clustering on the adapted system resulted in the same architecture as the clustering on the original system. The reasons are discussed later.

## 3. Component relevance analysis:

Table 8.4 shows the detected bad smells and relevance ratings for the components that were discovered in Palladio FileShare. The two interface violation occurrences were detected in the primitive component PC No.92 and in the containing composite component CC No.3. This time, the two relevance strategies differ in their calculations for the components. While the strategy CTT rates the component PC No.92 with a relevance value of 0.0118 on rank 8, the **Complexity** strategy rates it with 0.027 on rank 11. The composite component CC No.3 is rated with 0.0405 on rank 4 by CTT and with 0.0642 on rank 5 by **Complexity**. To conclude, the CTT strategy returns better results for the two components, where the bad smells were detected. But on



the whole, both strategies did not correctly detect these components to be more relevant than the others.

Selected Components	Bad Smells		Relevance Ratings				
	Interface Violation	Relevance	Relevance (rank)	CTT	CTT (rank)	Compl.	Compl. (rank)
CC No. 5	0	0,234	<b>1</b>	0,2445	<b>1</b>	0,223	<b>1</b>
CC No. 1	0	0,1508	<b>2</b>	0,0127	<b>7</b>	0,2128	<b>2</b>
PC No. 86	0	0,086	<b>3</b>	0,0031	<b>14</b>	0,1216	<b>3</b>
PC No. 106	0	0,0819	<b>4</b>	0,1045	<b>2</b>	0,05	<b>8</b>
PC No. 104	0	0,063	<b>5</b>	0,0737	<b>3</b>	0,05	<b>9</b>
CC No. 3	2	0,0537	<b>6</b>	0,0405	<b>4</b>	0,0642	<b>5</b>
PC No. 102	0	0,0514	<b>7</b>	0,033	<b>6</b>	0,0649	<b>4</b>
PC No. 100	0	0,0473	<b>8</b>	0,0332	<b>5</b>	0,0581	<b>6</b>
PC No. 88	0	0,0361	<b>9</b>	0,0063	<b>12</b>	0,0507	<b>7</b>
PC No. 90	0	0,0288	<b>10</b>	0,0034	<b>13</b>	0,0405	<b>10</b>
PC No. 92	2	0,0209	<b>11</b>	0,0118	<b>8</b>	0,027	<b>11</b>
PC No. 94	0	0,0135	<b>12</b>	0,0089	<b>10</b>	0,0169	<b>12</b>
PC No. 98	0	0,0078	<b>13</b>	0,0087	<b>11</b>	0,0068	<b>13</b>
PC No. 96	0	0,0075	<b>14</b>	0,0092	<b>9</b>	0,0054	<b>14</b>
PC No. 38	0	0,0059	<b>15</b>	0,0019	<b>15</b>	0,0081	<b>15</b>

Table 8.4: The components detected in Palladio FileShare, the detected interface violations and relevance ratings

#### 4. Bad smell relevance analysis:

Both interface violation occurrences received the same rating: The strategy **Class Locations** calculated a value of 0.3333, the strategy **Number of External Accesses** resulted in a value of 0.81, and the **Higher Interface Adherence** value is 0.0.

#### 5. Architecture prognosis:

The removal of the two bad smells had only little impact on the architecture. The architecture prognosis did not show any modifications. This finding matches the results from the comparison between the original architecture and the architecture with the added bad smells, which is further discussed in Section 8.4.

## 8.4 Discussion

This section discusses the evaluation results and problematic issues that were detected during the evaluation. First, it details on the clustering results, then on both relevance analyses, after that it focuses on the architecture prognosis, and finally issues of the proposed reengineering process in general are pointed out.

Most of the points mentioned here are taken up in Section 10.2 which discusses ideas for future work.

### 8.4.1 Clustering

The result of the application of several iterations of the reengineering process to the store system and to CoCoME was that even after all interface violations were removed, the predicted architecture did not result in the conceptual architecture where logic classes and user interface classes are separated. This shows that the systems may contain some more design problems that are no interface violations or non-transfer object communication occurrences. As a consequence, some more different types of bad smells should be investigated and supported in the process so that the actual architecture can be significantly improved by the application of the proposed reengineering process. But obviously good results can only be achieved, if the original developers of the system intended to follow the conceptual architecture during their implementation.

Another interesting aspect to discuss is the strange behavior of the clustering on systems that contain bad smells in opposition to “clean” systems, as reported in the results of the application on the process on Palladio FileShare (Section 8.3). In contrast to the assumptions, the clustering of the adapted version of PalladioFileShare that contains bad smells results in the same architecture as the initial clustering. As a consequence I decided to take a deeper look at the metric values for the architecture with the two interface violations and the original architecture.

Table 8.5 lists some of the metric values for the component candidate `<BusinessFacade, BusinessCore, BusinessRunner>`, `<ExistingFilesDatabase, DbAccess>`. The metric values for the component candidate from the original system differ from the values from the adapted system in the metrics `InterfaceAdherence`, `InterfaceAccesses`, `Coupling`, `InternalAccesses`, `ExternalAccesses` and the overall merge metric. The coupling between the two components is higher (0,3684) for the system with the interface violation than for the system without (0,2941). This is in line with the assumptions made throughout this thesis.

However, this change does not have the expected impact on the final clustering results: The merge value of the component candidate without interface violations and the low coupling is slightly higher (0,3388) than for the component candidate with interface violations and with the higher coupling (0,2886). The reason is that a coupling value higher than  $\varepsilon = 0.3$  leads to the involvement of the `InterfaceAdherence` value which reduces the overall merge value (see Section 2.3.1).

<BusinessFacade, BusinessCore, BusinessRunner>, <ExistingFilesDatabase, DbAccess>								
	Interface- Adherence	Interface Accesses	Coupling	Internal Accesses	External Accesses	Package- Mapping	Merge	Compose
With IVs	<b>0,1429</b>	<b>1</b>	<b>0,3684</b>	<b>7</b>	<b>19</b>	0,6667	<b>0,2886</b>	0,3191
Without IVs	<b>0,6</b>	<b>3</b>	<b>0,2941</b>	<b>5</b>	<b>17</b>	0,6667	<b>0,3388</b>	0,3191

Table 8.5: Metric values with and without interface violations

As a consequence, in another clustering configuration, where the merge threshold reaches a value between 0,3388 and 0,2886, an architecture without design problems probably has more merged and thereby more complex components, then an architecture containing bad smells like interface violation.

Another unexpected effect that can be seen in the metric values of other component candidates is that as a consequence of changes in one class and one interface, characteristics of a class that has no visible relations to the changed class, are modified. For example the adaptations done in the business logic component of Palladio FileShare strongly influence the class `BinTree` that is located inside the compression part of the system. This shows that the behavior of the calculations of the metric values and their behavior regarding changes in parts of the system should be more deeply investigated to benefit from them in the relevance analyses or in the architecture prognosis.

### 8.4.2 Component Relevance Analysis

As the evaluation results show, the component relevance analysis offers several opportunities for improvement.

One effect that occurs most of the time in the component relevance analysis is that the largest component is rated as the most relevant in both available strategies. This happens because both strategies depend on the size of the component: The largest component in most cases is also the most complex component; and the probability that one of the contained component candidates has a merge or composition metric value close to the threshold is the higher the larger the surrounding component is. Furthermore, in many cases the components clustered with SoMoX are all contained in one composite component, as it is the case in the configuration used for the evaluation with the store system (Section 8.1) and with CoCoME (Section 8.2). As a consequence, there exists a composite component that is by far the largest component because it contains all other components. In most cases this will be the component that is rated with the highest relevance so that the user might select the component that contains the whole system as input for the bad smell detection. It is probably natural that the largest component contain the most bad smell occurrences (which is similar to the idea behind the complexity strategy, namely that the most complex component may have a high probability to contain most bad smell occurrences). But since the runtime of the bad smell detection depends on the size of the input system, i.e., the size of the component selected to contain the search scope, this result probably does not help the user because she does not want to choose the largest component. Because of this, this problem should be further investigated. Maybe composite components that contain the whole system should be ignored in the component relevance analysis or some kind of automated analysis for the trade-off of size against relevance could be done.

However, this also shows that it is not sufficient to only show the pareto optimal candidates because a composite component that contains all other components of

the system, will always be the only pareto optimal candidate. By also calculating the geometric distance, a more precise measurement that can be used to distinguish between the relevance of all components, is presented.

The other issue of the component relevance analysis that is worthy of discussion, is that not all bad smell occurrences can be detected when only searching in one component in contrast to regarding a combination of components. The component relevance analysis at the moment does not take this into account and only rates single components.

### 8.4.3 Bad Smell Relevance Analysis

In the evaluation of CoCoME, the removal of a bad smell occurrence that is rated more relevant led to a modified architecture, while the removal of a bad smell occurrence that is rated less relevant led to no architecture changes. This shows that the bad smell relevance analysis delivers useful results. With the help of these results, the reengineer is supported in her decision which bad smell occurrences to remove.

To further improve the bad smell relevance analysis, groups of bad smell occurrences could be regarded. The interface violation occurrences detected in CoCoME are all very similar because they all bypass the same interface. The bad smell relevance analysis should recognize such similar occurrences as group of bad smells. The larger a group of similar occurrences is, the more relevant is its removal.

Furthermore, it has to be taken into account that the detection of bad smells depends highly on the context in which they are searched, i.e. the surrounding project. Because of this, the specifications have to be adapted to the project under analysis. For example the bad smell **Non-Transfer Object Communication** is only automatically detectable, if transfer objects are marked with the suffix **TO**, as it is the case in CoCoME. As a consequence, to allow a more comprehensive evaluation of the bad smell relevance analysis, some more projects have to be inspected in detail.

The evaluation on CoCoME also revealed that the bad smell specifications, e.g. the specification for **Non-Transfer Object Communication**, could be optimized to gain more precise results in the bad smell detection.

### 8.4.4 Architecture Prognosis

Another conspicuity is that in most cases the removal of one bad smell occurrence is not so crucial for the overall architecture that the architecture prognosis is worthwhile. Because of this, it should be considered to remove several bad smells in the same reengineering iteration. However, this is only possible if the removal of one selected bad smell occurrence does not influence the removal of another selected bad smell occurrence or if the dependencies between several bad smell occurrences can be taken into account in the reengineering strategies to

accomplish the removal. For example, the removal of an interface violation occurrence by extending the interface influences all other interface violation occurrences that are related to the same interface. But interface violations are likely to occur repeatedly in the same context. If an interface is bypassed one time, this can easily happen again because the involved classes are probably badly designed. As pointed out above, this is the case in CoCoME. It seems sensible to remove such groups of similar bad smell occurrences at the same time. In the interface violation case, for example, once the interface has been extended for the other similar occurrences only the cast has to be removed. To allow the removal of such invalidated interface violation occurrences, I created another bad smell specification, which detects exactly those cases in which the interface has already been extended and only the cast statement has to be removed. This specification `Invalidated_IllegalMethodAccess` and an according reengineering strategy are illustrated in the Appendix A.

### 8.4.5 Reengineering Process

A problem seems to be that the whole reengineering process depends on the clustering results. To verify this, I performed the evaluation process on CoCoME as pointed out in Section 8.2 with another clustering configuration. The other configuration differs from the one documented above only in the value for the minimum merge threshold. As a result, the clustering created a component structure similar to the one described above but apparently more “stable” against modifications. Even after several reengineering iterations in which all interface violation occurrences were removed, the predicted architecture did not differ from the original architecture. To handle this problem, more investigations on the clustering configurations should be done.



## 9 Related Work

This chapter presents some work related to this thesis and compares the results to the contributions of this thesis. The first section discusses research about bad smell detection in general. The second section deals with reengineering processes. In the third section, the work that refers to the validation of the relevance of bad smells is discussed. After that, related work to the architecture prognosis is presented.

### 9.1 Bad Smell Detection

In the last decade, much research has been done on bad smells at the source code level (see [ZHB11] for an overview). Code bad smells are widely used for detecting refactoring opportunities in software [MT04].

Many papers about bad smell detection were published. For example, one bad smell detection approach has been developed lately by Moha, Gueheneuc, Duchien and Le Meur [MGDLM10]. The technique allows the specification and the detection of code and design smells. The detection algorithms are generated from the specifications and then applied automatically on design models of systems. However, the detected bad smell occurrences have to be validated manually to verify that they are true positives. The refactoring is expected to be done manually, too.

Many of the bad smell detection approaches found in literature use metrics for the detection. For example, Munro developed an approach to automatically detect bad smells in Java systems using software metrics that are calculated on the source code [Mun05]. In this approach too, the Reengineer has to manually determine the relevance of a detected bad smell occurrence.

Furthermore, many tools for bad smell refactoring exist. For example, many development environments contain support for automatic code refactoring (e.g. Eclipse or IntelliJ). In Eclipse, even a preview on code level is available for most refactorings. However, these refactorings are not always related to bad smell occurrences and a bad smell detection is not integrated into these tools.

Design deficiencies on the architectural level are not investigated as exhaustively as code bad smells. A few approaches that regard design problems of a system are taken up in the Sections 9.3 and 9.4.

## 9.2 Reengineering Processes

Tourwé and Mens [TM03] propose a refactoring process that shows certain parallels to the reengineering process presented in this thesis. They detect bad smell occurrences in an application automatically and then let the user choose between several refactoring possibilities. Subsequently the refactoring is applied automatically. These steps are also part of the reengineering process in this thesis, but there, the user is supported in making her decision for a bad smell to remove and a refactoring possibility by the relevance analysis and the architecture prognosis. In Tourwé's and Mens' refactoring process, the user has to decide without this assistance. Apparently, the integration of the bad smell detection and the reengineering in an architectural context, as my thesis proposes, provides additional possibilities that are not available for approaches like the one of Tourwé and Mens, since they only deal with code bad smells.

Tourwé and Mens also identified the problem of the long run time of a bad smell search on a whole software system, but they try to overcome this by confining the number of searched bad smells instead reducing the search scope to a only a part of the system. This could also be an idea for the reengineering process proposed in this thesis, but the pattern detection algorithm of Reclipse uses an incremental bottom-up analysis which benefits from first detecting low-level patterns which can be reused in other pattern specifications without having to search for them again. According to this, the overall runtime of the detection process when doing several searches for only subsets the available bad smells, will not be significantly reduced.

## 9.3 Validation of the Relevance of Bad Smells

Only few studies investigate the impact of bad smells [ZHB11]. Among these are the works of Kasper et al. [KG08] and Li et al. [LS07]. The results of these studies imply that some bad smells may not be design deficiencies. Some bad smells even increase the reliability of software. This supports the assumption of this thesis that a relevance analysis on bad smells is necessary before executing a reengineering.

There are already several approaches on the detection and refactoring of design problems. However, none of these approaches includes an analysis of a design problem's relevance on the system's architecture.

In a metrics-based refactoring approach, Simon et al. use the calculation of distance-based cohesion between classes, methods and attributes [SSL01]. In contrast to our approach, they start with the refactoring strategy and search application locations, instead of starting with the design problem to search suitable refactoring strategies. Bad smell candidates are selected by evaluating their relevance to the reengineer's purpose, which can be understanding, modification or quality improvement.



Marinescu added a filtering mechanism to his metric-based bad smell detection approach that determines which occurrences are relevant for further processing [Mar04]. This approach also uses the composition of several metrics to detect design deficiencies. In the filtering mechanism, detected occurrences with extreme metric values or values that are in a particular range are searched.

Trifu et al. developed an approach to correct design flaws where the influence of a flaw on specified quality factors are defined [TSG04]. But instead of evaluating the influence of a detected design flaw occurrence, they use the influence values to determine which flaws to search first. Proposals for the most problematic design flaws are made based on severity values that are derived from a selected software context. For the refactoring of the system, they suggest a set of correction strategies for each design flaw.

Bourquin and Keller presented an approach that is focused on refactorings on the architecture level [BK07]. They analyze the relevance of their refactorings on the architecture after the application. To analyze the refactoring results, they use code metrics and a comparison between the number of detected bad smells before and after the refactoring.

## 9.4 Architecture Prognosis

Little work has been done in the area of architecture prognosis for reengineered software systems.

One methodology that is similar to an architecture prognosis is a change impact analysis. Change impact analyses have been performed on the code level repeatedly (e.g. [GL91]). In contrast, Zhao et al. present an approach to support a change impact analysis of software architectures [ZYXX02]. They use slicing and chopping techniques on an architectural level to analyze the effect of changes in a software component. The process can be executed automatically and supports maintainers of the system when adapting it.

Zhao et al. do not present a concept to visualize the original and the new architecture, which would be a useful addition for this approach. But steps from Zhao's approach could be usefully integrated into the architecture prognosis presented in this thesis. Further investigations have to be done, if architectural slicing could help to avoid to execute a whole new clustering to create the predicted architecture.

Structural Analysis for Java (SA4J) [SA4] detects anti patterns and provides guidelines for refactoring. The tool only performs a structural analysis for the anti pattern detection in contrast to Reclipse, which is capable of additionally analyzing dynamic information. In SA4J, the detected anti patterns are visualized as UML diagrams. The tool is also able to execute a "what-if" analysis on the impact of a change on functionality of an application. A prognosis on the system's design is not provided.



# 10 Summary and Future Work

This chapter summarizes this thesis, draws conclusions, and presents ideas for future work.

## 10.1 Summary

Since software systems are adapted and extended over a long time, the systems' design is prone to erode. With every modification, the risk of introducing design deficiencies which decrease a software system's quality, increases. The removal of design deficiencies can be accomplished by reengineering. This thesis deals with problems that occur during the reengineering of component-based software systems. It is based on a process that contains a clustering step which extracts a component structure from a system's source code and a subsequent bad smell detection to recognize design problems.

The first problem occurs before executing the bad smell detection. Since a bad smell detection suffers from a long run-time and an impractically large result set, the search scope has to be narrowed down. As a consequence, it is proposed to perform the bad smell detection on a subset of all components in the system. The problem is that then this subset has to be wisely selected. This saves extra work and time that comes with the need to execute several bad smell detection runs on the system. The problem is solved by automatically analyzing the clustered components of the system with respect to their relevance for a bad smell detection. This *component relevance analysis* currently consists of two rating strategies: one that evaluates the complexity of a component and one that detects indications for uncertain decisions made in the clustering.

After performing the bad smell detection, the reengineer gets a set of detected bad smell occurrences. Since not all bad smell occurrences are problematic design deficiencies, she has analyze which of them should be removed in order to improve the system's quality. To accomplish this, an automatic *bad smell relevance analysis* has been presented. This analysis rates which bad smell occurrences are problematic and should be removed and which occurrences are tolerable in the context of the system. This is currently done by four strategies. A set of applicable relevance strategies exist for each bad smell. For example for the bad smell *interface violation*, three applicable strategies have been presented: one evaluates the locations of the involved classes, one regards the classes' external accesses, and one calculates the changes on the interface adherence.

The question that comes next is how the removal of a bad smell occurrence can

be accomplished. Typically, several possibilities to remove a bad smell exist and the reengineer has to select one of them. To make an informed decision, it is helpful to know the consequences that the application of a given reengineering strategy has. Because of this, an architecture prognosis is proposed which compares the current architecture with the predicted architecture that results from the removal of the selected combination of bad smell occurrence and reengineering strategy.

The evaluation showed that the reengineer is supported in the decisions she has to make during the reengineering process. But many issues remain that currently lead to problems and could be improved in the future.

To conclude, the reengineer is supported in making a more informed and thereby probably better decisions when removing design deficiencies.

### 10.1.1 Discussion of the Limitations

As pointed out in Chapter 1, the contributions of this thesis are limited to component-based software architectures. This constraint allows to give more detailed statements about the quality of a system's architecture because certain rules (as described in Chapter 2.1) have to be obeyed. How the concepts can be applied to non-component-based software architectures, has to be investigated further. For example, the problem to narrow down the search space for the bad smell detection, would be more difficult if no clear boundaries between components are available.

The focus on bad smells on the architectural level limits the variety of problems in a software system to potential design deficiencies that concern larger parts of a system, than code bad smells. This allows to give statements about consequences on the overall architecture and how it changes when removing the bad smell which is not possible when only regarding bad smells on the code level.

Furthermore, the analyses done in the thesis focus on software written in an object-oriented programming language. Probably, most process steps can be adapted to work on non-object-oriented languages but this might be a labor-intensive task, since the existing concepts for the clustering, the bad smell detection and the contributions of this thesis were developed for object-oriented systems and the precise consequences of this adjustment have to be investigated.

## 10.2 Future Work

This section discusses ideas for future work that could not be realized within the scope of this thesis. It is structured into future work for the relevance analysis, for the reengineering strategies, for the architecture prognosis and miscellaneous future work.

### 10.2.1 Future Work for the Relevance Analysis

First of all, future work includes extending the relevance analysis by adding more relevance strategies to both, the component relevance analysis and the bad smell relevance analysis. By this, a more precise statement about the relevance could be given. For example, another relevance strategy in the component relevance analysis could use the metric value `InterfaceAdherence` or `InterfaceAccesses` and thereby rate components by the amount of communication via interfaces within them or between them and other components.

Furthermore, the bad smell relevance analysis should be extended to support a larger number of bad smells. At the moment, only the bad smells *Interface Violation* and *Communication via Non-Transfer Objects* are regarded within the scope of this thesis but there are many more interesting design deficiencies, like for example the bad smell *UnauthorizedCall*. It was also detected in CoCoME as Travkin reports in his thesis [Tra11].

In the component relevance analysis, larger improvements should be done. As already discussed in Section 8.4, some bad smells are only discovered when searching in more than one component. However, the component relevance analysis currently is based on suggestions for individual components to narrow down the search scope instead of regarding combinations of components. A future version of the component relevance analysis should rather consider the possibility to select a set of components as input for the bad smell detection than focusing on the rating of single components.

As already pointed out in Section 8.4, an interesting improvement of the bad smell relevance analysis could be to take into account groups of similar bad smell occurrences. Similar bad smell occurrences could be defined as involving the same interface or as sharing the same interface and the same accessing class. If an interface is bypassed several times (maybe even in the same class), the removal of these bad smells may be more critical than the removal of an interface violation of an interface that is only bypassed once. New relevance strategies that take considerations like this into account could be added to the bad smell relevance analysis and thereby improve the results significantly.

The extension of the result strategies that combine the values from the different relevance strategies could be worth considering. For example, instead of regarding the length of the vector from the origin (i.e. the geometric distance), also the manhattan distance [Bla04] could be used. This would have the effect that outliers in one of the relevance strategies receive more attention in the overall result.

Even though the relevance strategies can be realized to be loosely coupled from the rest of the analysis algorithm, which makes them easily replaceable and extendable, a more flexible solution to configure the relevance analysis could be useful. Particularly because the relevance strategies are specific for the bad smells which in turn depend on the project convention, it should be possible to let the user define her own relevance strategies. As a consequence, an idea for future work is to allow the user to specify the relevance strategies and even the result

strategies by herself. Then the relevance analysis could be adapted such that relevance strategies specified by the user are taken as input, calculated and presented in the result view. A specification editor for relevance strategies could list all available metrics from the clustering and provide operators for combining them to arithmetic expressions. These expressions should also be able to contain method calls for more complex relevance strategies that need to be formulated in a more powerful language, like the prevalent programming languages.

Another idea to improve the relevance analyses is to introduce weights for the relevance strategies. With this modification, the analyses can be made configurable, like the clustering. Since the clustering should be configured according to the characteristics of the system under analysis, this seems to be sensible for the relevance analyses, as well. For example, in systems in which the developers attach great importance to the package structure, the clustering should be configured with a high Package Mapping weight and according to this, a high weight for the Class Locations Strategy in the bad smell relevance analysis could also be considered.

### 10.2.2 Future Work for the Reengineering Strategies

More investigation has to be done on reengineering strategies, too. Additional strategies would give the user more selection possibilities to accomplish the removal of a bad smell occurrence in a way that fits her requirements best. As pointed out above, an extension to support a larger number bad smells is required. This holds for the reengineering strategies, too.

In particular, if more reengineering strategies are available, it is possible that some strategies are not always applicable for each bad smell occurrence of the same bad smell type. Because of this, it would be helpful to perform an automatic test of the applicability of a strategy. This could also be realized with story diagrams. Then, only the applicable strategies will be proposed to the user.

To further simplify the selection of a reengineering strategy, the strategy as well as the bad smell occurrence should be visualized in an adequate way. For the visualization of bad smell occurrences, the visualization of pattern detection results in Reclipse [PvDT11] could be reused.

To allow a more comfortable handling of the reengineering strategies, I recommend to create a reengineering strategies editor. In addition to a story diagram editor, the bad smell specification that belongs to a reengineering strategy could be regarded, e.g. the binding of annotated elements via parameters to object variables could be simplified, or the bad smell specification could be visualized beneath the editor. This would allow a much easier creation process for reengineering strategies because the input parameters and the object variables that have to be bound to them, can be calculated automatically. Furthermore, the annotations for the documentation of a reengineering strategy could be considered in such an editor.

Another idea is to regard sets of similar bad smell occurrences (cf. Section 8.4

and Section 10.2.1). For example, in some cases it could be sensible to remove all interface violations that concern the same interface at once, if the desired strategy is to extend the interface. It should be investigated if this is applicable for other bad smells or other reengineering strategies, as well. If this is the case, new reengineering strategies to realize this are to be created.

### 10.2.3 Future Work for the Architecture Prognosis

Currently, the architecture prognosis executes a new clustering on the whole architecture. This is sufficient at the moment but if larger software is analyzed and an architecture prognosis has to be executed for several combinations of bad smell occurrences and reengineering strategies, it could become a time-consuming process. A clustering of the whole system is not necessarily needed since only a few parts of the component structure are changed when applying a reengineering strategy. If it is possible to only regard a part of the system in the architecture prognosis, and how the concepts have to be changed in this case, could be a subject to further investigations. One idea is to use architectural slicing, as described by Zhao et al. [ZYXX02] (see Chapter 9).

Another part of the architecture prognosis that should be improved is the visualization. Some users might prefer a more graphical representation of the comparison between original architecture and predicted architecture, while the textual visualization could benefit from improvements that lead to a better overview, too. For this, existing architecture visualization approaches could be integrated. A promising candidate for this could be an approach to visualize enterprise architectures, based on software cartography developed at the TU München [BEL<sup>+</sup>07, Wit07]. This approach has already been used in combination with the Palladio Component Model [KSB<sup>+</sup>09].

Furthermore, more details on the predicted architecture could be of interest (see Chapter 6). Comparison criteria that are not realized yet could be taken into account in a next version of the architecture prognosis, for example, more details on modifications regarding the interfaces of components which have not been examined within the scope of this thesis.

In addition, parts of the architecture prognosis could easily be extracted to be used as an architecture comparison between two different architecture models that are available for the same project. With this extension, further interesting use cases would be supported, for example, the architecture comparison of different project branches that have been developed in parallel.

### 10.2.4 Miscellaneous Future Work

The proposed reengineering process provides much space for further enhancements, too.

One additional process step could be the integration of an automatic recommendation for a sensible order to remove bad smells. Here, it should be taken into

account that the removal of one bad smell occurrence could make another bad smell occurrence obsolete. Moreover, applying one reengineering strategy could give rise to other strategies, as considered by Tourwé and Mens [TM03]. The development of a concept for such a reengineering recommendation requires to study the dependencies between different bad smell occurrences or different reengineering strategies. A similar approach is described by Counsell et al. [CHN<sup>+</sup>06] and Liu et al. [LYN<sup>+</sup>09].

Another topic for future work is the last process step, the execution of the actual transformation of the system's source code. After the reengineer has made her decision for a reengineering (i.e., a bad smell occurrence to remove and a reengineering strategy to accomplish this), the transformation should be executed automatically (if possible) to avoid error-prone and time-consuming extra work for the reengineer. Currently, the reengineering results in a modified GAST model which can be the input for a new iteration of the reverse engineering process but the actual input system, i.e., the source code of the system, remains unchanged. Thus, a way to manipulate the underlying source code according to the model transformations has to be found. This could for example be the generation of new source code from the modified AST.

Another open question is when to end the process. At the moment, the reengineer has to decide by herself when the reengineering process is finished by regarding the architecture clustered in the current iteration (see Chapter 3). To support her decision, a further analysis could be done that determines if the current architecture of the system is satisfying or if it needs further reengineering iterations. Here, measurements as done by Sarkar et al. [SKR08] could be useful.

Furthermore, it is not satisfying that the whole process, including the bad smell detection, currently only uses a static analysis of the system's structure. It has been demonstrated that a dynamic analysis is required to perform a reliable pattern detection that excludes false positives [Wen07, vDP09, Vol10]. Tourwé and Mens also mentioned that an approach which combines static and dynamic information for the bad smell detection, seems promising [TM03]. It is conceivable that the relevance analysis or the architecture prognosis could also benefit from the additional usage of dynamic information. Because of this, it should be checked if a dynamic analysis can be used to improve the results of this reengineering process.



# Appendix A

## Specifications

This Chapter details on the used bad smell specifications and the reengineering strategies to accomplish the removal of the bad smell occurrences detected on the basis of these specifications.

### A.1 Bad Smell Specifications

This section shows and describes the used specifications used in Reclipse to detect the bad smells Interface Violation and Communication via Non-Transfer Objects.

#### A.1.1 Interface Violation

The Reclipse specification of interface violation used to detect this bad smell is depicted in Figure A.1. This specification is an adapted version of the specification Travkin uses in his thesis [Tra11]. The reason is that the detected structure is reused in the reengineering strategies and there, some elements are needed, that were not references in Travkin's specification (e.g. the interface object).

The method that contains the interface violation is represented by the object `accessingMethod`. The class `accessingClass` is the owner of this method. The `accessingMethod` accesses another method which is named `accessedMethod`. This `accessedMethod` is located in another class `accessedMethodOwner` which implements the bypassed `interface`. The `accessingMethod` also contains a statement `castStmt` that does a cast to the concrete type `accessedMethodOwner`. The same statement accesses a variable with the `interface` type.

When creating this specification, a trade-off between an exact and detailed specification that does not lead to any false positives in the pattern detection and a more generalized version that allows to detect implementation variants of the pattern, too, had to be found. For example the variable `var` of the `interface` type, could be a local variable or a parameter. Two other implementation variants are that the cast and the call of the accessed method could be done in the same statement, or in different statements. Because the specification covers these possibilities, the pattern detection with this specification may detect a few false positives, e.g. in the case that a more complex combination of casts and calls is



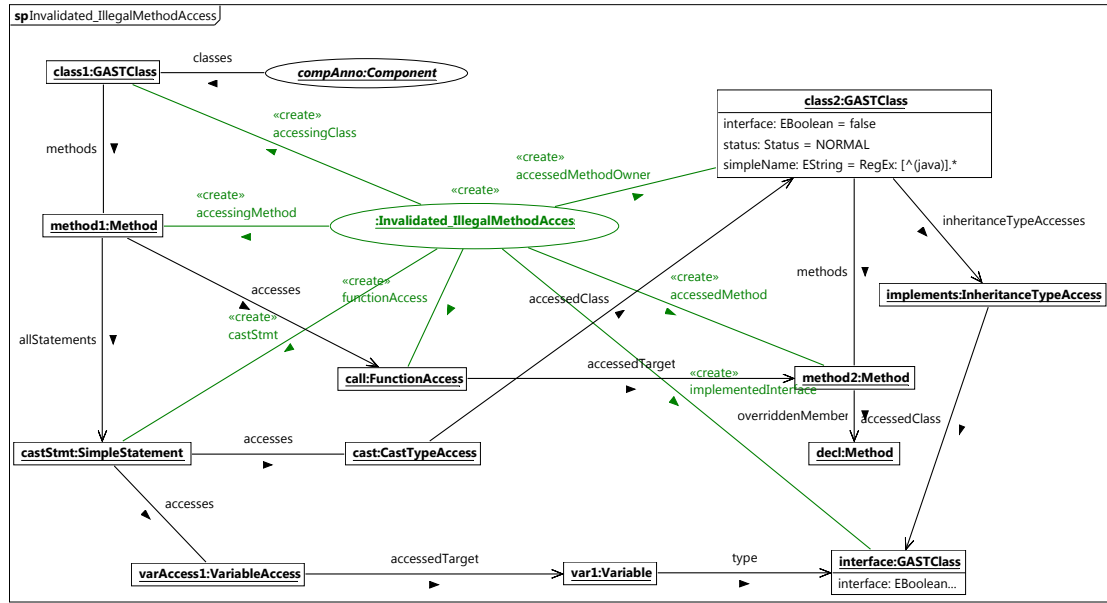


Figure A.2: Invalidated IllegalMethodAccess Structural Pattern

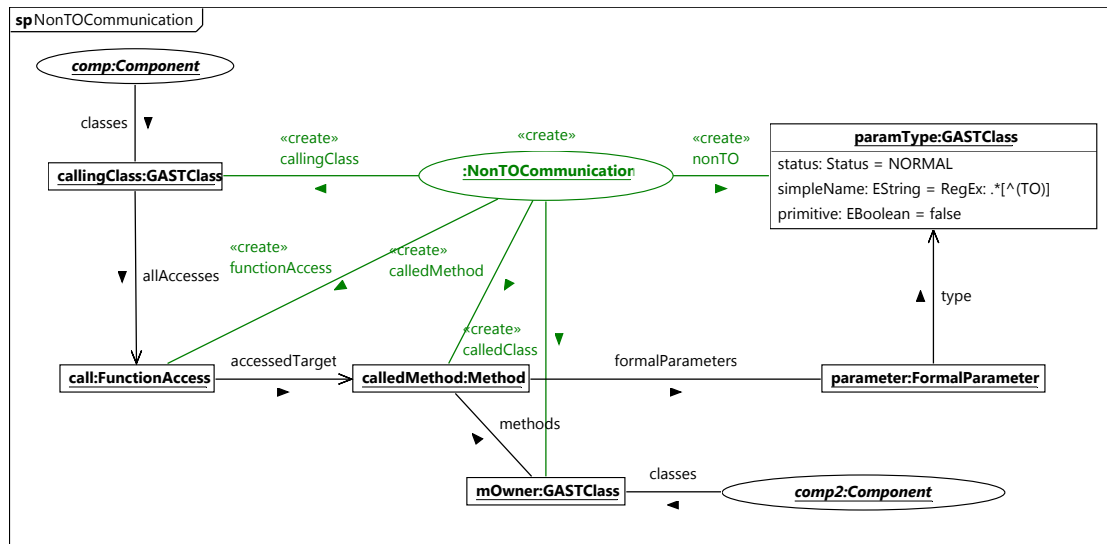


Figure A.3: NonTOCommunication Structural Pattern

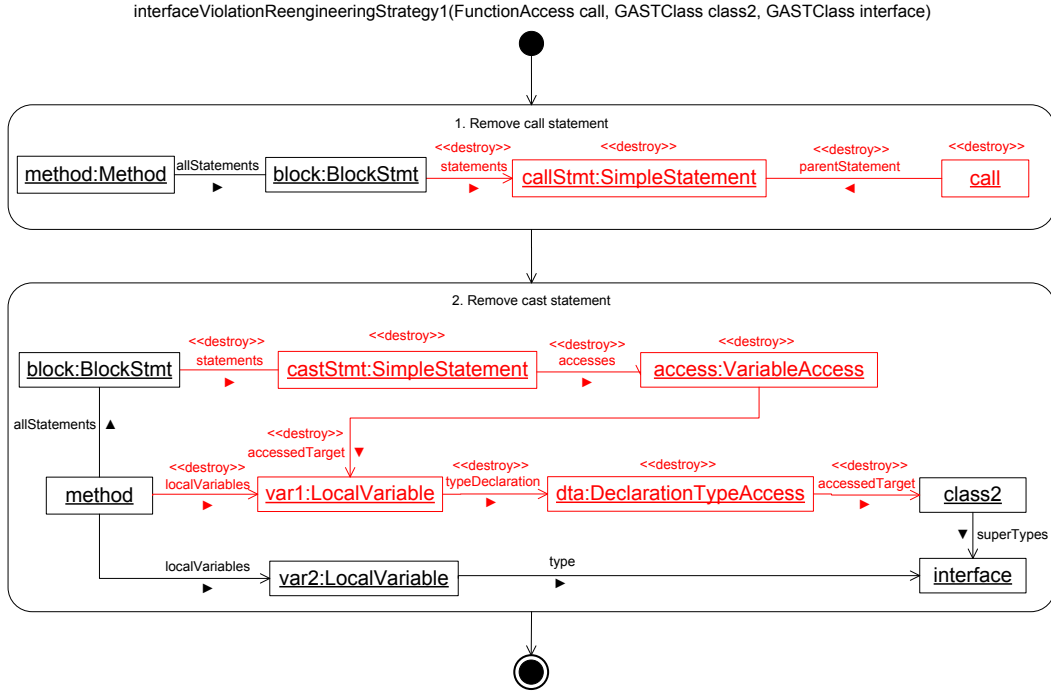


Figure A.4: Reengineering Strategy 1 for Interface Violation: remove call

problematic method call (`callingClass`) is part of another component then the class that owns the called method (`mOwner`). The type of the parameter of the called method (`paramType`) is the non-transfer object class. This specification is specialized for the detection in CoCoME, which can be seen at the `simpleName` attribute constrains, which is responsible for filtering transfer objects that are marked with T0.

## A.2 Reengineering Strategies

This section shows and describes the concrete story diagrams that realize the reengineering strategies to remove an interface violation occurrence, as explained in Section 5 and occurrences of the bad smell Communication via Non-Transfer Objects.

### A.2.1 Reengineering Strategies to Remove Interface Violations

Figure A.4 shows a reengineering strategy that removes an *Interface Violation* occurrence by deleting the call as described in Chapter 5. The interface violation specification this strategy corresponds is depicted in Section A.1.

In the story node 1, the statement `callStmt` that contains the call, and the `call` itself are removed from their containing block. After the deletion of the call, the cast statement is not needed anymore. Because of this, in the story node

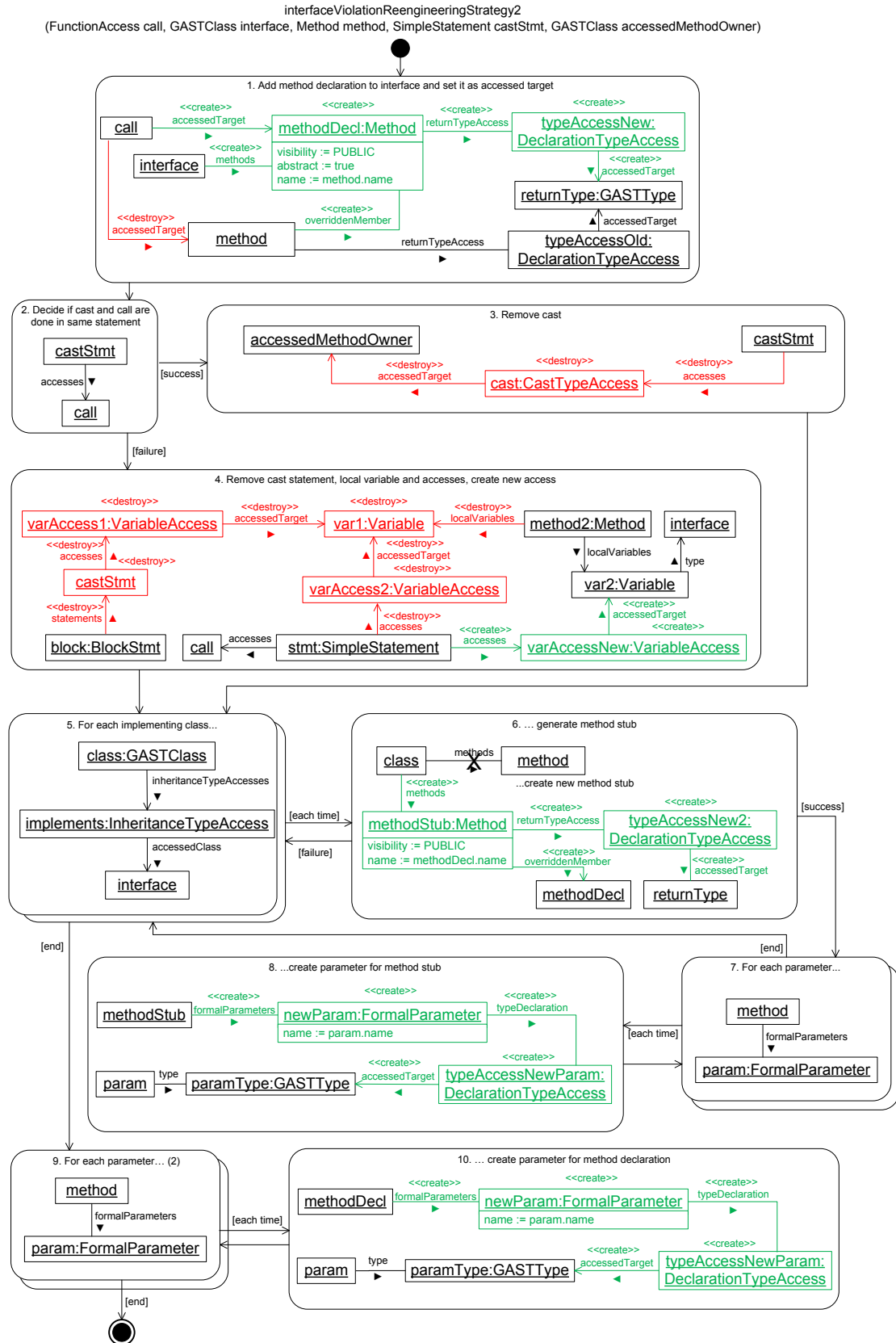


Figure A.5: Reengineering Strategy 2 for Interface Violation: add method declaration to interface

2, the cast statement `castStmt` is removed together with the therein contained variable access `access` and the accessed variable `var1` and the therein contained type access `dta`.

The reengineering strategy depicted in Figure A.5, removes an *Interface Violation* occurrence by extending the interface (see Chapter 5). In story node 1, a new method declaration `methodDecl` is created and added to the `interface`. This method declaration has the same return type as the concrete method `method` which overrides `methodDecl`. Furthermore, the target of the `call` is no longer `method` but now the newly created `methodDecl`. In the story node 2, a differentiation between two cases takes place: in the first case, the cast and the call are done in the same statement. If this is the case, only the cast has to be removed, which happens in story node 3. In the other case, the cast and the call are done in different statements. Then, story node 4 is executed. There, the no longer required cast statement (`castStmt`) and the local variable `var1` declared therein are deleted as well as all accesses to that variable. Furthermore, a new variable access `varAccessNew` on the variable with the type of the `interface` is created and added to the statement that contains the `call`. In the story nodes 5 and 6, for each `class` that implements the `interface`, a method `methodStub` for the newly created method declaration in the interface, is created and added to the class. The class that contained the interface violation is excluded from this procedure by the negative link to the `method` object variable and the `failure` edge. The `methodStub` objects get the same return types and parameters as the method that is pulled to the interface (story nodes 7 and 8). After that, the according parameters are also added to the newly created `methodDecl` (story nodes 9 and 10).

### **Reengineering Strategy to Remove Invalidated Interface Violation Occurrences**

Figure A.6 presents a reengineering strategy that is capable of removing interface violation occurrences that has been invalidated by the application of the reengineering strategy 2, depicted above. This reengineering strategies corresponds to the specification `Invalidated_IllegalMethodAccess`.

The reengineering strategy to remove invalidated interface violation occurrences is a part of reengineering strategy 2, which extends an interface. However, the strategy is a reduced version, since the interface does not have to be extended any more and only the cast and a variable access, if necessary, have to be deleted.

### **A.2.2 Reengineering Strategies to Remove Communication via Non-Transfer Object**

The trivial solution to accomplish the removal of a Communication via Non-Transfer Object occurrence is to simply remove the forbidden call. Figure A.7

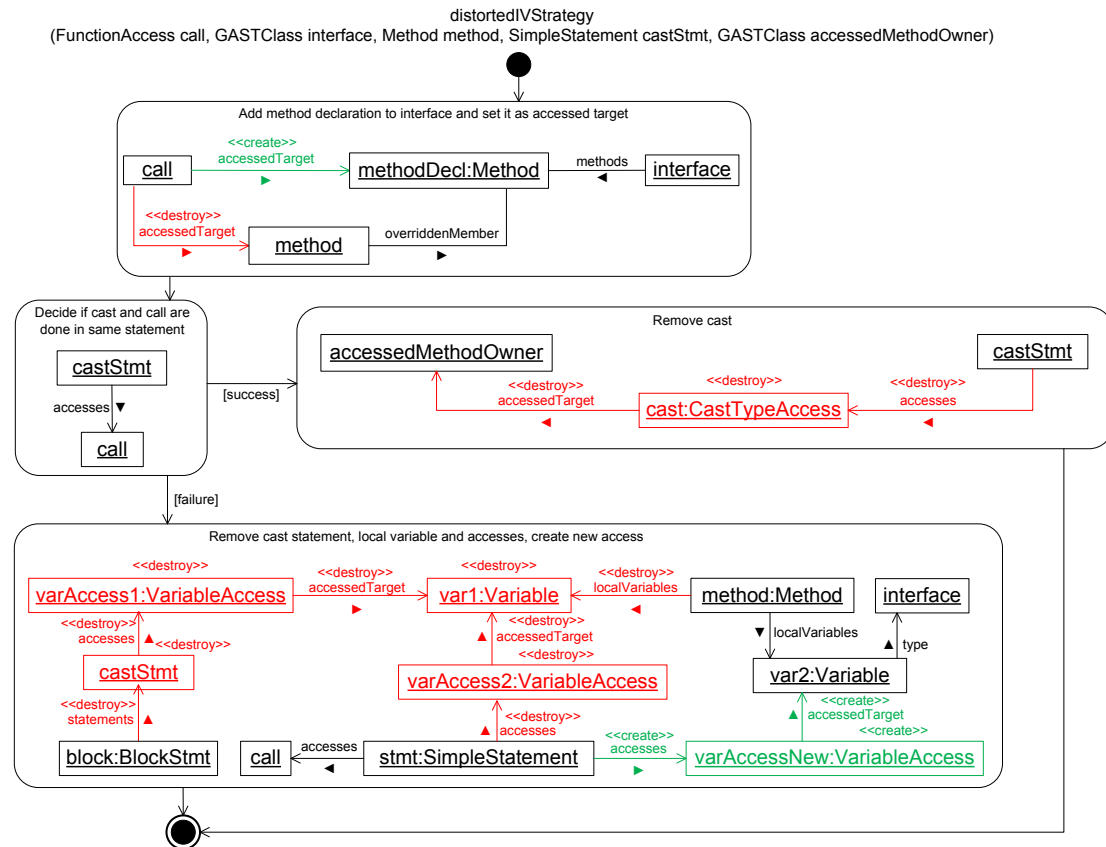


Figure A.6: Reengineering Strategy to Remove Invalidated Interface Violation Occurrences

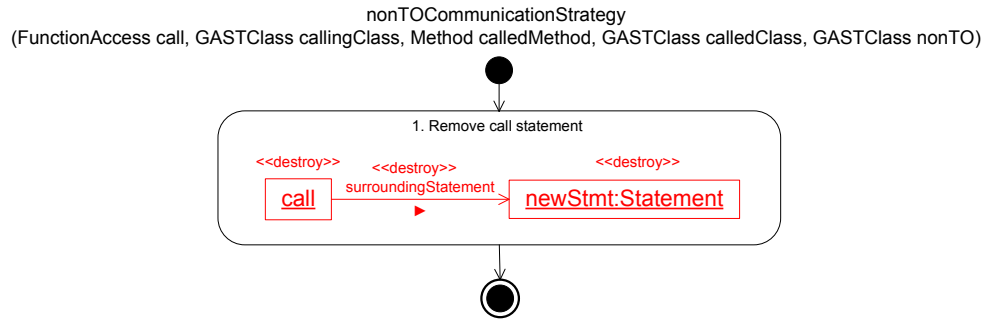


Figure A.7: Reengineering Strategy to Remove Non-Transfer Object Communication occurrences

depicts a story diagram that realizes this strategy. There, the bound `call` object is removed as well as the surrounding statement `newStmt`.

Another possible strategy could introduce a new interface that is implemented by the class of the non-transfer object. By this, the communication could be made explicit instead of violating the transfer object constraints. As a consequence the concerned components will be coupled tighter at each other. However, this strategy requires complicated interventions in the system's abstract syntax tree. To realize complex reengineering strategies like this is future work.



# Appendix B

## Recovered Architectures

This chapter lists the recovered components for the clusterings executed in the evaluation (see Chapter 8).

### B.1 Store Example, initial Clustering

Table B.1 depicts the configuration used for the clustering on the extended example store system.

The allocation of the classes to the components when using the configuration illustrated in Table B.1 is depicted in Figure B.1. The figure shows the components in a simplified component diagram. It is intended to show the hierarchical structure of the components. The interfaces and connectors are left out to beware the readability. In addition, the contained classes of the components are visualized by their names within the containing component.

Additionally, the following list shows the classes with their qualified names and their affiliation to the components:

- PC No. 58
  - de.upb.examples.reengineering.store.Main
  - de.upb.examples.reengineering.store.logic.StoreCreator
  - de.upb.examples.reengineering.store.logic.AccountOwnerCreator
  - de.upb.examples.reengineering.store.logic.ProductCreator
  - de.upb.examples.reengineering.store.logic.ProductSearch
  - de.upb.examples.reengineering.store.logic.StoreManager
  - de.upb.examples.reengineering.store.logic.PriceCalculator
  - de.upb.examples.reengineering.store.logic.ProducerSearch
  - de.upb.examples.reengineering.store.logic.CustomerSearch
  - de.upb.examples.reengineering.store.ui.MainMenu
  - de.upb.examples.reengineering.store.ui.ProductsListView
  - de.upb.examples.reengineering.store.ui.StorePresenter
  - de.upb.examples.reengineering.store.ui.ProductsListViewEntry
  - de.upb.examples.reengineering.store.ui.seller.SellerListView
  - de.upb.examples.reengineering.store.ui.seller.SellerMenu
- PC No. 60

Metric	Weight
Package Mapping	70
Directory Mapping	0
DMS	5
Low Coupling	0
High Coupling	15
Low Name Resemblance	5
Mid Name Resemblance	15
High Name Resemblance	30
Highest Name Resemblance	45
Low SLAQ	0
High SLAQ	15
Composition: Interface Adherence	40
Clustering Composition Threshold Max Value	100
Clustering Composition Threshold Min Value	25
Clustering Composition Threshold Decrement	10
Merge: Interface Violation	10
Clustering Merge Threshold Max Value	100
Clustering Merge Threshold Min Value	45
Clustering Merge Threshold Increment	10
Blacklist	java.*
Additional filter	.*TO

Table B.1: Configuration used for the Clustering on the Store System

- de.upb.examples.reengineering.store.model.impl.DVDImpl
- de.upb.examples.reengineering.store.model.impl.StorePackageImpl
- de.upb.examples.reengineering.store.model.impl.ProducerImpl
- de.upb.examples.reengineering.store.model.impl.StoreFactoryImpl
- de.upb.examples.reengineering.store.model.impl.WishlistImpl
- de.upb.examples.reengineering.store.model.impl.SellerImpl
- de.upb.examples.reengineering.store.model.impl.BookImpl
- de.upb.examples.reengineering.store.model.impl.ProductImpl
- de.upb.examples.reengineering.store.model.impl.CustomerImpl
- de.upb.examples.reengineering.store.model.impl.StoreImpl
- de.upb.examples.reengineering.store.model.util.StoreAdapterFactory
- de.upb.examples.reengineering.store.model.util.StoreSwitch
- PC No. 64
  - de.upb.examples.reengineering.store.ui.customer.CustomerListView
  - de.upb.examples.reengineering.store.ui.customer.CustomerMenu
- CC No. 1

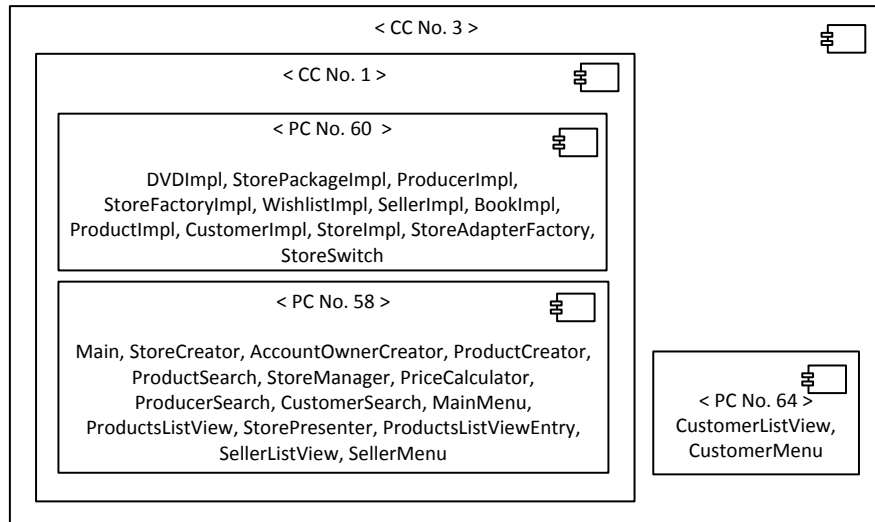


Figure B.1: Components recovered from the extended Store Example

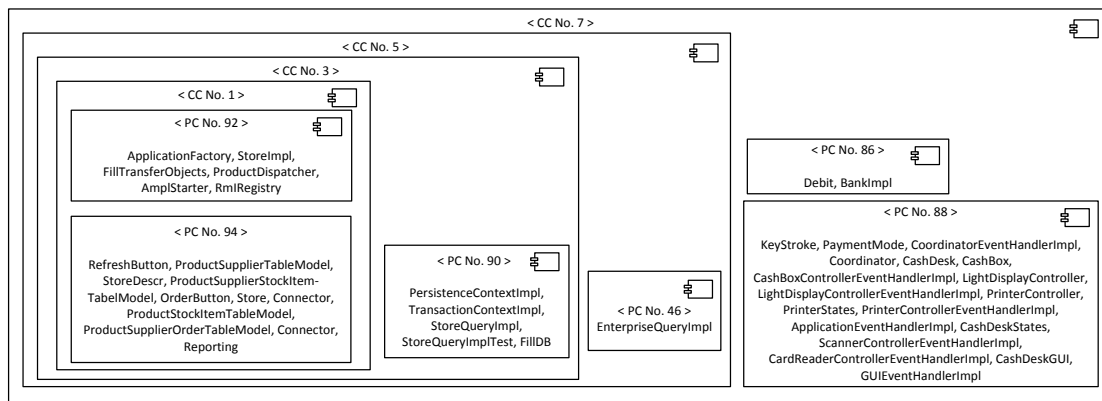


Figure B.2: Results of the initial Clustering in CoCoME

- PC No. 58
- PC No. 60
- CC No. 3
  - PC No. 64
  - CC No. 1

## B.2 CoCoME, initial Clustering

The recovered components and the classes they contained in the initial clustering with CoCoME is depicted in Figure B.2. The used configuration has been described in Section 8.2.

The classes with their qualified names and their affiliation to the components is also listed here:

- PC No. 46
  - org.cocome.tradingsystem.inventory.data.enterprise.impl.EnterpriseQueryImpl
- PC No. 86
  - org.cocome.tradingsystem.external.Debit
  - org.cocome.tradingsystem.external.impl.BankImpl
- PC No. 88
  - org.cocome.tradingsystem.cashdeskline.datatypes.KeyStroke
  - org.cocome.tradingsystem.cashdeskline.datatypes.PaymentMode
  - org.cocome.tradingsystem.cashdeskline.coordinator.impl.CoordinatorEventHandlerImpl
  - org.cocome.tradingsystem.cashdeskline.coordinator.impl.Coordinator
  - org.cocome.tradingsystem.cashdeskline.cashdenk.CashDesk
  - org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.impl.CashBox
  - org.cocome.tradingsystem.cashdeskline.cashdesk.cashboxcontroller.impl.CashBoxControllerEventHandlerImpl
  - org.cocome.tradingsystem.cashdeskline.cashdesk.lightdisplaycontroller.impl.LightDisplayController
  - org.cocome.tradingsystem.cashdeskline.cashdesk.lightdisplaycontroller.impl.LightDisplayControllerEventHandlerImpl
  - org.cocome.tradingsystem.cashdeskline.cashdesk.printercontroller.impl.PrinterController
  - org.cocome.tradingsystem.cashdeskline.cashdesk.printercontroller.impl.PrinterStates
  - org.cocome.tradingsystem.cashdeskline.cashdesk.printercontroller.impl.PrinterControllerEventHandlerImpl
  - org.cocome.tradingsystem.cashdeskline.cashdesk.application.impl.ApplicationEventHandlerImpl
  - org.cocome.tradingsystem.cashdeskline.cashdesk.application.impl.CashDeskStates
  - org.cocome.tradingsystem.cashdeskline.cashdesk.scannercontroller.impl.ScannerControllerEventHandlerImpl
  - org.cocome.tradingsystem.cashdeskline.cashdesk.cardreadercontroller.impl.CardReaderControllerEventHandlerImpl
  - org.cocome.tradingsystem.cashdeskline.cashdesk.gui.impl.CashDeskGUI
  - org.cocome.tradingsystem.cashdeskline.cashdesk.gui.impl.GUIEventHandlerImpl
- PC No. 90
  - org.cocome.tradingsystem.inventory.data.persistence.impl.PersistenceContextImpl
  - org.cocome.tradingsystem.inventory.data.persistence.impl.TransactionContextImpl
  - org.cocome.tradingsystem.inventory.data.store.impl.StoreQueryImpl
  - org.cocome.tradingsystem.inventory.data.test.StoreQueryImplTest
  - org.cocome.tradingsystem.inventory.data.test.FillDB
- PC No. 92
  - org.cocome.tradingsystem.inventory.application.ApplicationFactory
  - org.cocome.tradingsystem.inventory.application.store.impl.StoreImpl
  - org.cocome.tradingsystem.inventory.application.store.impl.FillTransferObjects
  - org.cocome.tradingsystem.inventory.application.productdispatcher.impl.ProductDispatcher
  - org.cocome.tradingsystem.inventory.application.productdispatcher.impl.AmplStarter
  - org.cocome.tradingsystem.inventory.application.util.RmIRegistry
- PC No. 94
  - org.cocome.tradingsystem.inventory.gui.store.RefreshButton

- org.cocome.tradingsystem.inventory.gui.store.ProductSupplierTableModel
  - org.cocome.tradingsystem.inventory.gui.store.StoreDescr
  - org.cocome.tradingsystem.inventory.gui.store.ProductSupplierStockItemTableModel
  - org.cocome.tradingsystem.inventory.gui.store.OrderButton
  - org.cocome.tradingsystem.inventory.gui.store.Store
  - org.cocome.tradingsystem.inventory.gui.storeConnector
  - org.cocome.tradingsystem.inventory.gui.store.ProductStockItemTableModel
  - org.cocome.tradingsystem.inventory.gui.store.ProductSupplierOrderTableModel
  - org.cocome.tradingsystem.inventory.gui.reporting.Connector
  - org.cocome.tradingsystem.inventory.gui.reporting.Reporting
- CC No. 1
  - PC No. 92
  - PC No. 94
- CC No. 3
  - PC No. 92
  - CC No. 1
- CC No. 5
  - PC No. 46
  - CC No. 3
- CC No. 7
  - PC No. 86
  - PC No. 88
  - CC No. 5

## **B.3 Palladio FileShare, initial Clustering**

The allocation of the classes to the components when using the configuration illustrated in Section 8.3 is illustrated in Figure B.3 and in the following list:

- PC No. 38
  - de.uka.ipd.palladiofileshare.algorithms.SimpleLZW
- PC No. 86
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Decoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Decoder\$LiteralDecoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Decoder\$LenDecoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Decoder\$LiteralDecoder\$Decoder2
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Encoder\$LiteralEncoder\$Encoder2
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Encoder\$Optimal
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Encoder\$LenPriceTableEncoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Encoder\$LiteralEncoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Encoder\$LenEncoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Encoder

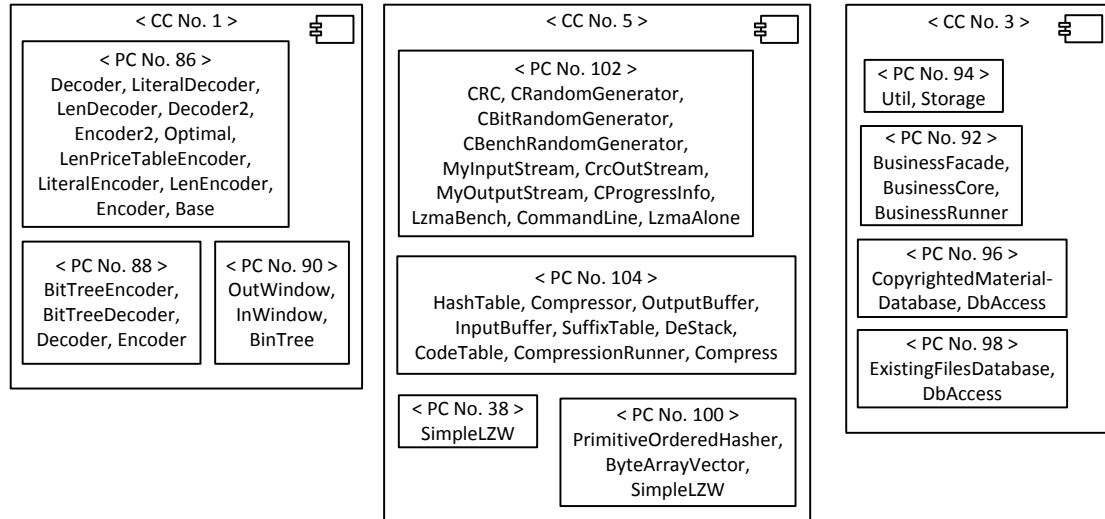


Figure B.3: Recovered Components in Palladio FileShare

- de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZMA.Base
- PC No. 88
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.RangeCoder.BitTreeEncoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.RangeCoder.BitTreeDecoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.RangeCoder.Decoder
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.RangeCoder.Encoder
- PC No. 90
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZ.OutWindow
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZ.InWindow
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.Compression.LZ.BinTree
- PC No. 92
  - de.uka.ipd.palladiofileshare.businesslogic.BusinessFacade
  - de.uka.ipd.palladiofileshare.businesslogic.BusinessCore
  - de.uka.ipd.palladiofileshare.businesslogic.BusinessRunner
- PC No. 94
  - de.uka.ipd.palladiofileshare.businesslogic.util.Util
  - de.uka.ipd.palladiofileshare.businesslogic.storage.Storage
- PC No. 96
  - de.uka.ipd.palladiofileshare.businesslogic.copyrightedmaterialsdb.CopyrightedMaterialDatabase
  - de.uka.ipd.palladiofileshare.businesslogic.copyrightedmaterialsdb.DbAccess
- PC No. 98
  - de.uka.ipd.palladiofileshare.businesslogic.existingfilesdb.ExistingFilesDatabase
  - de.uka.ipd.palladiofileshare.businesslogic.existingfilesdb.DbAccess
- PC No. 100

- de.uka.ipd.palladiofileshare.legacy.algorithms.PrimitiveOrderedHasher
- de.uka.ipd.palladiofileshare.legacy.algorithms.ByteArrayVector
- de.uka.ipd.palladiofileshare.legacy.algorithms.SimpleLZW
- PC No. 102
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.CRC
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaBench\$CRandomGenerator
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaBench\$CBitRandomGenerator
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaBench\$CBenchRandomGenerator
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaBench\$MyInputStream
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaBench\$CrcOutputStream
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaBench\$MyOutputStream
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaBench\$CProgressInfo
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaBench
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaAlone\$CommandLine
  - de.uka.ipd.palladiofileshare.legacy.algorithms.SevenZip.LzmaAlone
- PC No. 104
  - de.uka.ipd.palladiofileshare.legacy.algorithms.Compressor\$HashTable
  - de.uka.ipd.palladiofileshare.legacy.algorithms.Compressor
  - de.uka.ipd.palladiofileshare.legacy.algorithms.OutputBuffer
  - de.uka.ipd.palladiofileshare.legacy.algorithms.InputBuffer
  - de.uka.ipd.palladiofileshare.legacy.algorithms.Decompressor\$SuffixTable
  - de.uka.ipd.palladiofileshare.legacy.algorithms.Decompressor\$DeStack
  - de.uka.ipd.palladiofileshare.legacy.algorithms.CodeTable
  - de.uka.ipd.palladiofileshare.legacy.algorithms.CompressionRunner
  - de.uka.ipd.palladiofileshare.legacy.algorithms.Compress
- PC No. 106
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.Decompressor
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.CompressionRunner
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.OutputBuffer
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.Compressor
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.Compress
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.SuffixTable
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.CodeTable
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.HashTable
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.InputBuffer
  - de.uka.ipd.palladiofileshare.legacy.algorithms.compress\_refactored.DeStack
- CC No. 1
  - PC No. 86
  - PC No. 88
  - PC No. 90
- CC No. 3
  - PC No. 92

- PC No. 94
  - PC No. 96
  - PC No. 98
  - PC No. 38
- CC No. 5
  - PC No. 100
  - PC No. 102
  - PC No. 104
  - PC No. 106



# Appendix C

## Eclipse Plug-Ins

In the following sections, the Eclipse plug-ins that have been realized in the scope of this thesis as well as the plug-ins that are required to execute the realized plug-ins are listed.

### C.1 Required Plug-Ins

As described in Chapter 7, the parser **SISSy** and the **GAST** meta model are required. Furthermore, some plug-ins of the **Palladio Component Model (PCM)** are required. All these plug-ins are available via the update site of the **Q-Impress** release: <http://q-impress.ow2.org/release>.

Additionally, for the clustering with **SoMoX** some plug-ins from the **Q-Impress** repository (<svn://svn.forge.objectweb.org/svnroot/>) are needed:

- `eu.qimpress.reverseengineering.gast2seff`
- `eu.qimpress.samm`
- `eu.qimpress.seff`
- `eu.qimpress.sourcecodedecorator`
- `org.jgrapht`
- `org.somox.analyzer.sissymodelanalyzer`
- `org.somox.analyzer.sissymodelanalyzer.ui`
- `org.somox.core`
- `org.somox.filter`
- `org.somox.metrics`
- `org.somox.metrics.dsl`
- `org.somox.metricValuesPersistency`

- `org.somox.ProvidedRequiredIds`
- `org.somox.resource.defaultmodels`
- `org.somox.ui`
- `uk.ac.shef.dcs.simmetrics`

For the story diagrams, the story diagram meta model is required. The repository is available under <http://svn.codespot.com/a/eclipselabs.org/sdm-commons/> and the required plug-in is `org.storydriven.modeling`. From this repository also the plug-in `org.storydriven.modeling.interpreter.adapter` is required because it contains the classes that are used by the story diagram interpreter to interpret the story diagrams created with the meta model named above.

The story diagram interpreter itself is located in the repository of the HPI in Potsdam: <https://www.hpi.uni-potsdam.de/giese/gforge/svn/storyeditor/>. The required plug-ins are:

- `de.mdelab.sdm`
- `de.mdelab.sdm.interpreter.common`
- `de.mdelab.sdm.interpreter.common.eclipse`
- `de.mdelab.sdm.interpreter.ocl`

Reclipse can be downloaded via the update site:  
<http://dsd-serv.uni-paderborn.de/svn/updatesites/trunk/reclipse/>.

## C.2 Realized Plug-Ins

Within the scope of this thesis, several plug-ins have been realized:

**org.archimatrix.relevanceanalysis:** This plug-in realizes the relevance analysis.

**org.archimatrix.relevanceanalysis.ui:** This plug-in provides the user interface of the relevance analysis.

**org.archimatrix.architectureprognosis:** This plug-in realizes the architecture prognosis.

**org.archimatrix.architectureprognosis.ui:** This plug-in provides the user interface of the architecture prognosis.

**org.archimatrix.common:** In this plug-in some common functions and constants that are used by the relevance analysis and the architecture prognosis plug-ins are contained.

`org.somox.metricValuesPersistency`: This plug-in contains the meta model classes used to store the metric values during the clustering.

The plug-in that has been modified to store the metric values during the clustering is `org.somox.analyzer.sissymodelanalyzer`.



# Appendix D

## User Guide

This chapter briefly describes how the proposed reengineering process can be executed using the realized tool. For this, the user has to follow the steps described in the following:

1. Creating a GAST model of the system under analysis:

First, a generalized abstract syntax tree (GAST) of the system under analysis has to be created. For this, the parser SISSy can be used. A documentation for SISSy is available under <http://www.sqools.org/sissy>.

2. Initial clustering:

In the next step, SoMoX is used to cluster the system. A documentation for SoMoX is available under <http://www.sqools.org/somox>.

3. Component relevance analysis:

The component relevance analysis can be started via the menu bar: **Archi-metrix** → **Reengineering of Design Deficiencies** → **Find Relevant Components**. The opened wizard requires the source code decorator model from the clustering (\*.sourcecodedecorator) and the metric values model (\*.ecore) as input. Both files were created during the initial clustering. Using the default settings, they are both saved in a folder “model” in the surrounding project folder.

Depending on the size of the system under analysis, the component relevance analysis takes some time to load the required model elements. After that, the **Relevant Component View** opens (see Section 7.3.1 for a description of the view).

4. Bad smell detection:

The bad smell detection on a set of selected components can be started via the menu bar: **Reclipse EMF** → **Start Pattern Based Architecture Analysis**.

Another possibility to start the bad smell detection on a selected component is via the context menu in the **Relevant Components View**.

When the bad smell detection is finished, the detected bad smell occurrences are listed in the view **Annotations**. This view provides buttons for loading and saving the detection results into a file. To execute a bad smell relevance analysis, the detected bad smells have to be saved.

5. Bad smell relevance analysis:

The bad smell relevance analysis can be started via the menu bar: **Archimetric** → **Reengineering of Design Deficiencies** → **Find Relevant Bad Smells**. The opened wizard requires the file with the saved bad smell occurrences (\*.psa) and the metric values model as input.

Here again, depending on the size of the system under analysis, the bad smell relevance analysis takes some time to load the required model elements. After that, the **Relevant Bad Smells View** opens (see Section 7.3.1 for a description of the view).

6. Architecture prognosis:

The architecture prognosis can be started via the menu bar (**Archimetric** → **Reengineering of Design Deficiencies** → **View Architecture Prognosis**) or via the context menu for a bad smell occurrence in the **Relevant Bad Smells View**.

The **Architecture Prognosis Wizard** takes the metric values model, the detected bad smell occurrences and a file with reengineering strategies as input (\*.ecore) as input. On the second wizard page, the bad smell occurrence to be removed has to be selected and on the third page, the reengineering strategy for which the prognosis shall be executed, has to be selected.

After having finished the wizard, the **Architecture Prognosis View** shows the prognosis results (see Section 7.5.4 for a description of the view).

To perform the next iteration of the reengineering process the clustering results achieved in the architecture prognosis can be used as input for the bad smell detection.

# Bibliography

- [BEL<sup>+</sup>07] S. Buckl, A.M. Ernst, J. Lankes, C.M. Schweda, and A. Wittenburg. Generating visualizations of enterprise architectures using model transformations. *Enterprise Modelling and Information Systems (EMISA 2007)*, page 33, 2007.
- [BHT<sup>+</sup>10] S. Becker, M. Hauck, M. Trifu, K. Krogmann, and J. Kofron. Reverse engineering component models for quality predictions. In *14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pages 194–197. IEEE Computer Society, 2010.
- [BK07] F. Bourquin and R.K. Keller. High-impact refactoring based on architecture violations. In *11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 149–158. IEEE Computer Society, 2007.
- [Bla04] P.E. Black. *Dictionary of algorithms and data structures*. National Institute of Standards and Technology, 2004.
- [BMMM98] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mombroy. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, Inc., 1998.
- [CDJ10] C. Coello, C. Dhaenens, and L. Jourdan. Multi-objective combinatorial optimization: Problematic and context. *Advances in Multi-Objective Nature Inspired Computing*, pages 1–21, 2010.
- [CHN<sup>+</sup>06] S. Counsell, R.M. Hierons, R. Najjar, G. Loizou, and Y. Hassoun. The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph. In *Testing: Academic and Industrial Conference-Practice And Research Techniques*, pages 181–192. IEEE, 2006.
- [CKK01] Eun Sook Cho, Min Sun Kim, and Soo Dong Kim. Component metrics to measure component quality. In *Eighth Asia-Pacific Software Engineering Conference (APSEC 2001)*, pages 419 – 426, dec 2001.
- [CKK08] Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. Reverse engineering software-models of component-based systems. In Kostas Kontogiannis, Christos Tjortjis, and Andreas Winter, editors, *Proceedings of the 12th European Conference on Software Maintenance*

- and Reengineering (CSMR 2008)*, pages 93–102, Athens, Greece, April 1–4 2008. IEEE Computer Society.
- [DDN03] S. Demeyer, S. Ducasse, and O.M. Nierstrasz. *Object-oriented reengineering patterns*. Morgan Kaufmann, 2003.
- [DP09] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.
- [FNTZ00] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. *Theory and Application of Graph Transformations*, pages 157–167, 2000.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented design*. Addison-Wesley Reading, MA;, 1995.
- [GL91] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, pages 751–761, 1991.
- [HKW<sup>+</sup>08] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolk, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. CoCoME - The Common Component Modeling Example. In *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 16–53. Springer Berlin / Heidelberg, 2008.
- [KG08] C.J. Kapser and M.W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.
- [KKR10] K. Krogmann, M. Kuperberg, and R. Reussner. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Transactions on Software Engineering*, pages 865–877, 2010.
- [KP05] B. Ko and J. Park. Component Architecture Redesigning Approach Using Component Metrics. *Artificial Intelligence and Simulation*, pages 449–459, 2005.



- [Kro10] Klaus Krogmann. *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2010.
- [KSB<sup>+</sup>09] Klaus Krogmann, Christian M. Schweda, Sabine Buckl, Michael Kuperberg, Anne Martens, and Florian Matthes. Improved Feedback for Architectural Performance Prediction using Software Cartography Visualizations. In Christine Hofmeister Raffaella Mirandola, Ian Gorton, editor, *Architectures for Adaptive Systems (QoSA 2009)*, volume 5581 of *Lecture Notes in Computer Science*, pages 52–69. Springer, 2009.
- [KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-Based Reverse-Engineering of Design Components. In *Proc. of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 226–235. IEEE Computer Society Press, May 1999.
- [LS07] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 2007.
- [LTC02] M. Lindvall, R. Tesoriero, and P. Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *Eighth IEEE Symposium on Software Metrics*, pages 77–86. IEEE Computer Society, 2002.
- [LYN<sup>+</sup>09] Hui Liu, Limei Yang, Zhendong Niu, Zhyi Ma, and Weizhong Shao. Facilitating software refactoring with appropriate resolution order of bad smells. In Hans van Vliet and Valérie Issarny, editors, *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, pages 265–268. ACM, 2009.
- [Mar94] R. Martin. Oo design quality metrics – an analysis of dependencies. In *Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics (OOPSLA 1994)*, volume 94, 1994.
- [Mar04] Radu Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 350 – 359, September 2004.
- [MGDLM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A Method for the Specification and

- Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36:20–36, January 2010.
- [MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
- [Mun05] Matthew James Munro. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *Proceedings of the 11th IEEE International Symposium on Software Metrics*, pages 9–17. IEEE Computer Society, September 2005.
- [Mye75] G.J. Myers. *Reliable software through composite design*. Petrocelli/Charter, 1975.
- [Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [PvDT11] Marie Christin Platenius, Markus von Detten, and Dietrich Travkin. Visualization of Pattern Detection Results in Reclipse. In *Proceedings of the 8th International Fujaba Days*. University of Tartu, Estonia, 2011.
- [SA4] SA4J. Structural Analysis for Java. <http://www.alphaworks.ibm.com/tech/sa4j>. visited April 2011.
- [SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.
- [SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002.
- [SKR08] Santonu Sarkar, Avinash C. Kak, and Girish Maskeri Rama. Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software. *IEEE Transactions on Software Engineering*, 34(5):700–720, October 2008.
- [SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics Based Refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 30–39. IEEE Computer Society Press, 2001.
- [SSM06] F. Simon, O. Seng, and T. Mohaupt. *Code-Quality-Management*. dpunkt-Verl., 2006.

- [TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Seventh European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 91–100. IEEE, 2003.
- [Tra11] Oleg Travkin. *Kombination von Clustering- und musterbasierten Reverse-Engineering-Verfahren*. Masterarbeit, University of Paderborn, June 2011. In German.
- [TSG04] Adrian Trifu, Olaf Seng, and Thomas Genssler. Automated Design Flaw Correction in Object-Oriented Systems. In *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.
- [TvDB11] Oleg Travkin, Markus von Detten, and Steffen Becker. Towards the Combination of Clustering-based and Pattern-based Reverse Engineering Approaches. In *Proceedings of the 3rd Workshop of the GI Working Group L2S2 - Design for Future 2011*, February 2011.
- [vDB11] M. von Detten and S. Becker. Combining Clustering-based and Pattern Detection for the Reengineering of Component-based Software Systems. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, pages 23–32. ACM, 2011.
- [vDMT10a] Markus von Detten, Matthias Meyer, and Dietrich Travkin. Reclipse – A Reverse Engineering Tool Suite. Technical Report tr-ri-10-312, University of Paderborn, Paderborn, Germany, 2010.
- [vDMT10b] Markus von Detten, Matthias Meyer, and Dietrich Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010)*, volume 2, pages 299–300. ACM Press, May 2010. Informal Research Demonstration.
- [vDP09] Markus von Detten and Marie Christin Platenius. Improving Dynamic Design Pattern Detection in Reclipse with Set Objects. In *Proceedings of the 7th International Fujaba Days*, pages 15–19. Eindhoven University of Technology, 2009.
- [VGB02] J. Van Gurp and J. Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

- [Vol10] Andreas Volk. *Ein Verfahren zur Trace-Generierung für die verhaltensbasierte Entwurfsmustererkennung mit Hilfe eines Model Checkers*. Bachelor's thesis, University of Paderborn, November 2010. In German.
- [Wen07] Lothar Wendehals. *Struktur- und verhaltensbasierte Entwurfsmustererkennung*. PhD thesis, University of Paderborn, September 2007. In German.
- [Wit07] A. Wittenburg. Softwarekartographie: Modelle und Methoden zur Systematischen Visualisierung von Anwendungslandschaften. *München. Technische Universität München, Institut für Informatik*, 2007.
- [WYF03] Hironori Washizaki, Hirokazu Yamamoto, and Yoshiaki Fukazawa. A metrics suite for measuring reusability of software components. In *Proceedings of the 9th International Symposium on Software Metrics*, pages 211–, Washington, DC, USA, 2003. IEEE Computer Society.
- [Zö1] Albert Zündorf. Rigorous object oriented software development. *University of Paderborn*, 6, 2001.
- [ZHB11] Min Zhang, Tracy Hall, and Nathan Baddoo. Code Bad Smells: A Review of Current Knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202, April 2011.
- [ZYXX02] J. Zhao, H. Yang, L. Xiang, and B. Xu. Change impact analysis to support architectural evolution. *Journal of software maintenance and evolution: research and practice*, 14(5):317–333, 2002.