

Reengineering of Component-Based Software Systems in the Presence of Design Deficiencies

by

Markus von Detten



UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik

Heinz Nixdorf Institut und Institut für Informatik

Fachgebiet Softwaretechnik

Zukunftsmeile 1

33102 Paderborn

Reengineering of Component-Based Software Systems in the Presence of Design Deficiencies

PhD Thesis

to obtain the degree of

“Doktor der Naturwissenschaften (Dr. rer. nat.)”

by

MARKUS VON DETTEN

Balduinstraße 4

33102 Paderborn

Referee:

Prof. Dr. Wilhelm Schäfer

Paderborn, March 26, 2013

Abstract

The maintenance of component-based software systems requires up-to-date models of their concrete architecture, i.e. the architecture that is realised in the source code. These models help in systematically planning, analysing and executing typical reengineering activities.

Often no or only outdated architectural models of such systems exist. Therefore, various reverse engineering methods have been developed which try to recover a system's components, subsystems and connectors. However, these reverse engineering methods are severely impacted by design deficiencies in the system's code base, especially violations of the component encapsulation. As long as design deficiencies are not considered in the reverse engineering process, they reduce the quality of the recovered component structures.

Despite this impact of design deficiencies, no existing architecture reconstruction approach explicitly integrates a systematic deficiency detection and removal into the recovery process.

Therefore, I have developed Archimetrix. Archimetrix is a tool-supported architecture reconstruction process. It enhances a clustering-based architecture recovery approach with an extensible, pattern-based deficiency detection. After the detection of deficiencies, Archimetrix supports the software architect in removing the deficiencies and provides the means to preview the architectural consequences of such a removal. I also provide a process to identify and formalise additional deficiencies.

I validated the approach on three case studies which show that Archimetrix is able to identify relevant deficiencies and that the removal of these deficiencies leads to an increased quality of the recovered architectures, i.e. they are closer to the corresponding conceptual architectures.

Zusammenfassung

Für die Wartung komponenten-basierter Software werden aktuelle Modelle ihrer konkreten Architektur, d.h. der Architektur, welche im Quellcode umgesetzt wurde, benötigt. Diese Modelle unterstützen den Software-Architekten bei der Planung, der Analyse und der Ausführung von typischen Reengineering-Aktivitäten.

Allerdings existieren häufig keine oder nur veraltete Architekturmodelle solcher Software-Systeme. Daher wurden in der Vergangenheit zahlreiche Reverse-Engineering-Verfahren entwickelt, welche dazu dienen, die Komponenten, Subsysteme und Konnektoren komponenten-basierter Software wiederzuerkennen. Allerdings werden diese Reverse-Engineering-Verfahren durch Schwachstellen im Quellcode – vor allem durch Schwachstellen, die die Kapselung von Komponenten verletzen – stark beeinflusst. Werden solche Schwachstellen bei der Wiedergewinnung von Architekturmodellen nicht berücksichtigt, können sie die Qualität der erkannten Komponentenstrukturen erheblich verringern.

Trotz dieses signifikanten Einflusses von Schwachstellen, werden diese im Erkennungsprozess existierender Architektur-Rekonstruktions-Verfahren bisher nicht berücksichtigt.

Zur Lösung dieses Problems habe ich im Rahmen dieser Arbeit Archimetrix entwickelt. Archimetrix ist ein werkzeuggestütztes Architektur-Rekonstruktions-Verfahren. Es erweitert einen bestehenden, clustering-basierten Architektur-Rekonstruktions-Ansatz um ein erweiterbares, muster-basiertes Verfahren zur Schwachstellenerkennung. Nach der Schwachstellenerkennung unterstützt Archimetrix den Software-Architekten zusätzlich bei der Entfernung der gefundenen Probleme und ermöglicht es ihm, die Auswirkungen der Entfernung auf die Software-Architektur des Systems zu analysieren. Außerdem beschreibt diese Arbeit einen Prozess zur Identifikation, Dokumentation und Formalisierung von Schwachstellen.

Archimetrix wurde an drei Fallstudien evaluiert, welche zeigen, dass Archimetrix zuverlässig relevante Schwachstellen identifizieren kann und dass die Entfernung dieser Schwachstellen die Qualität der rekonstruierten Architekturen erhöht, d.h. dass diese Architekturen besser mit den ursprünglich dokumentierten Architekturen übereinstimmen.

Danke

Viele Menschen glauben, dass Informatiker lichtscheue Einzelgänger sind, die in ihrem stillen Kämmerlein vor dem Bildschirm hocken und direkten Kontakt höchstens zum Pizzaboten haben. Dieses Vorurteil kann ich nicht bestätigen. Ganz auf mich allein gestellt hätte ich diese Arbeit nie schreiben können. Stattdessen bin ich sehr vielen Menschen, die mich in den letzten Jahren begleitet und unterstützt haben, zum Dank verpflichtet.

Zuallererst gebührt mein Dank meinem Doktorvater Wilhelm Schäfer. Er hat mir nicht nur die Möglichkeit gegeben, diesen Weg zu gehen, sondern er hat auch in seiner Arbeitsgruppe ein Umfeld geschaffen, in dem man gemeinschaftlich und ohne überflüssiges Konkurrenzdenken miteinander forschen und arbeiten kann. Ich habe mich hier in den vergangenen fünf Jahren sehr wohl gefühlt.

Ich danke auch den weiteren Mitgliedern meiner Prüfungskommission, Ralf Reussner, Uwe Kastens, Steffen Becker und Stefan Sauer, für ihre Bereitschaft, sich mit meiner Arbeit auseinanderzusetzen. Ralf Reussner und Steffen Becker danke ich außerdem für die Anfertigung ihrer Gutachten.

Archimetrix wäre nicht entstanden, wenn ich nicht die Unterstützung von einigen ganz besonderen Personen gehabt hätte, sei es in intensiven Diskussionen, beim Schreiben von Papieren oder durch die Anfertigung von Masterarbeiten. Mein Dank gilt hier Steffen Becker, Marie Christin Platenius und Oleg Travkin.

Dass ich mich in der AG Schäfer so wohl gefühlt habe, ist natürlich vor allem meinen Kollegen in dieser Zeit zu verdanken. Neben all den fachlichen Diskussionen haben auch die Gespräche und Aktivitäten abseits der Wissenschaft – zum Beispiel in den Kaffeerunden, bei Filmabenden und bei den AG-Ausflügen – viel Spaß gemacht. Diese großartigen Kollegen sind Björn Axenath, Matthias Becker, Steffen Becker, Christian Bimmermann, Christian Brenner, Christopher Brink, Nicola Danielzik, Stefan Dziwok, Tobias Eckhardt, Remo Ferrari, Markus Fockel, Jens Friebe, Holger Giese, Joel Greenyer, Jutta Haupt, Christian Heinzemann, Stefan Henkler, Martin Hirsch, Jörg Holtmann, Ekkart Kindler, Sebastian Lebrig, Renate Löffler, Nazim Madhavji, Jürgen Maniera, Ahmet Mehic, Jan Meyer, Matthias Meyer, Marie Christin Platenius, Uwe Pohlmann, Claudia Priesterjahn, Jan Rieke, Wilhelm Schäfer, David Schmelter, Julian Suck, Oliver Sudmann, Matthias Tichy, Dietrich Travkin, Robert Wagner und Lothar Wendehals.

Insbesondere möchte ich den Kollegen danken, mit denen ich im Laufe der Zeit ein Büro geteilt habe: Matthias Tichy, Jan Rieke und vor allem Claudia Priesterjahn. Ihr wart immer da, um fachliche Fragen zu erörtern, meiner Motivation ein wenig auf die Sprünge zu helfen oder auch um einfach mal zu quatschen. Danke dafür!

Beim Umschiffen aller bürokratischen, organisatorischen und technischen Klip-

pen waren mir Jutta Haupt und Jürgen Maniera eine große Hilfe.

Archimetrix baut auf bereits vorhandenen Werkzeugen auf, insbesondere auf SoMoX und Reclipse. Mein Dank gilt denjenigen, die an der Entwicklung dieser Werkzeuge beteiligt waren und im Notfall zur Stelle waren um Unterstützung zu leisten: Steffen Becker, Klaus Krogmann und Benjamin Klatt, sowie Dietrich Travkin, Matthias Meyer und Lothar Wendehals.

Als wissenschaftlicher Mitarbeiter ist man stark davon abhängig, dass man Studenten findet, die in Form von Abschlussarbeiten oder SHK-Tätigkeiten an der Umsetzung der eigenen Konzepte und Ideen mit Engagement und Begeisterung mitwirken. Ich hatte das Glück, dass Markus Fockel, Aljoscha Hark, Marie Christin Platenius, Christian Stritzke, Oleg Travkin und Andreas Volk mir in dieser Hinsicht zur Seite gestanden haben.

Des Weiteren möchte ich meiner Familie Danke sagen. Das allergrößte Dankeschön geht an meine Eltern Marita und Matthias, die mir diesen Weg überhaupt erst ermöglicht haben, die mir immer Mut gemacht haben und ohne die ich heute nicht an diesem Punkt wäre. Danke auch an Andreas und Christoph, die besten Brüder der Welt, sowie an Janin, Sabrina und Pia. Auch meine Omas, Anneliese und Ursula, sowie die gesamte weitere Verwandtschaft – zu zahlreich um sie hier alle namentlich zu erwähnen – haben ihren Anteil an dieser Arbeit.

Zu guter Letzt danke ich meiner Freundin Katharina Wecker. Für das Mitmir-freuen an guten Tagen, für die Aufmunterung an nicht so guten Tagen, für das Ertragen meiner manchmal seltsamen Launen ganz besonders in der Endphase dieser Arbeit und vor allem für das “da sein”.

Contents

1	Introduction	1
1.1	Evolution of Business Information Systems	1
1.2	Reverse Engineering of Software Architectures	3
1.3	Problem Statement	4
1.4	Solution Overview	5
1.5	Application Scenarios	7
1.6	Scientific Contributions	8
1.7	Example System	8
1.8	Structure	9
2	Foundations and Related Work	11
2.1	Software Architecture Reconstruction	12
2.1.1	Terminology	12
2.1.2	Overview of the Methodology	14
2.1.3	Software Architecture Reconstruction in Archimetric	15
2.2	Pattern Detection	15
2.2.1	Terminology	16
2.2.2	Overview of the Methodology	18
2.2.3	Pattern Detection in Archimetric	19
2.3	Refactoring and Reengineering	19
2.3.1	Terminology	20
2.3.2	Reengineering in Archimetric	21
2.4	Hybrid Reverse Engineering Approaches	22
2.5	Bad Smell Detection and Removal	24
2.6	Architecture Reengineering	25
2.6.1	Architecture Conformance Checking	25
2.6.2	Architecture Migration and Modernisation	26
2.6.3	Modularisation	28
2.7	Classification of the Archimetric Approach	28
2.8	General Assumptions	29
3	Design Deficiencies	31
3.1	Types of Software Patterns	31
3.2	Describing Design Deficiencies	33
3.3	Running Example	35
3.4	Further Design Deficiencies	42
4	The Archimetric Process	45
4.1	Contributions.	45
4.2	Process Overview	46

4.3	Iterative Architecture Reconstruction	46
4.4	Design Deficiency Formalisation	49
4.5	Limitations	55
5	Influence of Deficiencies on the Architecture Reconstruction	57
5.1	Contributions.	57
5.2	Reconstruction Process	57
5.2.1	Metrics	61
5.2.2	Strategies	63
5.2.3	Dependencies Between Metrics and Strategies	64
5.3	Influence of Design Deficiency Occurrences on the Metrics	66
5.3.1	Influence of the Transfer Object Ignorance Deficiency	66
5.3.2	Susceptibility of Different Metrics and Strategies	69
5.3.3	Influence of Other Design Deficiencies Occurrences	70
5.4	Result Model	71
5.5	Limitations	72
5.6	Conclusion	73
6	Component Relevance Analysis	75
6.1	Motivation	75
6.2	Contributions.	76
6.3	Assumptions	76
6.4	Component Relevance	77
6.5	Relevance Metrics	77
6.5.1	Complexity Metric	78
6.5.2	Closeness to Threshold Metric	79
6.6	Relevance Calculation	81
6.7	Limitations	83
6.8	Related Approaches	84
6.9	Conclusion	84
7	Design Deficiency Detection	85
7.1	Contributions.	85
7.2	Assumptions	86
7.3	Pattern Detection with Reclipse	86
7.4	Integration with the Architecture Reconstruction	91
7.4.1	Input Model for the Deficiency Detection	91
7.4.2	Auxiliary Component Patterns	91
7.5	Improved Trace Collection through Symbolic Execution.	94
7.5.1	Shortcomings of Current Trace Collection	97
7.5.2	Systematic Trace Generation	97
7.5.3	Interpreting the Generated Traces	98
7.5.4	Related Approaches	99
7.6	Limitations	99
7.6.1	Limitations of the Deficiency Detection	99
7.6.2	Limitations of the Improved Trace Generation	100
7.7	Conclusion	100

8	Design Deficiency Ranking	101
8.1	Contributions.	101
8.2	Assumptions	101
8.3	Ranking Metrics	102
8.3.1	Structural Accuracy Metric	102
8.3.2	Deficiency-Specific Ranking Metrics	103
8.4	Rank Calculation	107
8.5	Limitations	107
8.6	Related Approaches	108
8.7	Conclusion	109
9	Deficiency Removal	111
9.1	Contributions.	111
9.2	Assumptions	111
9.3	Deficiency Removal Process.	112
9.4	Manual Deficiency Removal.	114
9.4.1	Removal Guides	115
9.4.2	Example	115
9.5	Automated Deficiency Removal	116
9.5.1	Removal Strategies	117
9.5.2	Behaviour Preservation	118
9.5.3	Propagating the Removal back to the Source Code	119
9.5.4	Architecture Preview	119
9.5.5	Related Approaches	122
9.6	Limitations	123
9.7	Conclusion	125
10	Validation	127
10.1	Prototype Implementation	127
10.1.1	Software Architecture	127
10.1.2	Example Session	128
10.2	Experiment Setup	136
10.3	Validation Questions.	137
10.4	Case Studies	137
10.5	Threats to Validity	138
10.5.1	Threats to Internal Validity	138
10.5.2	Threats to External Validity	140
10.6	Case Study 1: Store Example	140
10.6.1	System Overview	140
10.6.2	Validation Results	141
10.6.3	Discussion	143
10.7	Case Study 2: Palladio Fileshare.	146
10.7.1	System Overview	146
10.7.2	Validation Results	146
10.7.3	Discussion	149
10.8	Case Study 3: CoCoME.	151
10.8.1	System Overview	151

10.8.2	Reference Implementation Validation Results	152
10.8.3	SOFA Implementation Validation Results	158
10.8.4	Discussion	160
10.9	Time and Effort	163
10.10	Level-II-Validation	164
10.11	Lessons Learned	166
11	Conclusion	171
11.1	Results and Conclusions.	171
11.2	Future Research Challenges.	173
A	Meta Models	175
A.1	Generalised Abstract Syntax Tree	175
A.2	Service Architecture Meta Model	177
A.3	Source Code Decorator	180
B	Design Deficiencies	181
B.1	Interface Violation	181
B.1.1	Design Deficiency Problem	181
B.1.2	Example	181
B.1.3	Influence on the Architecture Reconstruction	182
B.1.4	Removal Strategies	183
B.1.5	Formalisation	188
B.2	Unauthorised Call	190
B.2.1	Design Deficiency Problem	190
B.2.2	Example	190
B.2.3	Influence on the Architecture Reconstruction	191
B.2.4	Removal Strategies	191
B.2.5	Formalisation	191
B.3	Inheritance between Components	193
B.3.1	Design Deficiency Problem	193
B.3.2	Example	193
B.3.3	Influence on the Architecture Reconstruction	193
B.3.4	Removal Strategies	194
B.3.5	Formalisation	194
C	Clustering Configurations	197
D	List of Abbreviations	203
	References	205
	Own Publications	221
	Supervised Theses	223
	List of Figures	225

List of Tables	229
Index	231

1. Introduction

Business information systems are one of the most pervasive, but also most diverse, classes of software in our world. These systems support the daily operation of many companies. They are used in a variety of tasks, including human-resource management, process control, and accounting.

Business information systems have to perform complex tasks, addressing a large number of requirements. Therefore, they are often large and complex pieces of software consisting of millions of lines of code. This leads to high costs in the creation and maintenance of these systems.

One way to address the complexity of these systems is the use of the *component-based software engineering* (CBSE) paradigm [SGM02]. In CBSE, systems are composed of ready-made, independently deployable software components. These components can be connected to each other and communicate via well-defined interfaces. This way, the complex functionality of the software can be structured and distributed to a number of components. Third-party components for specific tasks can be bought off-the-shelf and previously developed components can be reused.

1.1. Evolution of Business Information Systems

An important trait of business information systems is their long life-span. They are in use for many years, sometimes even as long as twenty or thirty years. Systems that are critical for the operation of a business, such as process control systems or banking software, cannot be easily exchanged or shut down. The failure of such a system might cause high financial losses. In the case of process control software, e.g. in a chemical plant, it might even incur danger to human life. Nevertheless, new requirements or the discovery of programming errors often necessitate the maintenance or extension of these crucial systems. This activity is called *software evolution* [Leh80, Art88, MD08]. Software evolution accounts for a significant part of the total cost of a software system. According to different studies, it is responsible for 40% to 90% of the total cost of software development [LS80, Gla03, Som10].

Based on these observations, Lehman phrased his famous laws of software evolution [Leh80, Leh96]. The first law states that a system “[...] must be continually adapted else it becomes progressively less satisfactory [...]” [Leh80]. The constant change in the software over time leads to a phenomenon described as *software aging* [Par94], *design erosion* [vGB02], or *architectural drift* [RLGB⁺11]. This stepwise decline of software quality is caused by different factors: changes and extensions often have to be carried out under a high time and cost pressure, the existing documentation of the software may be insufficient, or the maintenance activity may have been delegated to inexperienced

developers. Whatever the cause, design erosion leads to the fact that deficiencies are introduced into the software, the original architecture of the software gradually becomes distorted, and the source code becomes more and more cluttered and obfuscated. Reussner and Hasselbring describe this as the *'piggyback syndrome'*:

“Gerade weil es so kostspielig ist, Software-Architekturen zu modifizieren, werden diese häufig nicht angepasst und stattdessen neue Funktionalität in nicht optimaler Form hinzugefügt. Dadurch entstehen Software-Systeme mit dem 'Huckepack'-Syndrom:

- Viele Funktionalitäten wurden nachträglich, oft unter Umgehung der vorgesehenen, aber nicht ganz adäquaten Schnittstellen und durch Verletzung von Datenkapselungen, dem System hinzugefügt,
- Code wird teilweise nicht mehr genutzt oder ist in ähnlicher Form doppelt vorhanden,
- Code ist nicht in der Kompaktheit und für die Effizienz formuliert, wie es möglich wäre.”

[RH06, p. 134]¹

Design erosion, in turn, complicates future extensions: Normally, a high-quality software architecture is required for an effective reengineering. Design erosion, on the other hand, reduces the quality of a system. The quality of the software in this case is measured in terms of the compliance to generally accepted design principles. For example, system elements should exhibit low coupling and high cohesion [CK94] and components should communicate via their interfaces [SGM02]. If the existing architecture violates these principles, this impedes software architect's understanding of the system and may lead to the introduction of even more problems or bugs. It leads to a vicious circle in which bad software quality gives rise to new problems that further decrease the quality. Lehman describes this problem in his seventh law: “The quality of E-type systems² will appear to be declining unless they are rigorously maintained and adapted to operational environment changes” [Leh96].

Thus, maintenance can mitigate the negative effects of design erosion. As Bourquin and Keller argue, this does not only improve the architectural quality but it also reduces costs:

¹Translation from German: “Since it is so expensive to modify software architectures, they are often not adapted. New functionality in sub-optimal form is added instead. This leads to software systems with the *'piggyback syndrome'*:

- A lot of functionality is added to the system retroactively, often neglecting the provided, not entirely adequate interfaces and breaking the data encapsulation.
- Code is partially not used anymore or is duplicated with minor differences.
- Code is not written as succinctly and as efficiently as possible.”

²E-type systems in the sense of Lehman are systems which are part of the real world in contrast to simple algorithms.

“Solving those [architectural] problems leads to improvements that ultimately help reducing costs, including:

- Further actions of perfective maintenance are facilitated.
- Extensions of the application are easier to develop and show a lower defect rate.
- Existing components of the application are easier to reuse.”

[BK07]

A large number of different architecture maintenance and reengineering approaches exists already. Many of them require an in-depth understanding of the system. In order to detect architectural problems and design erosion, the software architect has to know the software architecture that was intended by the original developers. Many of the existing approaches like reflexion modeling [MNS01, KLMN06], expect the developer to have at least *some* kind of understanding of the system [PTV⁺10]. Sometimes, a complete architectural description is needed [TGLH00, LTC02]. If no such description is available, these techniques cannot be applied.

The architectural description on which the aforementioned approaches are based on is supposed to stem from the original development, or it has to be created ex-post by an expert. This is often an unrealistic assumption because the architecture description is insufficient or incomplete in many cases and the original developers are not available anymore [BM06].

1.2. Reverse Engineering of Software Architectures

If neither experts nor sufficient, up-to-date documentation are available, the software architect’s only starting point for understanding the system is the source code itself. However, since business information systems often consist of several millions of lines of code, it is infeasible to read all the code in order to understand the system. Instead, an automatic approach to extract information from the source code is necessary. As the architecture of the system is a good starting point for developing an understanding, it is often the first artefact that is extracted. This process is called *architecture reconstruction* [DP09] or *architecture recovery* [MEG03].

To support software architects in the task of understanding a system, many (semi-) automatic approaches have been proposed over the years. They have different goals and carry out their analyses at different levels of abstraction. Sartipi argues that “[... i]n a nutshell, the existing approaches to software architectural recovery can be classified as clustering-based techniques and pattern-based techniques [...]” [Sar03]. Reussner and Hasselbring also mention knowledge-based reverse engineering as a third technique [RH06, p. 153].

Clustering-based techniques try to reconstruct the architecture of software systems by grouping the contained entities (e.g. classes or modules) into components. The grouping is determined by some measure of similarity between these entities [Lak97]. For example, a simple similarity metric would be the

number of relations between two classes: Classes that have many relations are grouped together. Ducasse and Pollet present an overview of these approaches [DP09], and categorize them along different taxonomy axes. Due to the use of metrics, which can be efficiently measured, clustering-based approaches are usually very scalable. However, this advantage comes at the cost of abstraction. For example, the aforementioned number of relations metric does not distinguish different kinds of relations, e.g. references and inheritance. Therefore, clustering-based techniques cannot recover the intent of the clusters they reconstruct [DP09].

In contrast, *pattern-based* techniques have the goal of detecting certain recurring structures, so-called *patterns*, in the system. The rationale behind this idea is that the implementation of a certain pattern is linked to a specific design intent, the knowledge of which will enhance an engineer's understanding of the system. The most famous collection of software patterns are the *Design Patterns* which are described in the seminal book by Gamma et al. [GHJV95]. In the following years, many other pattern collections have been proposed for different domains [ACM01, BHS07, SSRB00] and different levels of abstraction [Bec07, BMR⁺96, GM05]. Dong et al. give an overview of different automatic pattern detection approaches [DZP09]. In contrast to clustering-based techniques, pattern-based approaches usually take more detailed information into account. Thereby, they can deliver more precise results but are less scalable than clustering-based techniques [SSL01, BT04a].

Knowledge-based reverse engineering attempts to recover complete semantic architecture descriptions by recognizing pre-defined clichés in control and data flow graphs. However, the automated creation of complete semantic documentation has by now been largely deemed infeasible [RH06, p. 153].

1.3. Problem Statement

The properties of business information systems discussed in Section 1.1 lead to several problems in the application of state-of-the-art reverse engineering approaches.

1. Architecture reconstruction is based on invalid assumptions for long-living business information systems As described above, business information systems have a long life-span and have to be constantly adapted and evolved. If no up-to-date architectural documentation is available, it has to be reconstructed first in order to enable a system adaptation. However, state-of-the-art architecture reconstruction techniques are based on the assumption that certain component-based design rules and principles have been adhered to during the development of such a system. For example, it is assumed that components communicate only via their interfaces. Even if this is true in the initial release of a system, software aging and design erosion can lead to the neglect of these principles.

In that case, architecture reconstruction techniques yield adulterated results because their assumptions are invalid for the system under analysis. In this

thesis, I call these architecture-related problems which influence architecture reconstruction techniques *design deficiencies*.

2. Deficiency detection does not scale without proper focus Design deficiencies such as the neglect of component-based design principles can in principle be detected through the application of pattern-based reverse engineering techniques. However, the application of such approaches to complex business information systems is impractical. Business information systems often consist of millions of lines of code but pattern-based reverse engineering approaches do not scale well [SSL01, BT04a]. Without information about the system architecture, the software architect cannot focus the deficiency detection of smaller parts of the system.

3. Pattern detection results are not manageable Another problem of pattern detection approaches is that they can yield hundreds or thousands of detection results [TSG04]. In contrast to a reconstructed architecture, which attempts to provide a coherent overview of the system's components and their connections, pattern occurrences are unconnected. Each detected pattern occurrence stands for itself and the software architect may have a hard time to distinguish the relevant occurrences from the less relevant, incomplete or incorrect ones [GD12]. Many existing approaches require an expert to validate detected problems and provide additional insight [SPL03, KOV03]. Even if such an expert is available, he has to examine the detected occurrences one by one which is very tedious.

1.4. Solution Overview

To solve the problems identified in the previous section, I propose an approach called *Archimetrix*. Archimetrix combines clustering-based and pattern-based techniques to allow for a scalable analysis of component-based software systems. This section gives an overview of the approach and explains how it solves the previously identified problems.

Figure 1.1 illustrates the reengineering process with Archimetrix. The process begins with the clustering-based architecture reconstruction in Step 1 which reconstructs an initial software architecture of the system. The architecture consists of a number of components and their connections. In Step 2, the components that were reconstructed are analysed with respect to their likelihood of containing design deficiencies. Components that are likely to contain such deficiencies are regarded as relevant for Step 3, the design deficiency detection. The pattern-based detection is executed for a relevant subset of the reconstructed components (as opposed to carrying out the detection on the whole system) which improves its scalability. The results of the deficiency detection are then analysed and ranked in Step 4, the design deficiency ranking. This step prioritises the detected deficiency occurrences with respect to their influence on the reconstructed architecture. The software architect can then decide which of those deficiency occurrences are to be removed in Step 5, the deficiency

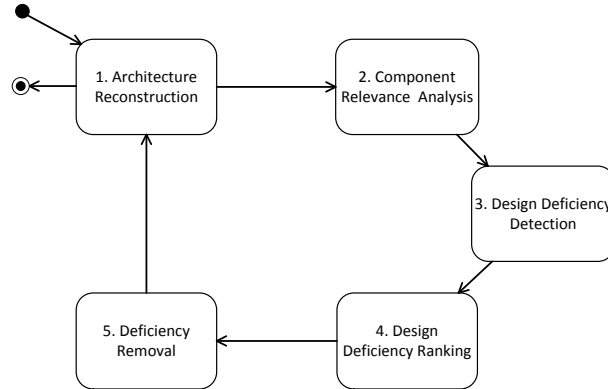


Figure 1.1.: Overview of the reengineering process with Archimetrix

removal. After the removal, Step 1 is repeated. The newly reconstructed architecture may now differ from the initially reconstructed one because the removed deficiency occurrences no longer influence the clustering. From here, the software architect can either end the process or can repeat it, thereby iteratively improving the architecture.

The suggested process solves the problems discussed in Section 1.3 as follows:

1. Reconstruction of an architectural description Archimetrix uses a standard clustering approach to reconstruct the software architecture of a given system. This allows a software architect to get an overview of a system even if no other information sources are available, e.g. an architecture documentation or the original developers.

2. Detection of design deficiencies which influence the architecture reconstruction A major problem of clustering techniques is the influence of the reconstructed software architecture by design deficiencies. Bourquin and Keller observe that design erosion and architecture violations are “[...] an architecture smell³ whose detection can largely be automated and which has proven to be key to high-impact refactorings [...]” [BK07]. However, techniques that detect such problems often require an architectural representation of the software system which is not available in many cases [BM06]. The use of clustering techniques in Archimetrix provides exactly such a representation which allows for the detection and removal of design deficiencies. Subsequent architecture reconstruction attempts may then yield a different architecture that is less influenced.

3. Relevance analysis identifies good candidate components for the deficiency detection Many pattern detection techniques suffer from scaling issues

³The authors use the term ‘architecture smell’ following the definition by Roock and Lippert. Roock and Lippert explain that smells indicate conspicuous features in a system and that architecture smells therefore may lead to extensive refactorings [RL04].

in large systems. The Archimetric process arranges for the deficiency detection to be executed *after* an initial clustering. This allows the software architect to select components from the reconstructed architecture and focus the deficiency detection to this selection. The component relevance analysis determines which components are a worthwhile input for the deficiency detection.

4. Deficiency ranking identifies the most severe problems Even if the detection scope of the deficiency detection is limited by the selection of components, the detection can possibly yield a large number of results [TSG04]. Archimetric ranks these detected deficiency occurrences according to several criteria, and thereby determines which deficiency occurrences are the most severe. The software architect can then concentrate on the removal of these critical deficiency occurrences.

1.5. Application Scenarios

Archimetric can be employed in a number of different application scenarios. This section provides an overview of how Archimetric can support them.

Architecture improvement and documentation In this main application scenario of Archimetric, the software architect wants to reconstruct an unadulterated architectural model of the system under analysis. In order to obtain this model, the architect detects and removes design deficiencies which would otherwise influence the architecture reconstruction. By executing multiple iterations of architecture reconstruction and deficiency removal, the software architect can successively improve (1) the understanding of the system and (2) the architecture of the system by removing the deficiencies. Afterwards, the architecture of the system is formally documented in the reconstructed model. This can be the basis for a number of further maintenance activities discussed in the following.

Improved modularisation Once the component-based architecture of the system under analysis has been reconstructed, single components from the created architecture can be extracted and reused in other systems [BPD12]. This can be an important use case for the reengineering of legacy systems as well as for the creation of software product lines [KK11].

Architecture reengineering When components are identified, the system can be reengineered better. For example, components can be adapted independently once their boundaries are known to the software architect. They can also be replaced by components off-the-shelf (COTS) or can be adapted to new paradigms (e.g. service-oriented architecture [OSL05, UZ09, EFH⁺11a, FHR11a]) or new technology, like cloud computing [FH10] or web technology [ACL05, Zdu05].

Creation of analysis models The reconstructed model consists of components in the strict sense of Szyperski's component definition [SGM02]. It enables the reconstruction of hierarchical component architectures. Thus,

it enables the creation of analysis models which can serve as an input for further analyses, e.g. for a performance prediction approach [BKR09].

Architecture conformance checking The reconstructed architecture can also be used for conformance checking. Once a satisfying architecture has been reconstructed, it can be used as a reference point for future development. When future extension and adaptation necessitate changes to the system, the resulting architecture can always be compared to this recovered architecture in order to check whether the changes caused the architecture to erode. This way, appropriate countermeasures, e.g. a refactoring, can be arranged. (Reussner and Hasselbring compare this scenario to construction surveillance from the engineering domain [RH06, p. 19].)

1.6. Scientific Contributions

This thesis is concerned with the following research questions:

- RQ1** Do design deficiencies that stem from the neglect of component-based design principles influence architecture reconstruction techniques?
- RQ2** Can the integration of pattern detection techniques into the architecture reconstruction process help in detecting such an influence?
- RQ3** How can relevant design deficiencies be discovered, documented, and formalised?
- RQ4** How can detected deficiencies be removed and can the influence of the removal on the architecture be predicted?
- RQ5** Can architecture reconstruction techniques be helpful in mitigating the scalability issues of pattern detection techniques?

By answering these questions, this thesis provides support in retaining high-quality software architectures in the code base of business information systems. Thereby it enables software architects to use standard architecture reconstruction techniques whose utility would be otherwise limited due to design deficiencies. This paves the way for further reengineering tasks which rely on the availability of precise architecture documentation.

Note that this thesis does not present new design principles for the development of component-based systems. The principles whose neglect is examined in this thesis are rather a selection of commonly accepted design principles from literature [ACM01, SGM02, Fow02].

1.7. Example System

In this section, I present a fictional business information system. It is a component-based trading system which will serve as an example system throughout this thesis. Figure 1.2 shows the trading system's architecture.

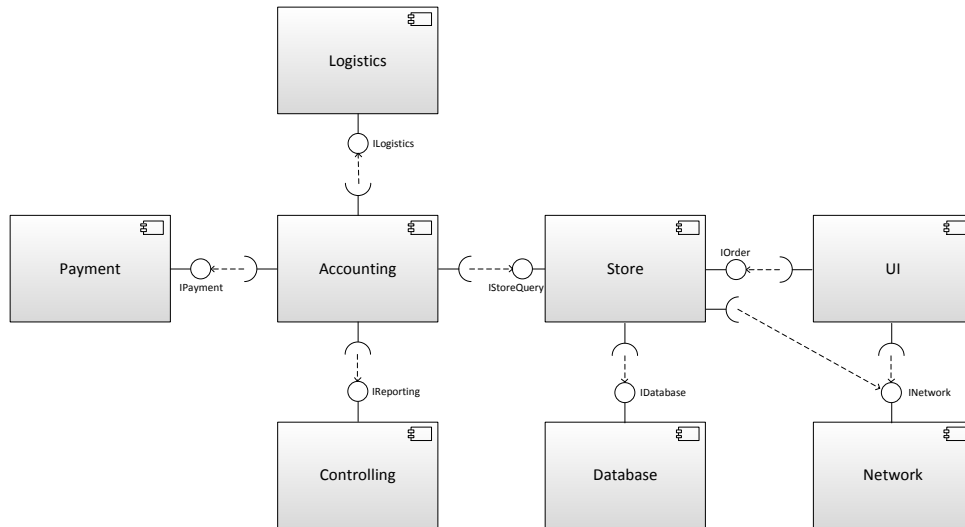


Figure 1.2.: An example business information system

The system consists of eight components. Its central parts are the two components Accounting and Store. They are complemented by components for typical tasks such as Logistics, Payment, Controlling, Database management, Network infrastructure, and user interface (UI). The architecture of the system is intentionally simplified and is only used for illustrative purposes. However, it strongly resembles the architecture of the Common Component Modeling Example (CoCoME) [RRMP08]. CoCoME also represents a component-based trading system which was designed as a benchmark for architecture analyses approaches. It is used for the validation of Archimetrix in Chapter 10.

1.8. Structure

The thesis is structured into twelve chapters. Chapter 2 lays the foundation for the remainder of the thesis, and discusses related work. Chapter 3 clarifies the notion of design deficiencies, and introduces the *Transfer Object Ignorance* design deficiency as a running example. The Archimetrix process is presented in Chapter 4. Chapter 5 explains how the architecture reconstruction approach used in Archimetrix works. It also analyses the influence of design deficiency occurrences on the architecture reconstruction with clustering-based techniques. Chapters 6 to 9 deal with the different steps of the Archimetrix process: While the component relevance analysis is treated in Chapter 6, the deficiency detection and its extensions are discussed in Chapter 7. Chapter 8 illustrates the ranking of deficiency occurrences. Chapter 9 presents the removal of deficiency occurrences. The whole approach is validated in Chapter 10. Finally, Chapter 11 concludes the thesis, and discusses possible future work.

The appendices contain technical information. Appendix A presents the var-

ious meta models that are the foundation for the analyses and transformations in Archimatrix. While Chapters 3 to 9 use one running example of a design deficiency, Appendix B contains detailed descriptions of all deficiencies used in the validation. Finally, Appendix C documents the metric weights used in the different validation scenarios.

2. Foundations and Related Work

This chapter presents the foundations of this thesis and discusses related work.

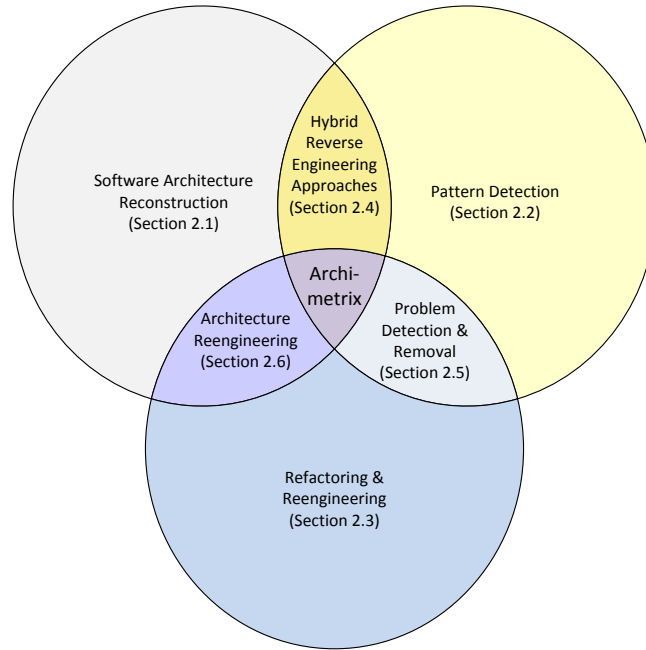


Figure 2.1.: Overview of the research areas related to this thesis

Figure 2.1 illustrates the different research areas that form the basis of this thesis: software architecture reconstruction (SAR), pattern detection, and refactoring and reengineering. Archimatrix combines these three areas and is therefore depicted at their intersection in the centre of Figure 2.1. This chapter presents the terminology and the gives a brief overview of each of this areas in Sections 2.1 to 2.3.

There is currently no approach that combines SAR, pattern detection, and reengineering as Archimatrix does. Therefore, the most closely related works lie in the intersections of two of the research areas.

There are several approaches which propose to combine SAR with pattern detection. These hybrid reverse engineering approaches are discussed in Section 2.4. As the detection of design problems in a system indicates reengineering opportunities, it is self-evident to augment pattern detection techniques with the means to remove detected problems. Section 2.5 explains these approaches. Finally, there is a lot of work that focuses on recovering the architecture of legacy systems in order to maintain, reengineer, or migrate them. Section 2.6

investigates approaches in this area, grouping them into the sub areas of architecture conformance checking, architecture migration and modernisation, and architecture modularisation.

2.1. Software Architecture Reconstruction

Taylor et al. motivate software architecture reconstruction as follows:

“A good architectural model offers the basis for maintaining intellectual control of the application. If no architectural model is available, or if the model in existence is not consistent with the implementation, then the activity of understanding the application must proceed in a reverse-engineering fashion. That is, understanding the application will require examination of the source code and recovery of a model that provides adequate intellectual basis for determining how the needed changes can be made.” [TMD09]

Consequently, an architectural model is the starting point for the understanding of a software system and also the basis for reengineering activities. Ever since the influential book by Shaw and Garlan [SG96], the topic of software architecture has attracted a lot of research interest. Therefore, a diverse vocabulary has developed over the years. The following section defines the most common architecture-related terms that are used in this thesis. Afterwards, an overview of the methodology of SAR is given, followed by a description of the integration of SAR in Archimetrix.

2.1.1. Terminology

This section presents the terminology which is used in this thesis with respect to software architecture in general and its reconstruction in particular.

Software Architecture The IEEE defines the term software architecture as follows:

“The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.” [IEE00]

It is important to note that for a given system, there is no such thing as *the* correct architecture. As there are different stakeholders with different concerns, there are also different viewpoints of the architecture. In the context of software architecture reconstruction most often a static viewpoint is assumed which is confined to the recovery of system’s components and their connections.

Software component The term *software component* has been used in many approaches, projects, and standards, yet there is no universally accepted definition what a software component is.

The definition of a software component used in this thesis is in line with the definition by Szyperski:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [SGM02, p. 34]

The components Szyperski describes are typically used in business information systems. They have clearly defined interfaces and are solely communicating via those interfaces. Interfaces define a number of operations and operations exchange data only via so-called data transfer objects, i.e. objects which only have fields of primitive types like integer. This leads to a clear decoupling of components and promotes component encapsulation, interchangeability, and reuse. It also increases the maintainability of the architecture.

There are a number of frameworks available which make components (and sometimes also their connectors) first-class entities in the software development process. On the one hand, there are commercial frameworks which were (at least for the most part) developed in industry. Examples include Microsoft’s COM [Mic12], CORBA [Obj06], and SCA [MR09]. On the other hand, there are several academic approaches, for example Palladio [BKR09], FRACTAL [BCL⁺06], and SOFA [BHP06]. A comparison of these frameworks is outside of the scope of this thesis. The survey by Lau and Wang [LW07] is a good starting point, however. It is also possible to mimic architecture concepts in languages that have no dedicated support for it (e.g. in Java).

A component is an entity of the architecture level and therefore more abstract than entities on the implementation level, like classes. A component can comprise a number of implementation level entities. In that case, it is called *primitive component*. A component can also be recursively composed of other components in which case it is referred to as a *composite component*.

Note that this notion of a component is different from the definition of components in other approaches like the one by Keller and al. [KSRP99] where each class is viewed as a component of the software. It also differs from the notion used in embedded systems where components are often active and communicate via asynchronous message-passing.

Concrete Architecture According to the taxonomy by Ducasse and Pollet, the concrete architecture of a software is the “[...] architecture that is derived from the source code [...]” [DP09]. It must be noted that this derivation process always introduces a certain degree of inaccuracy into the derived architecture. This may be due to human misjudgement, to the imprecision of automated heuristic analyses, or because of deliberate abstraction steps. Therefore, the concrete architecture can never be definitive. On the contrary, there may be a number of concrete architectures for a software system that are equally valid for different scopes, purposes, or stakeholders.

The concrete architecture is sometimes also called *as-built architecture* [OSL05], *actual architecture* [KLMN06], or *implemented architecture* [RLGB⁺11].

Conceptual Architecture The conceptual architecture is “[...] the architecture that exists in human minds or in the software documentation [...]” [DP09].

Ideally, it exactly matches the concrete architecture, e.g. because the concrete architecture is derived step-by-step from the conceptual architecture. However, more often than not the concrete architecture deviates from the conceptual architecture. This can happen for a number of reasons such as time pressure, ignorance on part of the designers, or technical limitations. Even if the initial implementation of a system adheres to the conceptual architecture, later changes may gradually lead to the concrete and conceptual architecture drifting apart (see *Architectural Drift*).

The conceptual architecture is also known as *as-designed architecture* [OSL05], *planned architecture* [KLMN06], or *designed architecture* [RLGB⁺11].

Architectural Drift According to Lehman’s laws of software evolution, software has to be changed during its lifetime in order to remain useful [Leh80, Leh96]. It has to be adapted to new requirements or a changing environment and discovered defects have to be removed. However, these changes, necessary as they may be, always incur the risk of violating the original conceptual architecture. When the original architecture was not built to cater for a new requirement, e.g. distributed execution of an application, it is very difficult to realise this requirement in the confines of the old architecture. When the architecture is consequently adapted, however, it deviates from the documented, conceptual architecture. In many cases, time pressure or simple neglect prevent the adaptation of the documentation. Over time, the concrete and conceptual architecture drift more and more apart. This phenomenon is called *architectural drift* [RLGB⁺11].

Architectural drift is also sometimes referred to as *software aging* [Par94], *design erosion* [vGB02], or *architecture degradation* [TMD09]. The term architectural drift refers to the increasing difference between conceptual and concrete architecture as a whole. According to Rosik et al., “[...] individual discrepancies [between conceptual and concrete architecture] are often referred to as violations or inconsistencies [...]” [RLGB⁺11].

2.1.2. Overview of the Methodology

Taylor et al. describe the methodology of architecture reconstruction as follows:

“A common approach to architectural recovery is clustering of the implementation-level entities into architectural elements. Based on the approach used for grouping source code entities, such as classes, procedures, or variables, software clustering techniques can be divided into two major categories: syntactic and semantic clustering.” [TMD09, p. 142f]

They distinguish *syntactic clustering* on the one hand which uses only static information that can be derived from the source code. On the other hand, *semantic clustering* also employs domain knowledge and behavioural information from the program’s execution.

Both, syntactic and semantic clustering are based on a number of metrics. Syntactic clustering could, for example, measure the coupling of two classes to

determine if they should be assigned to the same architectural element or not. In contrast, semantic clustering could analyse the number of interactions between two elements at run-time. In most architecture reconstruction approaches several metrics are aggregated to arrive at a clustering decision.

A big advantage of these metric-based clustering approaches is that metrics “[...] are known to scale up well” [DDL99]. Thus, it is possible to create an architectural overview quickly even for large systems. A downside of clustering approaches is the concept assignment problem [BMW93]. It means that “[...] even if correct and complete structural and behavioral information about the system were available to a clustering technique, a key challenge that remains is recovering design intent and rationale.” [TMD09, p. 143]. Just from assigning system elements to components, it is not necessarily clear which component plays which role in a system.

2.1.3. Software Architecture Reconstruction in Archimetrix

The reconstruction of the software architecture from source code is a central point in the Archimetrix process. Conceptually, any clustering-based SAR approach which reconstructs a component architecture of the system under analysis could be integrated into Archimetrix. A recent overview of architecture reconstruction techniques is presented by Ducasse and Pollet [DP09].

I decided to use the Software Model eXtractor SoMoX [CKK08, Kro10, KSB⁺11] which is an architecture reconstruction approach developed at the Forschungszentrum Informatik (FZI) in Karlsruhe. SoMoX uses a whole range of metrics to iteratively reconstruct the architecture of a system from its source code. SoMoX is focused on reconstructing architectural views from the static viewpoint, e.g. a repository view, containing the different reconstructed components and a service architecture view, showing how these components are statically connected in the system.

I chose SoMoX for various reasons. First, it reconstructs rigorous architectural models which follow the strict component definition by Szyperski (see Section 2.1.1). Second, expertise and support were easily available for me. On a technical level, the use of Eclipse and EMF in SoMoX allowed an easy integration with the other parts of Archimetrix.

The software architecture reconstruction process with SoMoX is explained in detail in Chapter 5.

2.2. Pattern Detection

In the context of this thesis, pattern detection means searching for occurrences of a pattern in a system. Detecting a pattern in a system can provide the architect with valuable information. If it is a “good”, desirable pattern, it may reveal part of the original developer’s intentions. If it is a “bad” solution on the other hand, it signals an opportunity for improving the software.

Similar to the previous section, this section first introduces some pattern-related terminology. Then, the general methodology of pattern detection is

explained. Finally, the application of pattern detection in Archimetric is discussed.

2.2.1. Terminology

Since software patterns have been the subject of research for nearly twenty years now, a very diverse vocabulary has developed in the community. Denier et al. attempt to give an overview of this vocabulary [DKG08]. In this section, I present the pattern-related terminology that is used throughout this thesis. It is in line with the terminology used in Reclipse, the pattern detection approach used in Archimetric (see Section 2.2.3).

Design Pattern / Architectural Pattern / Implementation Pattern Design patterns are “good” solutions to frequently recurring software engineering problems. One of the first collections of design patterns was the book *Design Patterns* by Gamma, Helm, Johnson, and Vlissides [GHJV95]. It presents 23 well-proven solutions to recurring problems in object-oriented software design. According to Gamma et al., a pattern is “[...] a solution to problem in a context” These solutions are presented using the same template which is structured into categories like problem, intent, structure, behavior, pros and cons, etc. All patterns are defined on roughly the same level of abstraction, i.e. the presented solutions are at the design or class level. Every pattern only comprises a few of classes and their relationships.

The first book in the famous *Pattern-Oriented Software Architecture* (POSA) series [BMR⁺96] took on a broader view and presented patterns at different abstraction levels. It contains patterns that represent good solutions at the architectural level, like *Pipes and Filters*, as well as design patterns and implementation patterns or *idioms*. Other books in the series contain patterns for certain domains like concurrency [SSRB00], resource management [KJ04], or distributed computing [BHS07].

Another collection of patterns at the implementation level is presented by Beck [Bec07].

AntiPattern / Bad Smell / Design Deficiency AntiPatterns were introduced by Koenig [Koe95] and made famous by Brown et al. [BMMM98]. According to the authors, an AntiPattern “[...] describes a commonly occurring solution to a problem that generates decidedly negative consequences” [BMMM98, p. 7]. In contrast to the ‘good’ patterns, AntiPatterns do not represent the solution to a problem - they *are* the problem. Nevertheless, they usually come with advice on how they can be removed. Similar to the POSA books, Brown et al. present AntiPatterns at different levels of abstraction. They include even commonly occurring problems in the management of software projects. The ‘bad’ counterpart to implementation patterns on the very low level of abstraction are bug patterns (e.g. [All02]).

The term *bad smell* was introduced by Fowler in his book on refactoring. Bad smells give “[...] indications that there is trouble that can be solved by a refactoring” [Fow99, p. 75]. In contrast to Anti Patterns, bad smells are only

indications instead of concrete problems. Bad smells are intentionally characterised by fuzzy statements like “a class has too many responsibilities” or “a method has too many parameters”. Fowler explicitly leaves the decision if action should be taken to “[...] informed human intuition” [Fow99, p. 75]. An important point to note about bad smells according to Fowler is that they can all be removed by refactorings. Refactorings “[...] do not alter the external behavior of the code yet improve its internal structure” [Fow99, p. xvi]. Consequently, the bad smells are at a low level of abstraction which allows for such statements. An overview on state of current knowledge about bad smells given by Zhang et al. [ZHB11].

In general, Archimetric can detect arbitrary patterns in source code (see Chapter 7). The focus in this thesis, however, lies on the detection of patterns that influence the architecture reconstruction. In order to distinguish these patterns from the very general AntiPatterns and from the fuzzy notion of bad smells, they are referred to as *design deficiencies* in this thesis. Design deficiencies are violations of component-based design principles, addressing elements on both levels of abstraction: the class level *and* the component level. One example is the call of a method which is not explicitly made available through an interface (*Interface Violation*, see Appendix B.1). As such, design deficiencies are not “bad” designs, they do not impact the functionality of the system. Calling a method is not forbidden, after all. But in combination with component-based design principles (components must communicate via their interfaces), a method call may not always be allowed. An architecture reconstruction approach which relies on these principles may be influenced by the presence of such design deficiencies (see Chapter 5).

Pattern Description / Deficiency Description A pattern description is the presentation of a pattern that is meant to explain the idea behind and the relevant aspects of a pattern to a human reader. Pattern descriptions can be found in all textbooks that introduce new patterns, e.g. [GHJV95, BMMM98, BMR⁺96]. Pattern descriptions usually make use of templates that are divided into parts like intent, consequences, and implementation. They use prose text, (sometimes informal) diagrams, and examples to illustrate the pattern and its application. In particular, pattern descriptions are unsuitable for automated processing, e.g. in pattern detection approaches, as they lack a formal foundation.

Pattern Formalisation / Deficiency Formalisation A pattern formalisation is a representation of a pattern with the goal to make it automatically processable, e.g. for their automated detection. A pattern formalisation is usually derived from a pattern description by an expert. It often only reflects a subset of the information given in the pattern description. This subset contains the elements that can be detected in the software such as the structure and behaviour of a pattern. Many different approaches have been presented to formalise pattern descriptions. The book by Taibi gives an overview of some of them [Tai07].

Pattern Candidate / Deficiency Candidate A pattern candidate is a part of a software system which is identified as the occurrence of a pattern by an automated pattern detection mechanism. As the detection mechanism is based on the pattern formalisation which only reflects a part of the pattern description, a candidate may actually be detected incorrectly. For example, a candidate may exhibit the same structure as the pattern formalisation but an inspection reveals that it was not intended to be an implementation of that pattern. In this case, the candidate is a *false positive*. On the other hand, a correctly detected pattern candidate is called a *true positive*.

Pattern Occurrence / Deficiency Occurrence A pattern candidate which is a true positive is also called a pattern occurrence. In literature, it is also known as *pattern instance* or *pattern implementation*.

Pattern Role Patterns consist of different elements which have different responsibilities. These responsibilities are also called pattern roles and provide a meaningful term to refer to an element and its responsibility. For example, the famous *Observer* pattern basically consists of two basic roles [GHJV95, p. 293ff]: A subject which has a state and an observer which can register with the subject and which is notified whenever the state of the subject changes. These roles can also be refined, for example by stating that there are two classes who play the roles of the abstract subject and the abstract observer, defining the corresponding interfaces for the interaction. In addition, there can be arbitrarily many classes which play the role of either a concrete subject or a concrete observer, implementing specific behaviour. Roles cannot only be played by classes but also by other elements of the system, e.g. by methods.

When a pattern occurrence is detected, its different elements are annotated with the roles that they play in that pattern. This allows the software architect to understand the responsibilities of all the different elements, when he inspects the detected pattern occurrences.

2.2.2. Overview of the Methodology

In order to detect pattern occurrences in a software system, a pattern detection mechanism and pattern formalisations are needed. Although, the formalisations can be hard-coded into the detection mechanisms, most approaches allow to specify them separately. Patterns are usually formalised with domain-specific languages. Taibi presents a selection of different pattern formalisation techniques [Tai07].

Pattern detection approaches can be classified by the type of information they use for the detection. Some only take the static structure of the code into account: The source code is parsed and patterns are detected based on the formalisations of their structural properties (e.g. classes and their relationships). This method is called *static analysis* or *structural analysis*. Other approaches also formalise the expected behavior of patterns and analyse the software's run-time behaviour accordingly. For this, execution traces have to be collected which can then be compared to the expected behaviour of each candidate. This

method is called dynamic or behavioural analysis. Because patterns can be quite similar (e.g. the State and the Strategy pattern [GHJV95] are structurally equivalent), some approaches combine static and dynamic analysis methods. They use a structural analysis to identify pattern candidates and analyse the candidates' behaviour afterwards to identify the true positives.

A general problem of pattern detection approaches is the scalability. Because very fine-grained information is considered during the detection, the analysis of large systems can take hours or days [SSL01, BT04a]. In addition, the number of detection results may become unmanageable for large systems. A system with millions of lines of code may contain hundreds or thousands of pattern occurrences [TSG04]. In that case, the value of the pattern detection decreases rapidly because the software architect cannot manage the large number of results effectively.

2.2.3. Pattern Detection in Archimatrix

In Archimatrix, pattern detection is used to identify design deficiencies, i.e. violations of well-known design principles and rules. Similar to software architecture reconstruction, any pattern detection approach could be used in Archimatrix, conceptually. The only requirement is that it allows for the specification of the patterns to detect. Dong et al. compare different pattern detection approaches considering aspects like analysed information, data representation, or analysed systems¹ [DZP09].

I decided to integrate Reclipse into Archimatrix. Reclipse was developed in the Software Engineering Group at the University of Paderborn under my participation [vDMT10b, vDMT10a, vDT10]. It uses graph matching for the structural detection of user-specifiable patterns in abstract syntax graphs [NSW⁺02, Nie04]. The static properties of the design patterns or design deficiencies to be detected are formalised as graph patterns. Reclipse then tries to find isomorphic matches for these graph patterns in the abstract syntax graph of the system under analysis. These matches are then presented to the user as pattern/deficiency candidates. Reclipse also provides the means to execute a behavioural analysis which takes run-time information into account [Wen04, Wen07].

Reclipse is described in more detail in Chapter 7.

2.3. Refactoring and Reengineering

This section begins with an overview of the terminology concerning refactoring and reengineering. As this area is more diverse than architecture reconstruction or pattern detection, no general methodology can be identified. Section 2.3.2 discusses the application of refactoring and reengineering techniques in Archimatrix.

¹Dong et al. refer to the detection of pre-defined patterns as “pattern mining” whereas the term “pattern detection” is used in this thesis.

2.3.1. Terminology

The terminology presented in this section is largely based on the definitions by Chikofsky and Cross [CC90]. Therefore, Figure 2.2 which relates the different terms in the field to each other is adopted from their publication.

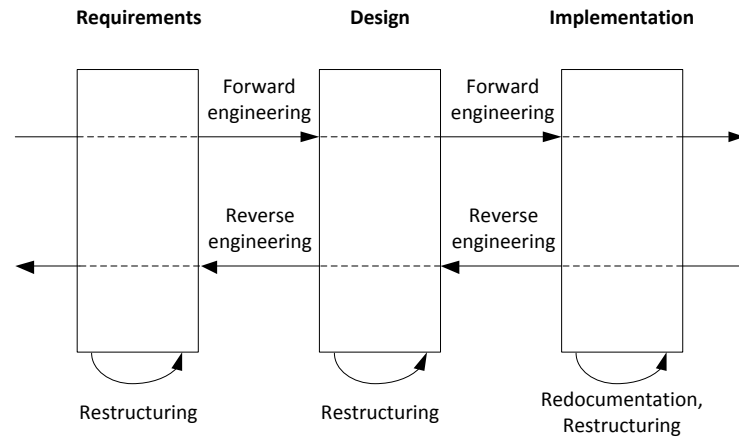


Figure 2.2.: Reengineering terminology (Figure adapted from [CC90])

Chikofsky and Cross identify three phases in the development of software: (collection of) requirements, design, and implementation. In classical software development, these phases occur exactly in that sequence which is called *forward engineering* by the authors. In contrast, they call the reversed process, i.e. going from the implementation back to the requirements, *reverse engineering*. Changes within one of these phases are termed *restructuring*. This use of the term reengineering was later adopted by Demeyer et al. [DDN03]. It deviates, however, a little from the definition by Sommerville who sees reengineering as an activity that “[...] takes place after a system has been maintained for some time and maintenance costs are increasing” [Som10]. In Sommerville’s definition, the initial construction of the system is not contained in the reengineering life cycle.

Reengineering According to Chikofsky and Cross, “[...] reengineering [...] is the examination, and alteration of a subject system to reconstitute it in a new form the subsequent implementation of the new form” [CC90]. Therefore, reengineering is the combination of reverse engineering (in order to understand the system) and forward engineering or restructuring (in order to change it). A related notion in this area is *software evolution* which describes the reengineering of software as the consequence of software aging [Leh80, Art88, MD08].

Reverse engineering “Reverse engineering is the process of analyzing a subject system to identify the system’s components and their interrelationships

and create representations of the system in another form or at a higher level of abstraction” [CC90]. Following this definition, both architecture reconstruction and pattern detection fall into this category. Reverse engineering is strictly non-invasive, i.e. the system is not changed but only examined.

Forward engineering “Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system” [CC90]. Therefore, forward engineering is the ‘normal’ activity of software engineers. Reverse engineering, in contrast, is a more specialised activity that only happens in some projects. In the context of reengineering, the term forward engineering is also sometimes used to describe just the modification of an existing system.

Restructuring / Refactoring Chikofsky and Cross state that “restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behavior” [CC90]. This is in line with Fowler’s definition of the term *refactoring*: “Refactorings do not alter the external behavior of the code yet improve its internal structure” [Fow99, p. xvi].

However, while ‘refactoring’ is perceived to be a behavior-preserving change in literature, the term restructuring is also used to describe large-scale modifications that may also alter or extend the system’s behavior. This is also called *modernisation* in some cases [vHFG⁺11].

Removal Strategy The removal of AntiPatterns or bad smells is sometimes called “refactoring” in literature. This means refactoring in the sense that the software structure is improved but the externally visible behaviour is preserved. In other cases, this constraint is a little more relaxed such that “in most cases, the goal is to transform the code without impacting correctness” [BMMM98].

In the process described in this thesis, the focus is on the reconstruction of a software architecture. Thus, the main goal is to disentangle the possibly complex interrelations between classes and components to obtain a clear and structured representation of the architecture. Therefore, we not only allow reengineering operations that leave the behavior untouched. In contrast, we explicitly include *removal strategies* that may change the behavior but may nevertheless improve the system’s structure, e.g. because they remove an illegal method call between two components. Of course, the software architect has to be aware of this and has to restore the system behavior if necessary.

2.3.2. Reengineering in Archimetric

In Archimetric, reengineering comes into play to remove previously detected design deficiency occurrences. On the one hand, deficiency occurrences can be removed by automated transformations. On the other hand, this can also be accomplished by manually changing the source code.

Automated deficiency removal is convenient from the software architect’s point of view. Executing a pre-defined transformation can remove a deficiency

occurrence quickly and reliably and the architect does not have to know the specifics of the removal strategy. In this thesis, story diagrams are used for this purpose [vDHP⁺12] (see Chapter 9). On the other hand, the transformations have to be designed by an expert (possibly the same expert who formalises the deficiencies) and have to be provided in a library together with the deficiency formalisations. Only a limited amount of transformations can be provided and it may be that none of them matches the software architect’s requirements. In this case, a manual removal of the deficiency occurrence is inevitable.

Manually removing deficiencies from the source code is more tedious and error-prone than applying automatic transformations. But it allows for more flexibility since the software architect can also apply removal strategies that were not provided in a library.

2.4. Hybrid Reverse Engineering Approaches

This section presents approaches that combine architecture reconstruction techniques with pattern detection. Their focus is not on the reengineering of the system or the removal of deficiencies.

Demeyer et al. present Code Crawler which is a hybrid reverse engineering approach that combines metrics with program visualisation techniques [DDL99]. The authors demonstrate how well-known metrics like lines of code or number of methods can be visualised using different types of graphs. They argue that this helps the developer to get a better overview of the system and, for example, identify problems more quickly. the focus on metrics makes their approach very scalable.

Tzerpos and Holt note that structural properties of the system under analysis should be considered during clustering [TH00]. Therefore, they define a number of “subsystem patterns” which are detected by their clustering algorithm. These patterns are hard-coded and are not meant to be extended by the user of their clustering tool. They do not consider deficiencies which could influence the clustering.

Mancoridis et al. present a web-based portal site which provides a number of reverse engineering tools to its visitors. Users can upload their own source code and select different analysis methods such as clustering, code browsing, code metrics, or visualisation [MSC⁺01]. They deliberately do not suggest a specific process for the combination of the tools as they want to leave this decision to the user.

Sartipi [Sar03] uses data mining techniques to structure a graph representation of a program. Then he defines architectural patterns (or, as he calls them, queries) on the resulting graph which are evaluated by graph matching. The queries are focused on simple architectural properties like the number of relations to a certain component and are not as expressive as the structural patterns used in Archimetrix.

Bauer and Trifu [BT04a, BT04b] use a combination of pattern detection and clustering to recover the architecture of a system. They detect so-called “architectural clues” with a Prolog-based pattern matching approach and use these

clues to compute a multi-graph representation of the system. The weighted edges in this representation indicate the coupling of the system elements and are used by a clustering algorithm to obtain an architecture of the system. In contrast to Archimatrix, the clustering is completely based on the information gathered by the pattern detection. Thus, the pattern detection has to be carried out first which can take very long for large systems. Archimatrix applies the clustering first to reduce the search space for the pattern detection. In addition, Bauer and Trifu focus on the detection of design patterns and do not consider the impact of bad smells on the clustering.

Han et al. present preliminary results on a similar approach [HWY⁺09]. They also detect design patterns to improve the clustering of source code. Because they also apply the pattern detection first, it stands to reason that they suffer from the same drawbacks as Bauer and Trifu.

Basit and Jarzabek [BJ05] identify clone patterns in programs and then apply a data mining approach to cluster clones which occur together frequently. However, they apply the clone detection and the clustering consecutively and do not consider relation between the two parts nor do they suggest multiple iterations of their approach. The detection of pre-defined patterns is not addressed.

Lung et al. [LXZS06] do not use clustering techniques to group related classes into components. Instead they try to identify functionally cohesive sections of long functions to find restructuring opportunities at the function level. Thus, they apply a reverse engineering technique that is usually employed at the architectural level to a lower level of abstraction. The possible restructurings are only suggested but cannot be carried out automatically. Archimatrix combines architecture reconstruction and pattern detection which operate at different levels of abstraction.

Binkley et al. present the concept of *Dependence Anti Patterns*, dependence structures in source code which may have negative effects on program comprehension, maintenance, and reverse engineering [BGH⁺08]. They define a set of seven dependence anti patterns and use a combination of program slicing and metric analysis to detect them. They do not consider the removal of these anti patterns. The authors explicitly state that they do not investigate the precise influence of the anti patterns on the aforementioned software engineering tasks. In contrast, this thesis analyses the impact of design deficiencies on the architecture reconstruction in Chapter 5.

Similar to our approach, Arcelli Fontana and Zanoni [AFZ11] use an AST representation of a system as a common basis for pattern detection and architecture reconstruction. However, they use the two techniques independently of each other but do not combine them.

Klatt and Krogmann sketch a process for the tool-supported creation of software product lines from different software systems that share common source code [KK11]. They propose to use SiSSy and SoMoX for the recovery of the software architecture and combine this with other code analysis techniques such as clone detection and code history analysis. This is related to the idea of using fine-grained analysis techniques like pattern detection to improve the architecture reconstruction. Their focus is on the extraction of product lines and not on the reengineering of the existing systems.

The Massey Architecture Explorer is a tool developed in the research group of Jens Dietrich at Massey University [Mas12]. It analyses Java systems and can visualise the dependencies between their different constituent Jar files, between packages, or classes. Each Jar file is viewed as a component. In addition, the tool is able to detect a number of AntiPatterns, e.g. strong circular dependencies between the archives. The tool does not use clustering techniques to reconstruct components from the different source code artefacts.

Keller et al. [KSRP99] describe an approach to detect “design components” in source code through “pattern-based reverse engineering”. However, they refer to a design component as “a package of structural model descriptions together with informal documentation, such as intent, applicability, or known-uses.” Hence, in the terminology used in this thesis, they detect design *patterns* rather than components. In contrast, the components recovered by Archimetrix are in line with the more rigorous component definition by Szyperski [SGM02].

2.5. Bad Smell Detection and Removal

This section discusses approaches which combine pattern detection with refactoring or reengineering techniques. In these cases, the pattern detection is used to detect bad smells, AntiPatterns, or other deficiencies. None of the approaches in this category is concerned with the reconstruction of the software architecture.

Tourwé and Mens detect “refactoring opportunities”, identify matching transformations and execute them automatically [TM03]. The bad smells and refactorings considered in their work are at a very low level of abstraction (e.g. identifying and removing obsolete parameters). The impact of the refactorings on the system architecture is not in their focus.

Tahvildari and Kontogiannis present a classification of design flaws in the intersecting categories of structural, architectural, and behavioural flaws [TK03]. They use metrics to measure, for example, the coupling and cohesion in a system in order to find those flaws. Then, they correct these deficiencies with appropriate meta-pattern transformations and re-evaluate the metrics. They argue that applying the transformations improves the metric values and therefore the quality of the system.

Trifu et al. detect and remove design flaws with respect to a user-selected quality goal, e.g. performance [TSG04]. They detect those flaws by using graph matching in combination with basic metrics. However, the authors point out that this leads to a large number of detection results which have to be manually validated by the user. Archimetrix provides an automatic deficiency ranking which facilitates the validation of detection results.

Stürmer et al. model guideline violations in MatLab models and their automatic correction [SKSS07]. For the detection, they use Reclipse, the same pattern detection approach that is integrated into Archimetrix. Thus, they have the same scalability problems as, e.g. Simon et al. [SSL01] or Bauer and Trifu [BT04a].

In his PhD thesis, *Meyer* [Mey09] presents an approach to identify bad smells

with a structural analysis and remove those deficiencies with automated graph transformations. In addition, he proposes to use inductive verification to prove that the application of the transformations does not introduce new deficiencies or violate certain constraints. However, he does not take the software architecture into account. Meyer's approach could be integrated into Archimetric in order to validate that the removal of design deficiency occurrences with respect to pre-defined constraints.

Arendt et al. [AKM⁺11] present a quality assurance process which uses a combination of metrics and structural patterns to identify model smells. They combine the smells with pre-defined graph transformations to provide an automated refactoring for identified model smells. They use their approach for the quality assurance in an industrial context, so they assume that the analysed architectural models already exist and do not have to be reconstructed.

Shah et al. detect strong circular dependencies in a package hierarchy and try to remove them by suggesting class movements [SDM12]. They use metrics to gauge the impact of moving the suggested classes on the package structure. Although their reengineering goal is architecture-oriented, they do not reconstruct an architectural view in their work.

2.6. Architecture Reengineering

The field of architecture reengineering is very diverse. However, the large number of approaches can be classified by their goal. Accordingly, this section is split into subsections dealing with architecture conformance checking approaches, architecture migration, and modularisation.

2.6.1. Architecture Conformance Checking

Knodel et al. [KLMN06] present an integrated process for the creation and evaluation of software architectures for software product lines. For that, they combine product line development approach PuLSE with the architecture- and domain-oriented reengineering technique ADORE. ADORE employs reverse engineering as well as renovation and extension techniques for the assessment and integration of existing components into the architecture. However, these steps are only mentioned at a high level of abstraction and are not explained in detail. The authors' focus lies on the comparison of the conceptual architecture to the concrete architecture produced by their approach.

Bourquin and Keller present an approach that is focused on manual refactorings on the architecture level [BK07]. First, they manually create a target architecture for their legacy software. Then, they assign existing packages to the target architecture and detect the architecture violations (i.e. accesses that violate the target architecture) in the concrete architecture. They analyse the relevance of their refactorings on the architecture after the application of those refactorings by using code metrics and a comparison between the number of detected bad smells before and after the refactoring. They do not reconstruct an architectural model automatically.

Rosik et al. investigate architectural drift, i.e. the growing distance between conceptual and concrete architecture, in a case study of an industrial software system [RLGB⁺11]. They use a tool-supported reflexion modelling approach [MNS01] to discover discrepancies between the conceptual and the concrete architecture and discuss them with the developers. Like all reflexion modelling approaches, they require someone (in their case the original developers) to have an idea of the system’s conceptual architecture. In Archimatrix, a general assumption is that no knowledge about the conceptual architecture is available.

Christl et al. use clustering to automatically establish the mapping between the manually created, hypothetical architecture and the source code [CKS07]. Thereby, they mitigate the main drawback of many reflexion modelling approaches, i.e. the high manual effort of creating such a mapping up-front. They do not consider the impact of deficiencies on the clustering results.

Sonargraph-Architect [Son12] (formerly known as SonarJ) is a commercial tool which also employs a classical reflexion modelling approach. The software architect can define a conceptual architecture in the tool. Afterwards, Sonargraph-Architect can check if the concrete architecture complies with the conceptual architecture. Refactorings are offered to fix rule violations.

The three commercial products Structure101, Restructure101, and Structure101build [Str12] form a tool suite with similar capabilities as the Sonargraph-Architect. A architect can define a conceptual architecture and communicate it to the development. The concrete architecture can be analysed, restructured, and architectural rules and constraints can be specified and enforced. Similar to reflexion modelling, both tool suites require up-front knowledge about the system under analysis to define the conceptual architecture.

2.6.2. Architecture Migration and Modernisation

Krikhaar presents an architecture improvement process [Kri97] in which an “ideal” architecture is constructed manually. The existing software is then analysed regarding import relations, part-of hierarchies and use relations at code level. This can partly be done automatically. The ideal and the “reverse-architected” architecture are then to be compared manually to identify violations. Actions to remove violations are not discussed in the paper. The author also suggests to incorporate code metrics in future work.

In follow-up work, Krikhaar et al. present a two-phase process for the improvement of software architectures [KPS⁺99]. Here, a model is generated from code. The architect has to manually evaluate this model and think of ideas to improve it. Metrics can also be used in this step although they are not discussed by the authors. Then, “recipes” to apply the ideas to the code are created manually. Finally, the architect should implement automatic transformations for the created recipes. The technique to do this seems to require manual code annotations. The impact of the improvement ideas can be evaluated on the model by using metrics or “box-and-arrow” diagram visualisation. In our approach, the improvement opportunities are automatically identified by detecting the bad smells. Automated transformations can be provided for the bad smells and an automated architecture prediction can analyse and visualise

the impact of the transformation.

Seacord et al. present the “risk-managed modernization (RMM) approach” for the modernisation of legacy systems [SPL03, p. 27ff]. Similar to Archimetric, their process also begins with the reconstruction of a legacy system’s architecture. Taking a broader view however, it examines a more general approach than Archimetric. RMM begins with the identification of stakeholders and their requirements and ends with the creation of a modernisation plan and a resource estimation step. The single steps are described at a higher level of abstraction than in this thesis and are described from a management perspective.

Bianchi et al. [BCM03] present a process to iteratively reengineer a complete systems without shutting it down. The process supports the iterative migration of functionality and data. The authors propose to first break down a system into components that can then be reengineered individually. In contrast to the approach in this thesis, neither the identification of components nor the reengineering is the focus of their work. Both steps are assumed to be carried out manually.

Frey and Hasselbring [FH10] present the CloudMIG approach, which provides a process to reengineer legacy applications for the cloud. While this is related to the motivation for my work, the authors emphasise aspects like resource efficiency and scalability of the target architecture. Although the recovery of the original architecture is mentioned in their work, it is not focused on. During the migration, the CloudMIG approach can be used to identify elements of the target architecture that violate constraints of the cloud environment. In contrast, our approach concentrates on revealing deficiencies that do not stem from the migration but from the long-term architecture erosion of the system. Frey and Hasselbring also use metrics to assess the quality of their target architecture. As they do this before the final system is generated this could be seen as related to the architecture preview step in Archimetric (see Chapter 9).

SOAMIG [EFH⁺11a, EFH⁺11b] is an iterative, generic process model for the migration of legacy code to a service-oriented architecture. Similar to our approach, the process contains reverse engineering steps for legacy code analysis and refactoring steps for the improvement of the migrated system. In addition, the authors try to automate as much of the migration as possible. However, as their approach is more generic, they do not specifically define how, e.g. the target architecture for the migration should be obtained. They also do not focus on the detection of deficiencies in the legacy system. One application of the SOAMIG process model is a tool suite for the migration of legacy Java and COBOL system towards service-oriented architectures [FHR11a, FHR11b]. Archimetric could perhaps be incorporated in another specific instance of the SOAMIG process model.

The DynaMod project [vHFG⁺11] aims at the modernisation of long-living software systems. For that, static and dynamic analyses in combination with expert knowledge are used to create an architectural representation of the system. These architectural models are then transformed into the target architecture. Template-based code generation is employed to create wrappers and connectors for the target architecture. The approach does not focus on the detection and removal of deficiencies in the legacy system.

2.6.3. Modularisation

Dietrich et al. [DYM⁺08] present BARRIO, a clustering tool which supports a developer in analysing how a legacy system can modularised. For that the system's classes are clustered and the tool detects two anti patterns: 1) a cluster containing elements from multiple packages and 2) packages which are distributed over multiple clusters. In the former case, this is seen as an indication to merge several packages. The latter case is interpreted as a situation in which a package should be split up. The actual reengineering is not covered by the approach but is left to the user as a manual task.

Sarkar et al. [SRK⁺09] report on their endeavour to break up a legacy banking application into components. The presented process includes the creation of a modular architecture, semi-automatic identification of architecture violations, a completely manual refactoring step and manually implemented checks ("gatekeeper tools") to enforce compliance with the reengineered architecture. Archimetrix provides substantially more support to the software architect through automated process steps such as the deficiency ranking or the architecture preview.

Dietrich et al. [DMTS10] investigate the influence of certain patterns on the decomposability of Java programs. Their studies show that there are a number of anti patterns which make it hard to break up an application into modules. They present an algorithm to identify these problems and discuss their removal briefly. However, they do not present a complete tool-supported process for the detection and removal of those problems. They also do not use architecture reconstruction techniques.

2.7. Classification of the Archimetrix Approach

This section places Archimetrix in two classifications of software maintenance approaches. Lientz and Swanson [LS80] take on a very general view of software maintenance techniques. Knodel et al. [KLMN06], on the other hand, list ten purposes of software architecture evaluation. Here, I relate them to the application scenarios of Archimetrix presented in Chapter 1.

Lientz and Swanson identify four different types of software maintenance: adaptive, perfective, corrective, and preventive [LS80]. Adaptive maintenance aims at modifying a system in order to adapt it changes in the system's environment, e.g. the operating system. Perfective maintenance is concerned with implementing new functionality that satisfies new or changed user requirements. Corrective maintenance deals with the diagnosis and correction of software errors while preventive maintenance increases the software maintainability in order to prevent future problems. Following this classification, Archimetrix supports corrective maintenance and preventive maintenance. Corrective maintenance is supported in that Archimetrix allows to detect and remove design deficiencies. In doing so, Archimetrix enables the architect to improve the quality of the code and the quality of the recovered architecture. On the other hand, the improved understanding of the software architecture and the removal of design deficiencies can prevent further design erosion and therefore increase

maintainability.

Knodel et al. enumerate ten different purposes of software architecture evaluation [KLMN06]. Three of them fall into the application scenario of “improved modularisation” explained in Section 1.5. These are: the identification of potential for the creation of a product line across different systems; the alignment of a system with an existing product line; and the identification of components that can be reused within an existing system. Three purposes are in line with the “architecture improvement and documentation”. Knodel et al. identify: the comprehension of software systems; the re-documentation of software systems; and the traceability from the architecture to the source code. Yet another two purposes are subsumed under the application scenario of “architecture conformance checking”. The authors call them: the assessment of consistency between documentation and implementation and the control of software evolution in the sense that the implementation does not deviate from the conceptual architecture. The two final purposes mentioned by Knodel et al. are the assessment of component adequacy and the identification of un-documented architectural entities. These tasks are not per se among Archimetrix’s application scenarios.

2.8. General Assumptions

Use of a single programming language Large business information systems usually consists of several parts that are often implemented in different programming languages. Although the parser integrated in Archimetrix, SISSy [Sis11], supports Java, C++, and Delphi code, it cannot construct a coherent model from system parts in different languages. This thesis thus does not consider this problem but assumes that the system under study is programmed in a single programming language.

Use of an object-oriented language I assume a scenario in which an *object-oriented* programming language is used to develop component-based systems. For the purpose of the examples in this thesis and for the implementation of the prototypical tool suite, I assume a system that was implemented in Java without the use of component frameworks such as Java EE [Jav12] or Fractal [BCL⁺06]. However, Archimetrix in general is not dependent on the use of a concrete programming language as it operates at the level of a program’s abstract syntax tree. Therefore, my approach can be modified to be applicable to any language for which the architect is able to extract the abstract syntax tree.

Availability of architecture documentation In this thesis, I assume that no documentation of the software system under analysis is available. This is either the case when the documentation is so outdated that it has become useless or when it has not been created in the first place. This also implies that no formal models of the system exist. Therefore, the only reliable source of information about the system is the source code itself.

3. Design Deficiencies

Design deficiencies stem from the neglect of principles and guidelines that are meant to promote good component-oriented design. They are a component-oriented analogy to bad smells which represent the neglect of rules that should promote good object-oriented design [Fow99]. In this thesis, I selected four design deficiencies as examples to illustrate my concepts and to serve as example deficiencies in the validation: they are called *Transfer Object Ignorance*, *Interface Violation*, *Unauthorised Call*, and *Inheritance between Components*. I derived these deficiencies from Szyperski's principles of component-oriented design [SGM02]. In general, there are many more design deficiencies. A structured process for their discovery, documentation, and formalisation is presented in Section 4.4.

This chapter gives an introduction to design deficiencies. It begins with a discussion of the different types of software patterns (and deficiencies as 'negative' patterns) in Section 3.1. Section 3.2 then presents and explains a template for the description of design deficiencies. Finally, Section 3.3 describes the *Transfer Object Ignorance* deficiency which is used as a running example throughout this thesis. The other three example deficiencies are explained briefly in Section 3.4. More detailed descriptions and formalisations can be found in Appendix B.

3.1. Types of Software Patterns

Software patterns can be categorised in a number of ways. On the one hand, a pattern can describe either a 'positive' situation that is desired or a 'negative' situation that should be avoided. The former are often subsumed under the term *design patterns*. The latter go by a variety of names such as *bad smells* or *anti patterns*. In this thesis, I call them design deficiencies. See Section 2.2 for a discussion of the terminology.

Another possible categorisation axis deals with the generality of a pattern. While some patterns can be regarded as 'universally applicable', others are written with a specific programming language, a specific technology, or even a certain project in mind. Table 3.1 gives an overview and examples of the different categories.

In Table 3.1, the more universally applicable patterns are listed in the top rows. The lower in the table, the narrower the focus of a pattern. For instance, Gamma et al.'s design patterns [GHJV95] or Martin's guidelines in his book "Clean Code" [Mar09] are intended to promote good object-oriented design. In both cases, the presented rules and recommendations are general and are universally applicable to every programming language and every project. The only restriction is the underlying development paradigm, i.e. that they are targeted

	Positive	Negative
Paradigm	Design Patterns [GHJV95], Principles of good component-oriented design [SGM02], Clean Code [Mar09]	AntiPatterns [BMM98], Bad Smells [Fow99]
Domain	Patterns for Concurrent and Networked Objects [SSRB00], Patterns for Resource Management [KJ04], A Pattern Language for Distributed Computing [BHS07]	More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot [SW03], A Process to Effectively Identify Guilty Performance Antipatterns [CMRT10]
Technology	Core J2EE Patterns [ACM01], Patterns of Enterprise Application Architecture [Fow02]	Modeling Guideline Violations in Simulink and Stateflow [SKSS07]
Programming Language	Java Code Conventions [Jav99], Checkstyle guidelines [Che11]	Bug Patterns in Java [All02], FindBugs bug patterns [Fin12]
Company	Company-specific guidelines and best practices	Company-specific worst practices
Project	Project-specific guidelines and best practices	Project-specific worst practices

Table 3.1.: Examples of positive and negative patterns at different levels of generality

at object-oriented software. Similarly, the AntiPatterns described by Brown et al. [BMMM98] and Fowler's bad smells [Fow99] are very general.

In contrast, there are more specific patterns. They depend on a certain technology, e.g. the use of Enterprise Application architectures [Fow02], or on a programming language¹, e.g. Java [Jav99]. Furthermore, patterns may be specific for a given company or even a given project.

Design deficiencies often cut across the different abstraction levels. In general, design deficiencies represent violations of principles of component-based design which is located at the paradigm level. The Transfer Objects Ignorance deficiency presented in Section 3.3 is based on the use of transfer objects. Transfer objects are presented as positive technology-specific patterns by Alur et al. [ACM01, p. 415] and Fowler [Fow02, p. 401]. The combination with the adherence to a specific naming scheme, however, adds a project-specific aspect to it. In this case, it is specific for the CoCoME project [RRMP08].

3.2. Describing Design Deficiencies

For the description of patterns and AntiPatterns, Brown et al. advocate the use of templates to give patterns a “consistent rhetorical structure” and to assure that “important questions are answered about each pattern” [BMMM98, p. 49]. This practice is in line with the presentation of design patterns used by Gamma et al. [GHJV95] and was also adopted by Demeyer et al. [DDN03] and Kerievsky [Ker04].

To describe deficiencies in this thesis, I use a template that strongly resembles the Full AntiPattern Template by Brown et al [BMMM98, p. 57ff]. The template consists of a number of sections with specific purposes which are explained in the following. Most of the sections are from Brown's AntiPattern template, but some are also slightly modified, renamed or omitted to fit into the context of this thesis.

Design Deficiency Name The concise and evocative name of the deficiency.

Removal Strategy Names The similarly evocative names of the strategies that can be applied to remove the deficiency.

Root Causes Brown et al. identify several root causes that can lead to the introduction of deficiencies into a system. These are: haste, apathy, narrow-mindedness, sloth, avarice, ignorance, pride, and responsibility.

Unbalanced Forces Brown et al. name six “primal forces” that have to be considered in the development of software systems and which can be unbalanced by the introduction of deficiencies. They call them: management of functionality, management of performance, management of complexity, management of

¹Patterns for programming languages can be seen as a special case of technology-specific patterns.

change, management of IT resources, and management of technology transfer. Another force that is not mentioned by Brown et al. but that could be added here is management of security.

Background The background section is meant to contain useful and interesting information that helps in understanding the deficiency and that motivates the need to resolve it.

General Form of this Design Deficiency This section describes characteristics of the deficiency in a generic form (i.e. not by means of an example). Often diagrams (e.g. class diagrams) and prose text are used for this.

Symptoms and Consequences This section should contain a list of consequences that arise from the introduction of the deficiency into a system. In this thesis, I use this section to point out why the given deficiency has an influence on the reconstruction of the system's software architecture.

Typical Causes Here, a bulleted list is used to enumerate the typical causes for the introduction of this deficiency. They should be more specific than the root causes mentioned above.

Known Exceptions Sometimes, there are exceptions to a rule. For example, in a certain context, the problem that are normally created by a deficiency may be negligible. Under such circumstances which are listed in this section, a deficiency may be tolerable in a system. This has to be decided on a case-by-case basis by the software architect. The deficiency ranking presented in Chapter 8 aims at supporting the architect in this decision.

Removal Strategies This section describes one or more solutions to the problems created by the deficiency. Brown et al. call this section "Refactored Solution" [BMMM98]. In this thesis, I avoid the term "refactoring" as it is commonly understood as a behaviour-preserving change which often is carried out at the source code level [Fow99]. Instead, I call this information *removal strategy* as this, to my mind, better addresses the higher abstraction level and the broader approach that is taken to the removal of design deficiencies. It is to be noted that different removal strategies for a deficiency may have different goals and are applicable only in certain situations. This is reflected in the strategies' descriptions. Removal strategies may be automatable. In these cases, the effect of their application on the recovered architecture can be calculated and presented to the architect (see Chapter 9). However, sometimes the considerations or necessary changes for a removal strategy are too complex to be automated. Still, the prose description of the removal strategy can be helpful to the architect.

Variations Similar to design patterns [GHJV95], a deficiency does not have *one* unique and immutable form. There may be variations which, in turn, may necessitate different removal strategies. In order to keep the section on the general form and the removal strategies focused and avoid clutter, variations of the deficiencies and their removal strategies are discussed in this section.

Example In this section, an example of a concrete deficiency occurrence and the possible removal strategies to remove it are given.

Related Solutions The last section lists related deficiencies and patterns, either from this thesis or from different pattern catalogues. This allows for an easy comparison and can also serve as a place to point out differences in terminology between different related patterns and deficiencies.

3.3. Running Example

In this section, I introduce the *Transfer Object Ignorance* design deficiency. I chose it as a running example because it is a non-trivial example of a common guideline for the design of good component-oriented architectures, i.e. the guideline that components should exchange data via transfer objects. This deficiency is used throughout this thesis to illustrate the Archimetric process. It is also one of the deficiencies that I used in the validation presented in Chapter 10. The deficiency is described with the AntiPattern template presented in Section 3.2.

Design Deficiency Name

Transfer Object Ignorance

Removal Strategy Names

Mark exposed class as transfer object, Move called method, Introduce transfer object

Root Causes

Ignorance This deficiency is easily introduced by developers that are unaware of communication design patterns for component-based systems such as data transfer objects.

Background

In component-based or service-oriented architectures, two components or services should exchange data only via data transfer objects [ACM01, Fow02]. A data transfer object is a data class² that contains only the data that is needed for

²Although the name of the pattern is “data transfer *object*”, it deals with the creation of specialised data *classes*. These classes are obviously created at design time. In a strict

a specific task and has no additional behaviour. The only methods of a transfer object are getters and setters for the contained data. In object-oriented programming languages, classes that are meant to be used as data transfer objects are usually designated by a special name prefix or suffix. This way, they can be clearly distinguished from unwanted data classes which are normally a bad smell in object-oriented programs [Fow99, p. 86]. For example, in the Java reference implementation of the Common Component Modeling Example (CoCoME, [RRMP08]), classes representing data transfer objects are marked with the suffix *TO*.

In contrast to common practice in object-oriented programming, communication by exchanging objects which are not transfer objects should be avoided in component-based systems. This has two main reasons: First, passing an object reference to another component in order to allow access to that object's data is a security risk and breaks the sending component's encapsulation. It inadvertently offers the receiving component the opportunity to invoke arbitrary methods of the exposed object. Second, as each component can be deployed independently [SGM02], each method call may possibly be a remote call that incurs a significant communication overhead. Polling necessary data by invoking a number of getters on (possibly different objects of) a remote component can therefore be very inefficient [ACM01, Fow02]. Thus, transfer objects should be used. They can be specifically constructed to contain all the relevant data for a certain activity and can then be sent to the receiving component as a whole.

Unbalanced Forces

Management of performance Ignoring transfer objects can decrease the performance of the system due to communication overhead.

Management of security Passing object references to other components can give them access to functionality that is not normally provided to them.

General Form of this Design Deficiency

Figure 3.1 shows the general form of the *Transfer Object Ignorance* deficiency in a component diagram. It contains two components, Component A and Component B. Following the notions of Szyperski ([SGM02], see Section 2.1.1), they are supposed to communicate exclusively via the interface *ICalledClass* which is provided by Component B and required by Component A. *ICalledClass* offers access to the method *calledMethod* of the *CalledClass* which expects a parameter *param* of the type *ExposedClass*. The *ExposedClass*, however, together with the *CallingClass* belongs to Component A.

The problem becomes obvious in the implementation of the *callingMethod* in the *callingClass*. The *callingMethod* has a reference *e* to an instance of the

sense, their instances which are used at run-time are the “real” data transfer objects. In order to be consistent with the terminology from the pattern description, I use the name “data transfer object” in this thesis, even when referring to the data class.

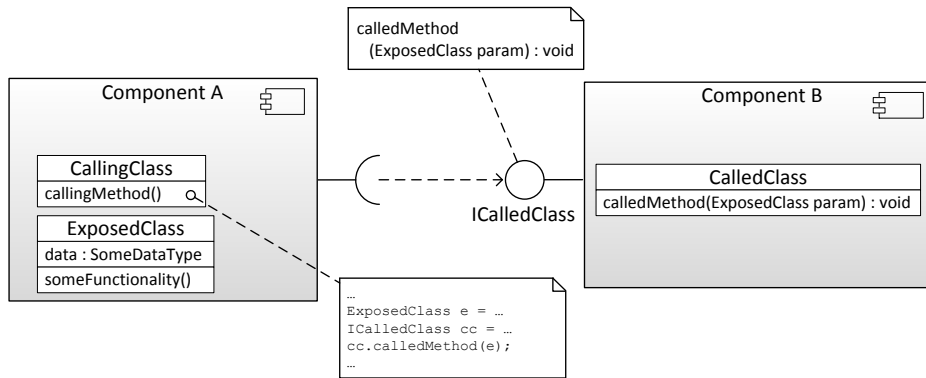


Figure 3.1.: General form of the *Transfer Object Ignorance* deficiency

ExposedClass and another reference `cc` of the type `ICalledClass`. The `callingMethod` calls the `calledMethod` of `cc` and passes `e` as an argument.

As explained in the background section, this may lead to security and performance problems. On the one hand, by passing the reference to `e` to the `CalledClass` in Component B, Component A loses control over which data of `e` is read by `cc`. `cc` may even change the data or call public methods of the `ExposedClass`, e.g. `someFunctionality()`. On the other hand, if `cc` was to interact a lot with `e`, each of these interactions may possibly be a remote call causing significant overhead as stated in the background section.

Symptoms and Consequences

If the example system was subject to a clustering-based architecture reconstruction algorithm, *Transfer Object Ignorance* deficiencies could gravely impact the reconstructed architecture. By directly passing the reference to an instance of the `ExposedClass` to the `CalledClass`, a strong coupling between the classes `ExposedClass`, `CalledClass`, and `CallingClass` is created. Coupling between classes is a metric that is used by many architecture reconstruction algorithms to group classes into components. Therefore, it stands to reason that all the classes involved in a *Transfer Object Ignorance* occurrence might be assigned to one component by the architecture reconstruction algorithm instead of grouping them into separate components. The impact of deficiencies on the metrics used during architecture reconstruction is examined in detail in Chapter 5.

Typical Causes

- Passing object references as arguments of method calls is a common practice in object-oriented programming. Programmers are used to this kind of communication between classes and may be ignorant to the fact that it is a deficiency for the communication between classes in different components.

- Programming languages that do not support components as a first-level concept (e.g. Java) have no means to detect this deficiency. Thus, programmers that introduce this deficiency in a system may simply be unaware that they should have used a transfer object.

Known Exceptions

Sometimes it may not be a problem to pass an object reference to another class even if a transfer object could be used in theory. For example, if the *CallingClass* and the *CalledClass* are part of the same conceptual component, they can communicate directly without violating component-based development principles. Also, in cases where *Component A* and *Component B* are parts of a common composite component, it may be tolerable if they do not use transfer objects for their communication.

Removal Strategies

There are several removal strategies that can be applied to remove the *Transfer Object Ignorance* deficiency. There is no clear answer as to which of them is suitable for the removal of a given deficiency occurrence. There may also be more strategies than the ones pointed out in this section.

Mark exposed class as transfer object The *Transfer Object Ignorance* deficiency may occur when a class that is intended to be used as a data transfer object is not correctly designated as such. For example, if the exposed class only contains fields of primitive data types and according access methods, it may be intended to be a transfer object. In order to remove a deficiency, the *ExposedClass* could be adapted to be an actual data transfer object. For example, in case of the CoCoME system, the suffix “TO” could simply be appended to the class name. Of course, this may necessitate an adaptation of the involved interfaces. In the deficiency’s general form depicted in Figure 3.1, changing the name of the *ExposedClass* will entail a change of the interface *ICalledClass*.

Move called method It may be the case, that the *calledMethod* may not be placed ideally in the *CalledClass* at all. Fowler describes this situation in his bad smells *Feature Envy* and *Inappropriate Intimacy* [Fow99]. If the method were moved, e.g. to the *CallingClass*, the components would no longer need to communicate with each other (at least as far as this particular interaction goes). This would also prevent the *Transfer Object Ignorance* deficiency. Of course, such a reengineering would also necessitate a change in the component’s interfaces and could have severe repercussions on other components and classes in the system.

Introduce transfer object Maybe the most obvious way to remove a *Transfer Object Ignorance* deficiency is to introduce a new data transfer object and use it for the passing of data instead of the object reference. This, however, requires an in-depth analysis of the deficiency occurrence to

determine which data or functionality of the `ExposedClass` is actually used in the `calledMethod`. If only data is used, a new class can be created that is correctly marked as a data transfer object. In the `calledMethod`, an instance of this class has to be created and filled with the required data. In order to accept this new transfer object, the `CalledClass` and the appropriate interface have to be adapted accordingly. (The changing of the interface may, of course, break other classes that implement it.). In addition, the behaviour of the `calledMethod` has to be changed to correctly process the transfer object. If functionality of the `ExposedClass` is used, the reengineering becomes even more complicated as this functionality may need to be moved to Component B in order to be available to the `calledMethod`. So in spite of being an obvious removal strategy, introducing a new transfer object consists of so many steps and considerations that it is hardly automatable.

Variations

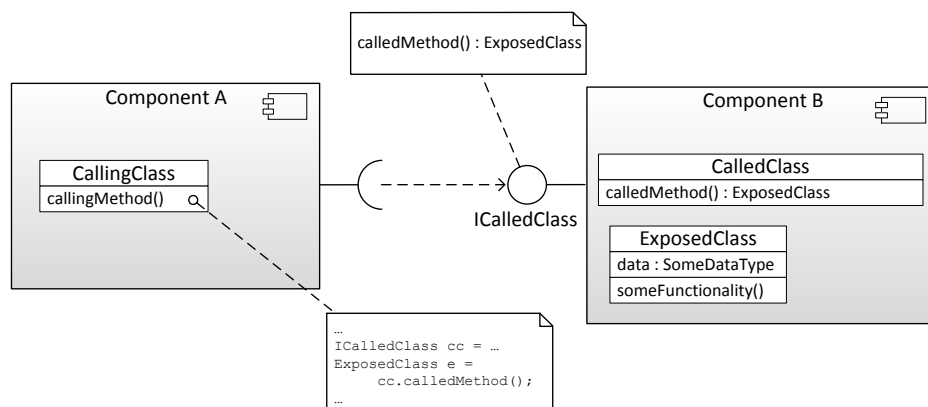


Figure 3.2.: Variation of the *Transfer Object Ignorance* deficiency: A class is exposed by returning it on a call.

Figure 3.2 shows a variation of the *Transfer Object Ignorance* deficiency's general form. In contrast to the general form where a parameter is used to pass an object reference across component boundaries, the same is accomplished here by returning an object reference in response to a call. In this case, the `ExposedClass` belongs to Component B. The `calledMethod` returns an object of the type `ExposedClass`. Because the `calledMethod` is available through the interface `ICalledClass`, it is possible for the `callingMethod` to call the `calledMethod` and thereby obtain an object reference to an instance of the `ExposedClass`. As shown in the code snippet for the `callingMethod`, `someFunctionality` which is not provided by the interface can then be called from Component A although this should not be allowed. Similar to the general form of the deficiency, if the `CalledClass` intended to return some data in response to the call of the `calledMethod`, it should do so

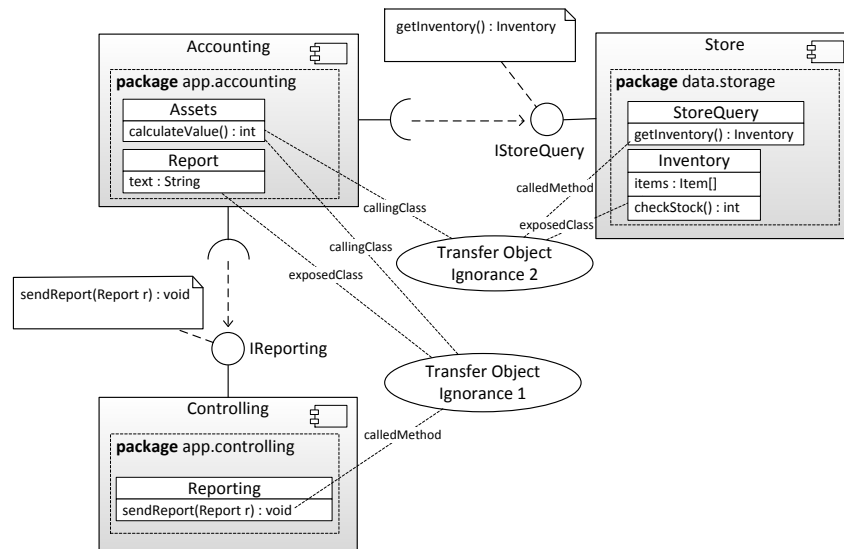


Figure 3.3 shows an excerpt of the concrete architecture of the example system that is used as a running example throughout this thesis. The figure shows the three components **Accounting**, **Controlling**, and **Store**. The components **Store** and **Controlling** provide the interfaces **IStoreQuery** and **IReporting** which are both required by the **Accounting** component. The system contains two occurrences of the *Transfer Object Ignorance* deficiency which are marked by labelled ellipses. They represent the two variants of the *Transfer Object Ignorance* deficiency. Variant no. 1 exists between the components **Accounting** and **Controlling** while variant no. 2 is located between **Accounting** and **Store**.

The Accounting component can send a Report to the Controlling component by calling the method `sendReport` of the class `Reporting`. In doing so, a `Report` object is passed to the Controlling component. This is an occurrence of the general form of the *Transfer Object Ignorance* deficiency as described above. The calling class of this occurrence is `Assets`, the called method is `sendReport` of the class `Reporting`, and the exposed class is `Report`. While `Report` is marked as the exposed class, it has the characteristic appearance of a data transfer object, i.e. it contains an attribute (whose access methods are omitted in Figure 3.3) but no methods that implement application logic. Although `Report` looks very much like a transfer object, it lacks the appropriate suffix “TO”. Thus, an appropriate removal strategy to remove this deficiency occurrence would be the strategy *Mark exposed class as transfer object* as described above. Since this strategy is rather simple, the removal could be performed by applying an automated removal strategy.

In the second occurrence, again, `Assets` is the calling class, the called method is `getInventory` of the class `StoreQuery`, and the exposed class is `Inventory`. We assume that the method `calculateValue` calls the method `getInventory` of the interface `IStoreQuery`. An instance of `Inventory` is returned to the calling class `Assets` on calling `getInventory`. This way, `Assets` gets access to `Inventory`'s attribute `items` as well as to its method `checkStock`. These members should not be available to classes from the `Accounting` component since they are not accessible via the interface `IStoreQuery`.

One possibility to remove this deficiency occurrence is to apply the *Introduce transfer object* removal strategy. By creating a new data class for the transfer object, in this case, e.g. `InventoryTO`, all relevant data that is required by the `Assets` class could be bundled and made available without exposing the `Inventory` class itself. To find out which data is required, the architect would have to analyse the `calculateValue` method. Afterwards, the method `getInventory` would have to be adapted, to create an instance of `InventoryTO`, populate it with the necessary data, and return it instead of the `Inventory` instance. As mentioned in the description of the removal strategy above, analysing which data is used by the `Accounting` component is very complex. Thus, it is very difficult to create an automatic transformation that performs this reengineering. The architect will probably have to carry out part of this reengineering manually.

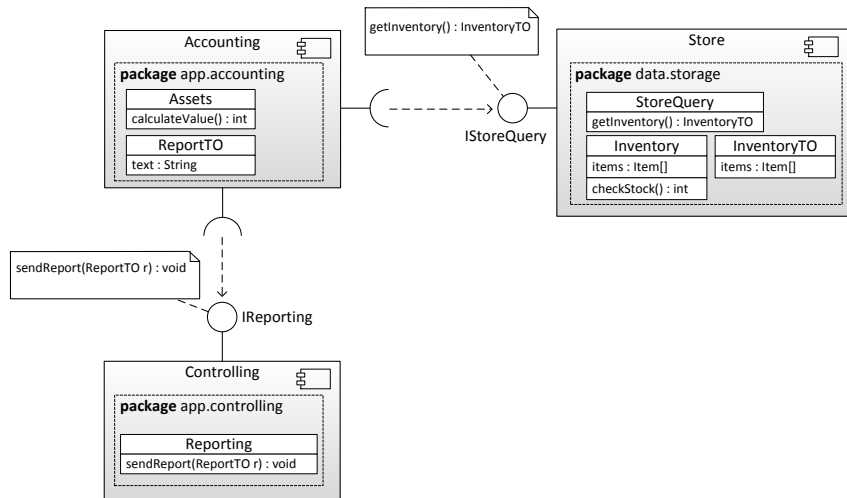


Figure 3.4.: Reengineered version of the running example from Figure 3.3

Figure 3.4 shows a reengineered version of the example system from Figure 3.3. As suggested above, the two removal strategies *Mark exposed class as transfer object* and *Introduce transfer object* have been applied to the deficiency occurrences no. 1 and 2, respectively. The class `Report` has been correctly marked as a transfer object by renaming it to `ReportTO` and adapting the interface `IReporting` accordingly. To remove deficiency occurrence no. 2, the new class `InventoryTO` has been created and the interface `IStoreQuery` has been changed to return an instance of this new class instead of an instance of `Inventory`. The

substantial changes that are required in the method `getInventory` which needs to create an instance of `InventoryTO` and populate it with the correct data from the `Inventory` are not visible in this figure.

Related Solutions

This paragraph provides short explanations of the patterns and deficiencies related to the *Transfer Object Ignorance* deficiency.

Data Class The data transfer objects used for the communication are *Data Classes* as described by Fowler [Fow99, p. 86]. Fowler characterises them as bad smells in an object-oriented design because they “are dumb data holders and are almost certainly being manipulated in far too much detail by other classes”. However, in the context of component-oriented design, these *Data Classes* are exactly what is needed for the data exchange between component to preserve the encapsulation of component behaviour.

Data Transfer Object The *Data Transfer Object* design pattern as described by Alur et al. and Fowler presents the rationale and implementation of component communication via transfer objects in detail [ACM01, Fow02].

Interface Violation When a class is exposed to other components by a *Transfer Object Ignorance* deficiency, an *Interface Violation* may occur when the components call methods of the exposed class that are not provided regularly via interfaces (See Appendix B.1).

Unauthorised Call Exposing a class and its methods through a *Transfer Object Ignorance* deficiency promotes *Unauthorised Calls* between components (See Appendix B.2).

3.4. Further Design Deficiencies

This section provides an overview of the further design deficiencies that have been examined in the course of this thesis. Detailed descriptions of these deficiencies can be found in Appendix B.

Interface Violation In component-based systems, the communication between components should be accomplished via the declared interfaces [SGM02]. However, this convention cannot always be enforced statically, for example in cases where the programming language does not support the concept of components directly. In these cases, an unwary developer may introduce a method call between classes that conceptually belong to different components even though the called method is not made available in an interface. This is called an *Interface Violation*.

Unauthorised Call Interfaces in component-based systems according to Szyper-ski are unidirectional: class A may call methods of class B as long as they are declared in the interface used by A. However, this does not allow B to call methods of A. Unfortunately, the connection of the classes in

one direction may convey the notion that these classes are “intended to work together” to inexperienced developers. Therefore, they may introduce calls from B to A without checking if this communication direction would be allowed by the architecture. As it also describes a method call in the absence of an appropriate interface, an *Unauthorised Call* is strongly related to the *Interface Violation* deficiency.

Inheritance between Components According to Szyperski, components are units of independent deployment [SGM02]. Therefore, there may not be an inheritance relationships between classes in different components. A clustering-based architecture reconstruction approach may however assign such classes to different components, for example because the classes are only loosely coupled otherwise. This deficiency informs the software architect that the reconstructed components are probably incorrect at this point.

4. The Archimetrix Process

This chapter presents the Archimetrix process, a reengineering process that aims at the recovery of a software’s architecture from its source code while taking design deficiencies into account. The process was first sketched in [TvDB11] and was refined in later publications [vDB11, PvDB12, vDPB13]. The chapter begins with a description of the scientific contributions of the process in Section 4.1. It is followed by an overview of the process in Section 4.2 which describes the two parts of the process and the involved roles. Section 4.3 provides details on the part of the process that is concerned with the architecture reconstruction. The discovery, documentation, and formalisation of design deficiencies is covered in Section 4.4. Finally, Section 4.5 discusses the limitations of the current process.

4.1. Contributions

The process presented in this chapter contributes to the area of component-based reengineering in the following ways:

- The process combines established techniques for the reverse engineering of component-based systems, namely clustering-based architecture reconstruction and pattern detection. This combination improves the reconstructed architecture by leveraging the individual strengths of both approaches and thereby mitigating their shortcomings: Architecture reconstruction is scalable and very useful to produce an overview of the system. However, its results can be adulterated by design deficiency occurrences. Pattern detection on the other hand allows for the precise identification of such deficiency occurrences. In isolation, however, pattern detection does not scale to large systems and may produce a large, confusing number of results.
- The Archimetrix process comprises a sub process for the structured discovery, documentation and formalisation of design deficiencies. In general, this is a creative process that is driven by domain knowledge, examination of existing systems, and the fusion of these elements into general patterns. However, I propose a guideline to structure these steps.
- Archimetrix is designed as an extensible approach. The different process steps can easily be extended, e.g. by adding new design deficiency formalisations and removal strategy formalisations (see Section 4.4), new component relevance metrics (see Section 6.5), and new metrics for the ranking of design deficiencies (see Section 8.3). In this thesis, I also suggest several future extensions of the process itself.

4.2. Process Overview

Figure 4.1 shows the Archimetric process. It consists of two parts: the iterative architecture reconstruction process on the left (shaded in a light grey), and a process for the formalisation of design deficiencies (shaded in a darker grey). The former part combines clustering-based architecture reconstruction with design deficiency detection. It is an iterative process with one iteration consisting of five steps. In addition, the deficiencies and removal strategies used in the iterative process have to be discovered, documented and formalised which is in the focus of the second part. The two parts are executed by different roles: the *software architect* carries out the iterative architecture reconstruction while the *deficiency expert* formalises the design deficiencies.

The steps in which pre-existing approaches are reused are marked with dashed lines in Figure 4.1. The iterative architecture reconstruction process is explained in Section 4.3. The formalisation of design deficiencies is described in Section 4.4.

4.3. Iterative Architecture Reconstruction

At the beginning of the process, I assume that only the source code of the system is available to the software architect. The conceptual architecture is unknown. Also, the architect does not know if and where design deficiencies exist in the source code.

Step 1: Architecture Reconstruction The Archimetric process starts with the architecture reconstruction of the system. This step takes the system’s source code as an input. The code is parsed and transformed into an abstract syntax tree representation. From this abstract syntax tree, an initial software architecture for the system is reconstructed. This initial architecture can provide a first rough overview of the system which is in line with Kazman’s advice to initially “obtain a high-level architecture view of the system before beginning the detailed reconstruction process” [KOV03]. However, it may be adulterated by design deficiencies if they are present in the system. Details on the architecture reconstruction step and on the impact of deficiencies thereon are given in Chapter 5.

Archimetric uses the Software Model Extractor (SoMoX) [CKK08, Kro10] for the architecture reconstruction (see Section 2.1.3). However, it would also be possible to integrate other clustering-based architecture reconstruction approaches into the Archimetric process.

Step 2: Component Relevance Analysis In order to prevent the adulteration of the reconstructed architecture, I propose to detect the design deficiencies and remove them from the system. As an additional benefit, this will improve the code quality. However, executing a design deficiency detection on the complete system is very time-consuming in the general case and does not scale well for large systems [SSL01, BT04a]. As a consequence, I suggest that the software

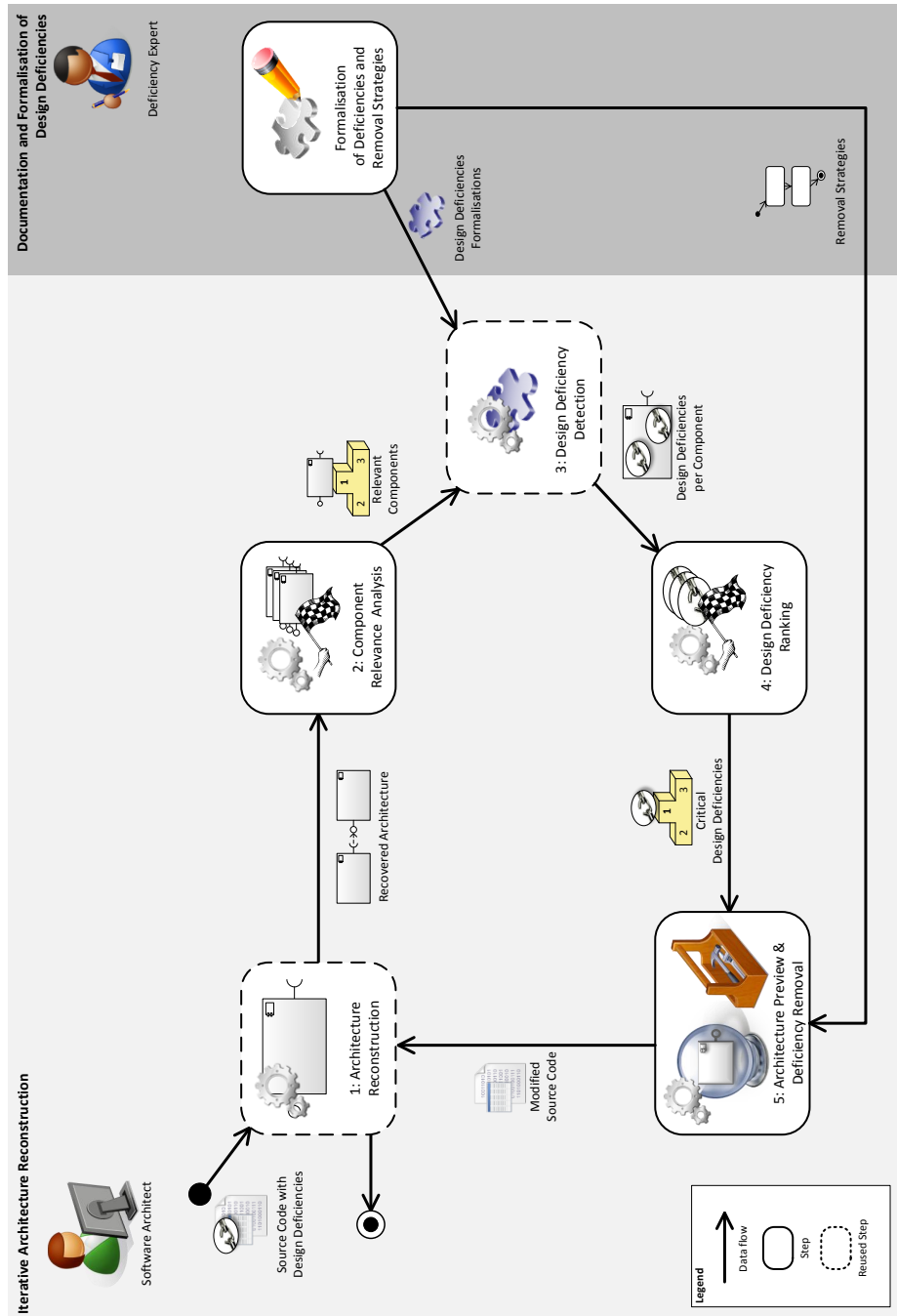


Figure 4.1.: The reengineering process with Archimetric

architect should select components from the initially recovered architecture to limit the search scope for the design deficiency detection. Ideally, the detection should be focused on components in which such a detection is worthwhile. These can be components that are very complex and therefore have a high probability of containing design deficiencies. On the other hand, components which are prone to change when the deficiencies are removed are a worthwhile detection target. To support the architect in the decision which components are a worthwhile input for the design deficiency detection, I propose the *Component Relevance Analysis* step. It takes the components from the initial clustering, rates them and thereby suggests a sensible input for the design deficiency detection. The relevance analysis is explained in detail in Chapter 6.

Step 3: Design Deficiency Detection In the next step, the design deficiency detection can be executed on the selected (relevant) component(s). The deficiency detection uses pattern detection techniques to find occurrences of predefined deficiencies in the selected components. (The formalisation of these deficiencies is discussed in Section 4.4.) Archimetrix mainly focuses on the structural analysis of the system, i.e. on the detection of deficiencies in the result model of the architecture reconstruction based on their structural properties. The deficiency detection yields a set of design deficiency occurrences in the previously selected components. This is explained in detail in Chapter 7.

Archimetrix uses Reclipse for the design deficiency detection (see Section 2.2.3). Similar to the architecture reconstruction step, Archimetrix does not depend on Reclipse on a conceptual level. Therefore, it is also possible to integrate different pattern detection approaches into Archimetrix.

As Reclipse, in general, also has behavioural pattern detection capabilities, i.e. the detection of patterns based on their run-time behaviour, Chapter 7 also discusses the combination of structural and behavioural detection approaches. It also points out how the behavioural pattern detection techniques can be improved.

Step 4: Design Deficiency Ranking Depending on the context in which a design deficiency occurs, some occurrences may be more critical than others, i.e. they may have a stronger influence on the architecture reconstruction or their removal may be either easier or more pressing. As a consequence, the architect has to decide which design deficiency occurrences should be removed and in which order. In large systems, however, this may be difficult. On the one hand, the number of detected deficiency occurrences may be so high that the architect can not easily get an overview. On the other hand, he may not be familiar enough with the system to gauge the importance (or even the correctness) of the detected occurrences. In order to support this decision, Archimetrix performs a *Design Deficiency Ranking* step that judges the severity of the detected design deficiency occurrences. This ranking mechanism takes all detected design deficiency occurrences as input and assigns a value between 0 and 1 to them. This can serve as an indication for the architect which deficiencies may be the most interesting to look at. Chapter 8 provides details on the design deficiency

ranking.

Step 5: Architecture Preview & Deficiency Removal To accomplish the removal of a design deficiency, different removal strategies exist. Sometimes, pre-defined removal strategies can be applied automatically. But there are also situations in which the architect has to intervene and to remove the design deficiency partly or completely manually. In both cases, the removal of the deficiency will affect the metric values measured in the clustering and thus will influence the architecture reconstruction.

If a pre-defined removal strategy is applied, its architectural consequences can be visualised by an *Architecture Preview*. This step takes a selected design deficiency occurrence and a chosen removal strategy as input. It produces a comparison of the current architecture and the architecture that would result from the application of the removal strategy. The software architect can then preview the effects of different removal strategies and determine which of the resulting architectures fits his requirements best. If no pre-defined removal strategy can be applied, the deficiency can also be removed manually. The architecture preview and the deficiency removal are described in Section 9.

Further iterations After the architect has removed one or more deficiency occurrences, the architecture reconstruction can be repeated. The newly reconstructed architecture may be different from the initially recovered one because the removed deficiencies no longer influence the clustering. The software architect can compare the different reconstructed architectures to each other. This comparison has to be performed manually at the moment. Tool-support for this step is the currently being developed in a master's thesis [Str13].

If the architect is satisfied with the newly reconstructed architecture, the process ends at this point. Otherwise, the reengineered system can be the starting point for a new iteration of the reengineering process.

4.4. Discovery, Documentation, and Formalisation of Design Deficiencies

Before design deficiencies can be detected in a software system, it has to be determined which deficiencies shall be detected. Sometimes there are pre-defined catalogues of deficiencies (e.g. [All02, BMMM98]). At other times, it is unclear which deficiencies commonly appear in a system so they have to be discovered first. The discovered deficiencies and their removal strategies then have to be documented and formalised in order to allow for an automatic detection. Removal strategies can be discovered and formalised at the same time as the corresponding deficiencies. Therefore, I do not explicitly mention them every time in the following description of the process. Section 9.5 explains the formalisation language and the application of removal strategies.

This section describes a process for the systematic discovery, documentation, and formalisation of design deficiencies. The process was first presented in

[vDPB13]. For the formalisation of deficiencies, I reuse an existing, domain-specific, graphical language which was developed in the Software Engineering Group at the University of Paderborn [NSW⁺02, Nie04] [vDT10].

Design deficiencies often occur because design principles, guidelines, or conventions are not adhered to. The reasons for this are manifold. For example, high time pressure may force developers to quickly implement something without spending much time on devising careful designs. In other cases, inexperienced developers may simply be unaware of guidelines or the guidelines may not be sufficiently enforced in a company, e.g. through regular code reviews.

Figure 4.2 shows the process for the discovery, documentation, and formalisation of such design deficiencies as a UML Activity. This process is performed by a *deficiency expert*, i.e. someone who is familiar with the formalisation language used in Archimetric.

The process consists of four phases.

1. The discovery of design deficiencies (Steps 1.1 to 1.3).
2. The documentation of the discovered design deficiencies (Step 2).
3. The formalisation of the documented design deficiencies (Step 3).
4. The validation of the design deficiency formalisations (Steps 4 to 6).

The enumeration of the phases already suggests that the phases have to be carried out in this order. For example, deficiencies can only be documented when they have already been discovered. The validation phase happens in iterations. The deficiency formalisations are tested on a test system or a real system. If the detection results are not satisfying, the formalisations are refined and the validation is repeated. The process ends when the deficiency detection yields satisfying results for the deficiency formalisations.

The following paragraphs explain the process steps in more detail.

Steps 1.1 to 1.3: Deficiency Discovery The first step in discovering deficiencies is to identify principles, guidelines, and conventions that are used in the system under study. These may be, for example, common principles of good component-based design like architectural styles or design patterns [Fow02, SGM02] (Step 1.1). The use of data transfer objects described in Section 3.3 is one example of such a common design principle. Design principles can be identified by studying textbooks or by talking to experienced software architects. The corresponding design deficiencies can then easily be derived by imagining violations of design principles like the ignorance of data transfer object in the running example.

On the other hand, there may be company- or project-specific guidelines which are not universally applicable to arbitrary component-based systems (see Section 3.1). For example, there may be company-specific naming conventions or development guidelines. The naming convention for transfer objects presented in Section 3.3 is an example of a project-specific convention that stems from the CoCoME project [RRMP08]. Other conventions may arise from the

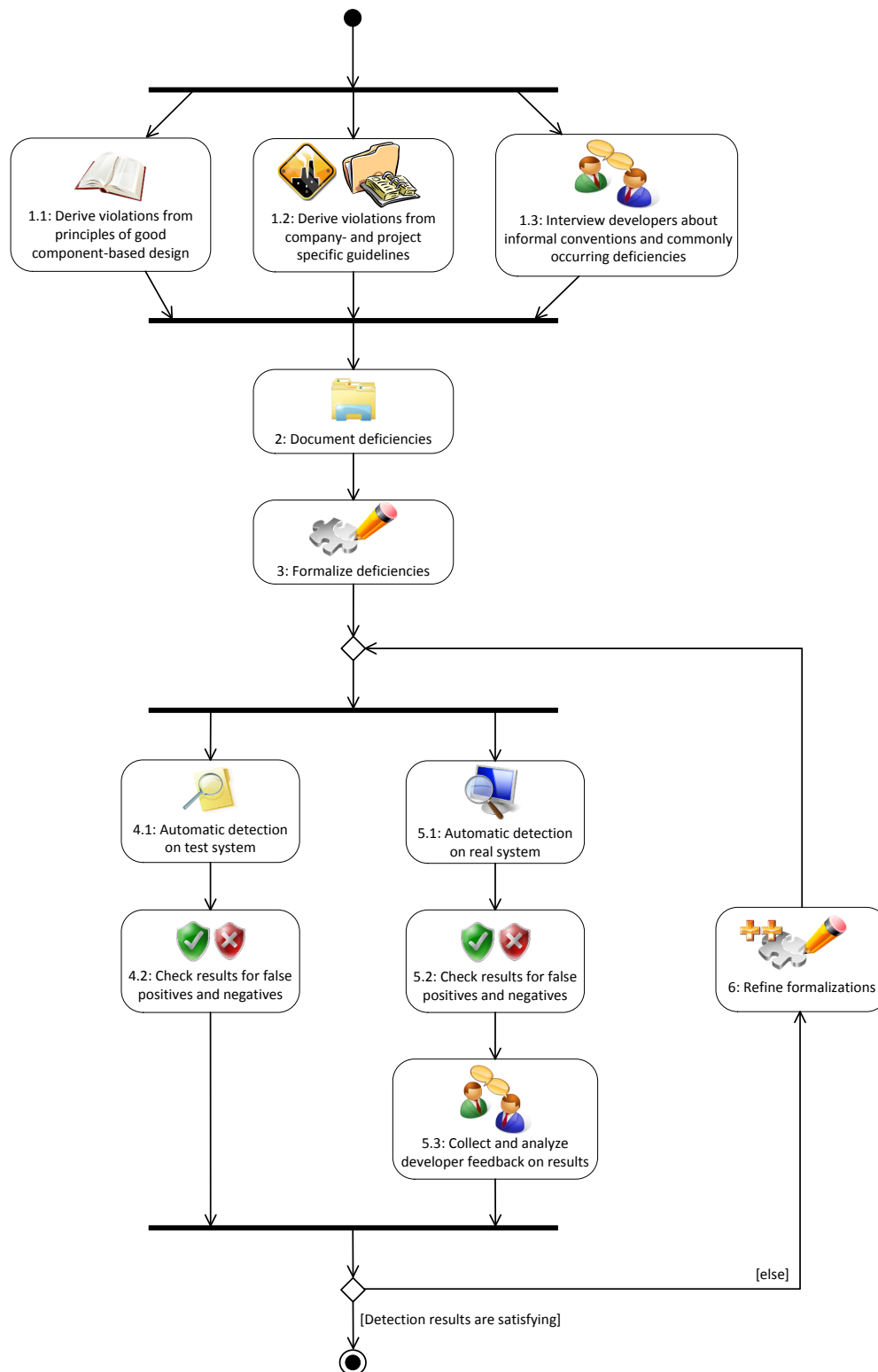


Figure 4.2.: Process for the documentation and formalisation of design deficiencies

use of a specific framework (for example JEE [Jav12]) in a project [ACM01]. Principles and guidelines like this can be discovered by studying documentation that is in use in the project, e.g. specific textbooks, guideline documents, or wikis that are used by the developers (Step 1.2). Similar to the more universal design deficiencies from Step 1.1, project-specific design deficiencies can be derived by thinking of ways how the discovered guidelines and conventions could be violated.

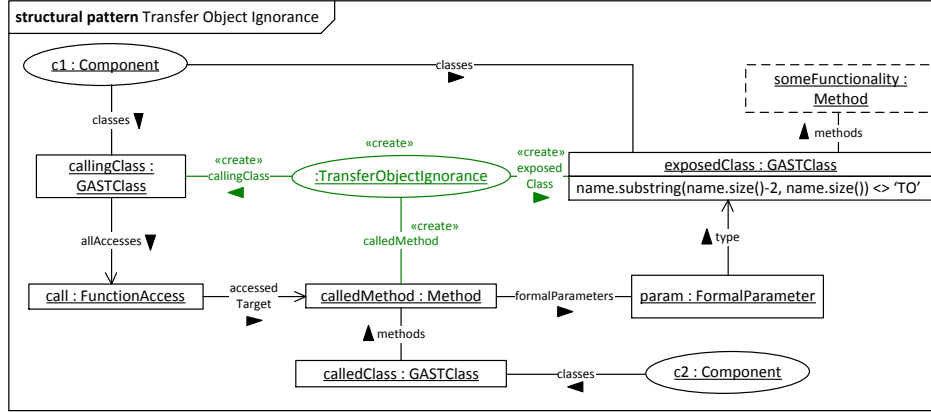
A third possible source of information about deficiencies are the system's developers. Developers may have conventions that are not formally documented and are disseminated by communication among the developers ("We always do it that way..."). Of course, the spread of this knowledge is highly non-transparent. Different developers may have different, possibly contradicting, knowledge of such conventions and it is hard to find out if everybody in the team has all relevant information. This can best be learned by interviewing the developers and documenting and consolidating their knowledge (Step 1.3). On the other hand, developers can be asked about commonly occurring deficiencies, especially deficiencies that may often be introduced by new and inexperienced developers. Detecting those deficiencies may not only be useful in the context of the Archimetric process. It can also increase the productivity of these developers by detecting their faults early and reliably.

Step 2: Documentation of Deficiencies Once the principles, guidelines, and conventions which should have been adhered to in the system in question are identified, the deficiencies that arise from their violation can be documented (Step 2). At first, this can be an informal, textual documentation. It is also possible to use pre-defined templates for the documentation of design deficiencies like the ones presented by Brown et al. [BMMM98]. The templates can be easily adapted to specific preferences or requirements in a company or project. Section 3.3 shows an exemplary documentation of the *Transfer Object Ignorance* deficiency that makes use of Brown's AntiPattern template. Independent of the formalisation and the later detection of deficiencies, this documentation can be very useful for the developers and their company. It can serve as a catalogue of worst practices that makes knowledge about possible deficiencies explicit and allows to avoid them in the future.

Step 3: Deficiency Formalisation As soon as the deficiencies are identified and documented, they can be formalised (Step 3). There is a large variety of domain-specific languages (DSL) for the formalisation of software patterns. Taibi presents a selection thereof [Tai07].

Archimetric uses Reclipse for the detection of design deficiencies [vDMT10a, vDMT10b, vDT10]. Therefore, in this thesis, the DSL provided by Reclipse is used to formalise the design deficiencies [NSW⁺02, Nie04]. In general, Reclipse can use static and dynamic (i.e. run-time) information for the detection of patterns.

In the scope of this thesis, the focus is on the use of static information. Hence, the deficiency formalisation in this section is accomplished with so-

Figure 4.3.: Structural formalisation of the *Transfer Object Ignorance* deficiency

called *structural patterns*. For every deficiency that is to be detected by Reclipse there has to be one structural pattern. Reclipse can then search for deficiency occurrences based on these formalisations.

The following section explains the formalisation of design deficiencies by means of the running example from Section 3.3. As the DSL for the formalisation of structural patterns is not a contribution of this thesis, it is only explained to the extent that is necessary for the understanding of the example formalisation.

Structural Formalisation The occurrence of a design deficiency is described by an object structure that constitutes such a deficiency. In the deficiency formalisations, this exemplary object structure is specified. In Reclipse, the objects are typed over an exchangeable meta model. In the case of Archimetric, I chose the source code decorator meta model which connects the reconstructed architecture model (the service architecture model or SAM) with the generalised abstract syntax tree (GAST) of the source code (see Section 5.4 and Appendix A). This way, the deficiency formalisations can reference information from both, the architecture and the source code. A more detailed explanation can be found in Chapter 7.

Figure 4.3 shows a formalisation of the *Transfer Object Ignorance* deficiency’s general form shown in Figure 3.1. In the upper left corner of Figure 4.3, an object labelled `callingClass` of type `GASTClass` is connected to an ellipse labelled `c1: Component`. The ellipse represents an annotation of a pattern occurrence that has been matched earlier in the detection process. In this case, it marks the component `c1` to which the `callingClass` belongs. (The use of the component patterns is explained in Section 7.4.)

The calling class contains a `FunctionAccess` call to a target `calledMethod`. The `calledMethod` belongs to a `calledClass` which in turn is part of a component `c2`. The `calledMethod` has a `FormalParameter` `param` of type `exposedClass`. The `exposedClass`

is a regular `GASTClass` which also belongs to component `c1`.

The `exposedClass` is not marked as a transfer object which is signified by the constraint that is imposed on its name. The constraint is expressed in the Object Constraint Language (OCL, [Obj12]). This expression states that the name may not end with the suffix “TO”. The `ExposedClass` contains a `Method someFunctionality`. This method is marked to be an additional object which is signified by the dashed border of the `someFunctionality` object. It means that the detection algorithm will try to find a method in the exposed class but that the deficiency occurrence can still be detected if such a method does not exist. These additional elements can be used to capture multiple structural variants of a deficiency with a single formalisation. In addition, the detection algorithm can attach a percentage to each deficiency occurrence that expresses which fragment of the formalisation could be detected [Tra07].

More deficiency formalisations are presented in Appendix B. A more detailed explanation of Reclipse’s structural pattern DSL and exemplary specifications of design patterns can be found in [vDT10].

Steps 4 to 6: Validation of the Formalisation The formalisation of deficiencies is a non-trivial task that has to be carried out manually. Therefore, a validation is necessary to determine if the formalisations are suited to detect the deficiencies they are supposed to detect. In the process presented in this section, this is accomplished in two different ways. On the one hand, a search for the formalised deficiencies can be executed on a test system (Step 4.1). The test system usually is a small, manually created system which deliberately contains occurrences of the deficiencies in question. The detection results are then inspected manually by the deficiency expert for false positives and false negatives (Step 4.2). If there are false positives, i.e. detected deficiency occurrences that match the formalisation but are not really deficiencies, the formalisation is not sufficiently exact. A false negative is a deficiency occurrence that was implemented in the test system but that is not detected. In this case, the formalisation may be too restrictive.

Another possibility to validate the deficiency formalisations is to execute a deficiency detection on a part of a real system (Step 5.1). Ideally, the formalised deficiencies are known to (or at least suspected to) occur in that part. After the detection, the detection results can be analysed for false positives and negatives (Step 5.2). Again, this is a manual task that has to be accomplished by the deficiency expert. Of course, the false negatives in this scenario can only be determined with respect to the known deficiency occurrences. Afterwards, the detection results should be discussed with the developers that are responsible for the analysed part of the system, provided they are still available (Step 5.3). They can give additional insight into the occurrence of the deficiencies and can also judge whether the detection results are of interest to them.

Finally, the formalisations can be adapted in accordance with the obtained insight (Step 6). Afterwards, the validation (Steps 4.1 to 5.3) and adaptation can be repeated until sufficiently good detection results is achieved. If the validation shows that the detection results are satisfying, the formalisation of

the deficiencies ends.

4.5. Limitations

The Archimetrix process leaves still room for extensions.

- At the moment, the software architect has to decide when he wants to terminate the process. Section 4.2 suggests that the process ends when the architect “is satisfied with the reconstructed architecture”. Kazman et al. point out that it is a general problem of architecture reconstruction processes that “there are no universal completion criteria” [KOV03]. This is also true for Archimetrix. For now this architect’s satisfaction with the architecture can only be based on a ‘gut feeling’ or simple metrics like ‘The system contains no more bad smells’. Thus, the architect obviously runs the “risk of stopping too soon” as Parnas puts it [Par11]. This situation could be improved by employing metrics to measure the modularisation quality of the system [SKR08]. In addition, design heuristics like the ones presented by Riel [Rie96] and by Cho et al. [CKK01] could be taken into account. For example, the architect could follow the process until a number of metrics reaches a certain threshold.
- The process presented in this chapter provides some guidance in the discovery of possible design deficiencies. However, it does not guarantee that all deficiencies that exist in the system under study are really discovered. Some deficiencies may be unknown to the developers or they may not be a direct violation of a general or specific design principle. These deficiencies may only be discovered if a developer or architect comes across them by chance.
- Design deficiencies can only be detected automatically when they are specified in a formal way. This formalisation is far from trivial and a predefined catalogue of common deficiencies could be helpful. Especially architects that are unfamiliar with the formalisation languages could benefit from such a catalogue. However, there is no predetermined, static set of deficiencies for component-based systems in general. While some deficiencies may occur in nearly all systems, others are highly dependent on technology choices or company-specific conventions (see Section 3.1).
- The formalisation of deficiencies as described in this chapter is sometimes complicated by the structure of the underlying type graph. It may be necessary to create several formalisations to cover all variants of a pattern. For instance, the type graphs used in this thesis (GAST, SAMM, and SCD; see Appendix A) use the same type to represent classes and interfaces (GASTClass). However, different elements are used to document the assignment of classes and interfaces to components (ComponentImplementingClassesLink and InterfaceSourceCodeLink, respectively). In some cases, this necessitates the creation of two deficiency formalisations which cover the same deficiency for classes and interfaces, respectively.

5. Influence of Design Deficiency Occurrences on the Architecture Reconstruction

This chapter deals with the reconstruction of component-based software architectures and how this is influenced by the presence of design deficiencies in the source code. It starts with the contributions of this chapter in Section 5.1. Archimetrix uses the tool SoMoX for the architecture reconstruction. Therefore, the reconstruction process of SoMoX is explained in detail in Section 5.2 and the dependencies between the metrics used in the process are analysed. Building on those observations, Section 5.3 shows how occurrences of design deficiencies can influence the metric values and thus adulterate the reconstructed architecture. The architecture model that is created by SoMoX is described in Section 5.4 because it is the input for the subsequent process steps of Archimetrix. Section 5.5 discusses limitations while Section 5.6 concludes the chapter.

5.1. Contributions

The contributions of this chapter are as follows.

- This chapter presents an explanation of the architecture reconstruction process reused in Archimetrix in terms of the running example. Thereby, it provides the foundation for understanding the influence of deficiency occurrences on the architecture reconstruction.
- The chapter also provides a detailed analysis of the influence of design deficiency occurrences on the architecture reconstruction. It clarifies the calculation of the metrics used in the architecture reconstruction. This allows for an illustration of the impact that simple details like added references between classes can have on the clustering metrics.

5.2. Reconstruction Process

This section explains the architecture reconstruction process that is realised in SoMoX. It is completely based on the work by Krogmann [Kro10] and the implementation of SoMoX.

Figure 5.1 depicts the architecture reconstruction process in SoMoX which consists of eight steps. First, the source code of the system is parsed by the parser SISSy [Sis11] and a Generalised Abstract Syntax Tree (GAST, see Appendix A.1) is created from it (1). SISSy already tries to detect classes that

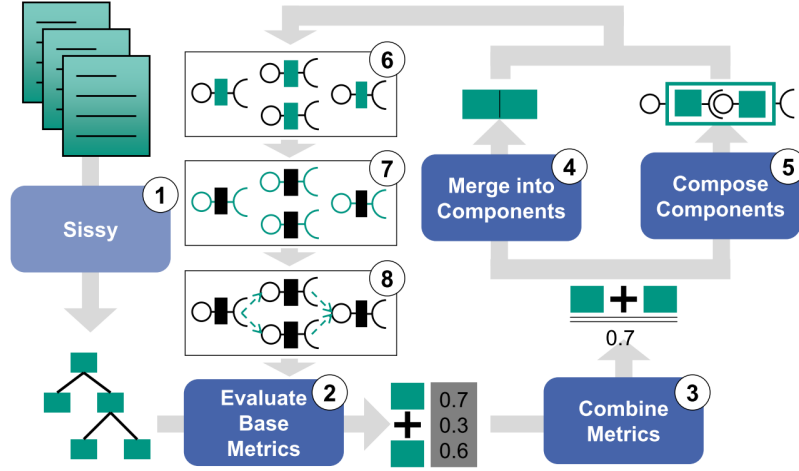


Figure 5.1.: Architecture reconstruction process in SoMoX (from [Kro10])

are meant to represent data transfer objects (see Section 3.3) and excludes them from the architecture reconstruction. This is done because on the code level, transfer objects are tightly coupled to their sending and receiving classes. This would cause SoMoX to cluster components together although the transfer objects are meant to separate them.

Steps (2) to (8) are carried out iteratively. Each iteration builds on the results of previous iterations and tries to create new components from the components reconstructed so far¹. The initial iteration produces one primitive component for each class. In the subsequent iterations, every combination of two components, either primitive or composite, constitutes a *component candidate*, i.e. a candidate for the creation of a new component by combining two smaller ones. There are three possibilities for combining a component candidate (see Figure 5.2):

- If the candidate consists of two primitive components, it can be merged, i.e. unified into one larger primitive component.
- A candidate can also be composed, i.e. encapsulated in a new composite component. All components of the candidate are retained as sub components of the newly created component. This can happen for both, primitive and composite components.
- Finally, a candidate can also be discarded. In this case, no new component is created.

To determine whether a component candidate is merged, composed, or discarded, new metric values are calculated in every iteration (Step 2 in Figure 5.1). Then the metric values are combined into a merging value v_{merge} and a composition value $v_{compose}$ for every potential candidate (3). The computation of

¹These clustering iterations are not to be confused with the iterations of the Archimatrix process.

v_{merge} and $v_{compose}$ is explained in detail in Section 5.2.2 below. If, for a candidate, v_{merge} exceeds a certain merging threshold t_{merge} , the candidate is merged (4). If $v_{compose}$ exceeds a certain composition threshold $t_{compose}$, the two components that constitute the candidate are composed (5). If both thresholds are exceeded, merging takes precedence over composition. If both aggregated values are below the respective thresholds, the candidate is discarded.

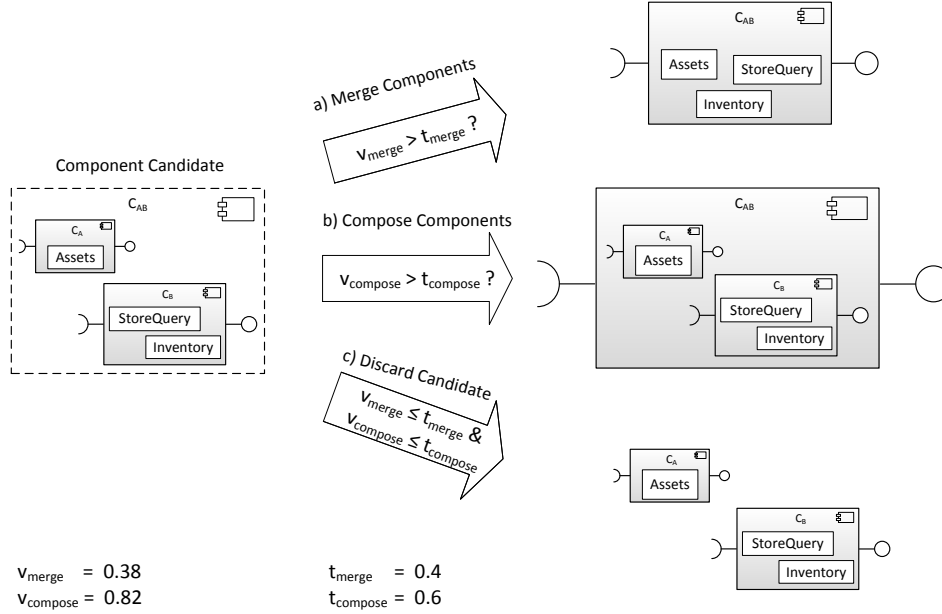


Figure 5.2.: Merge, composition, or discarding of a component candidate from the running example

Figure 5.2 shows a situation that could occur during the architecture reconstruction of the running example. The component candidate C_{AB} consists of two basic components, C_A and C_B . C_A contains the class `Assets` while C_B contains the classes `StoreQuery` and `Inventory`. If this candidate was merged, a new basic component containing all three classes would be created. If the candidate was composed, the two basic components would be retained but they would be integrated into a new composite component. If the aggregated metric values indicated neither a merge nor a composition, the candidate would be discarded.

In the example, v_{merge} is 0.38 and therefore below the threshold t_{merge} of 0.4. In contrast, $v_{compose}$ is 0.82 and therefore exceeds the threshold of 0.6. Hence, the candidate would be converted into a new composite component.

t_{merge} and $t_{compose}$ are configured by the user before the clustering starts. After each iteration, t_{merge} is increased while $t_{compose}$ is decreased. This way, component merges are more likely at the beginning of the clustering while component compositions become more likely towards the end. Then, new candidates are constructed from the recovered components and the next iteration commences. The clustering process ends when no new components can be produced by merging or composing component candidates during one iteration.

After the current component candidates have been merged, composed, or discarded, they are integrated into the current architecture model (Step 6 in Figure 5.1). Then, their interfaces are derived (7) and connectors between the interfaces are added (8). Then, the next iteration begins in which new component candidates are formed. The clustering process ends when no new components can be produced by merging or composing component candidates during one iteration.

The result of the clustering process is a mapping of the system elements to a number of reverse engineered components together with the connectors between those components.

The metrics used in step 2 of the reconstruction process are presented in Section 5.2.1. The strategies used in step 3 are explained in Section 5.2.2. For the purpose of explaining the metrics and strategies, I use the example component candidate shown in Figure 5.3.

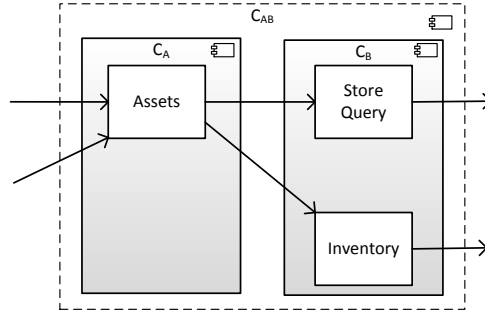


Figure 5.3.: Example of component candidates during the clustering

The component candidate is the same as in Figure 5.2. However, the interfaces have been omitted in Figure 5.3. Instead, the figure shows a number of references to and from the classes contained in the components. The references are relevant for the calculation of the metric values. The metrics and strategies in the next sections are explained in terms of the two components C_A and C_B and the component candidate C_{AB} from Figure 5.3.

The following description of the metrics and strategies used in SoMoX are based on the implementation of SoMoX. This way the following explanations and observations are in line with the validation in Chapter 10. In Krogmann's thesis, most of the metrics are explained from a conceptual point of view [Kro10]. There are however slight deviations between the description in his thesis and the implementation of SoMoX. Some deviations are marginal and can be addressed by translating metrics into each other ². However, the metrics described by Krogmann also contain errors ³. These inconsistencies are

²For example, Krogmann describes the Interface Violations metric which is calculated by subtracting the implemented Adherence to Interface Communication metric from 1 ($1 - Adh.To.InterfaceComm.$).

³The range of the Interface Violations metric is inverted, i.e. the metric is 1 if all commu-

circumvented by using SoMoX's implementation as a basis in this thesis. The following sections give an impression of the metrics and strategies used in SoMoX without explaining all the details of their calculation which can be found in [Kro10].

5.2.1. Metrics

This section briefly presents the metrics that are used in SoMoX and explains how they are combined in the architecture reconstruction. Other clustering-based architecture reconstruction approaches use identical or at least similar metrics. Especially the very basic metrics which measure the connection of elements, e.g. the different access counts and the different types of coupling, are used in all clustering-based architecture reconstruction techniques [Kos00, DP09].

External Accesses Count(C_A) This metric counts the number of accesses from within a given set of types C_A to all types outside of C_A .

Internal Accesses Count(C_A, C_B) This metric counts the number of accesses from within a given set of types C_A to all types in a second set C_B .

Interface Accesses Count(C_A, C_B) This metric counts the number of accesses from within a given set of types C_A to interfaces in a second set C_B .

Efferent Coupling(C_A) The efferent coupling metric counts the number of types within a set of types C_A which depends on types outside of C_A .

Afferent Coupling(C_A) The afferent coupling metric counts the number of types which are not with the set of types C_A but depend on types in C_A .

Abstract Types Count(C_A) This metric returns the number of abstract types within a set of types C_A .

Total Types Count(C_A) The total types count metric returns the number of types in a set of types C_A .

Name Resemblance(C_A, C_B) This metric calculates the resemblance of type names between types in C_A and C_B . The metric is commutative, i.e.

$$NameResemblance(C_A, C_B) = NameResemblance(C_B, C_A).$$

Coupling(C_A, C_B) The coupling metric relates the number of internal accesses between C_A and C_B to all external accesses of C_A as follows:

$$Coupling(C_A, C_B) = \frac{InternalAccessesCount(C_A, C_B)}{ExternalAccessesCount(C_A)}$$

Coupling is not commutative.

ication *bypasses* the interfaces instead of *using* them. For the Package Mapping metric, the commonRootHeight should not appear in the denominator but only in the numerator of the formula.

Adherence To Interface Communication(C_A, C_B) This metric relates the number of accesses through interfaces between two sets of types C_A and C_B to the number of all accesses between C_A and C_B .

$$Adh.ToInterfaceComm.(C_A, C_B) = \frac{InterfaceAccessesCount(C_A, C_B)}{InternalAccessesCount(C_A, C_B)}$$

This metric is not commutative.

Instability(C_{AB}) The Instability of a component candidate C_{AB} indicates if the candidate has many external dependencies that make its implementation likely to change if external classes of interfaces change. It makes use of the efferent and afferent coupling metrics:

$$Instability(C_{AB}) = \frac{EfferentCoupling(C_{AB})}{EfferentCoupling(C_{AB}) + AfferentCoupling(C_{AB})}$$

Abstractness(C_{AB}) The Abstractness metric indicates the portion of abstract types in relation to all types of a component C_{AB} .

$$Abstractness(C_{AB}) = \frac{AbstractTypesCount(C_{AB})}{TotalTypesCount(C_{AB})}$$

Distance from the Main Sequence(C_{AB}) Components should neither be too unstable nor too abstract. This is captured by the Distance from the Main Sequence metric which was first introduced by Martin [Mar94]. Components on the main sequence represent a good trade-off between instability and abstractness.

$$DMS(C_{AB}) = |Abstractness(C_{AB}) + Instability(C_{AB}) - 1|$$

Package Mapping(C_A, C_B) The Package Mapping metric expresses the “distance” of two types or sets of types C_A and C_B in terms of the system’s package structure. Types that reside in the same part of the package structure receive a high Package Mapping value while types in completely different branches of the package tree receive a low value. The metric is commutative.

$$PackageMapping(C_A, C_B) = \frac{commonRootHeight(C_A, C_B)}{maxHeight(C_A, C_B)}$$

Directory Mapping(C_A, C_B) The Directory Mapping metric is comparable to the package mapping metric and mainly targets programming languages like C which do not support packages but in which directories are used to organize the types. For languages like C++ that use both packages and directories, this metric can be combined with Package Mapping. The metric is commutative.

Slice Layer Architecture Quality A common architectural style of organizing

business information systems is the separation into slices and layers. According to Krogmann, “slices are service oriented cuts of a software system, like for example, contracting, billing, and customer data management. Layers are cross-cutting technology-induced cuts of software system, like for example, a view layer, a middle-tier, and a database access layer.” [Kro10]. This metric tries to capture the quality of this slice layer architecture (SLAQ). The necessary information is derived by a heuristic which analyses the system’s package structure. In contrast to the other metrics, this analysis is realised by an algorithm that cannot be captured in a single, simple formula.

Natural Subsystem(C_{AB}) A Natural Subsystem is set of types a system that realises exactly one slice in one layer of the complete system. This metric indicates the likelihood of a component candidate representing such a natural subsystem. Like the SLAQ metric, this metric is also heuristically derived based on the system’s package structure and it cannot be expressed in a simple formula.

5.2.2. Strategies

Name Resemblance After Coupling(C_A, C_B) Sometimes, types may be named similarly although they have nothing to do with each other. The name resemblance metric does not account for this case. The Name Resemblance After Coupling strategy⁴ therefore is based on the Name Resemblance and the Coupling metrics. The Name Resemblance value of two sets of types C_A and C_B is only taken into account if they are coupled at the code level, i.e. if $\max(\text{Coupling}(C_A, C_B), \text{Coupling}(C_B, C_A))$ is greater than a user-definable threshold ϵ .

Subsystem Component(C_A, C_B) The Subsystem Component strategy is very similar to the Natural Subsystem metric on which it is based. It scales the Natural Subsystem value however in order to take small metric values into account.

Component Merge Together with Component Composition, Component Merge is one of the two strategies that decide if a component candidate is converted into a new component during the clustering. It is the weighted sum of the five strategies and metrics Adherence To Interface Communication, Name Resemblance After Coupling, Package Mapping, Directory Mapping, and Subsystem Component. The weights w_1 to w_5 can be set to a value between 0 and 1 in order to adapt the architecture recovery to the system under analysis. By setting a weight to 0, a given metric be completely left out of the calculation of the merge value. On the other hand, a greater emphasis can be placed on a given metric by increasing the corresponding weight. SoMoX provides a default configuration of the weights which can serve as a starting point for tuning the weights to the

⁴This strategy is called Consistent Naming by Krogmann [Kro10]. Here, I use the name from the implementation of SoMoX.

system under analysis. In the formula, w_1 is negative because an adherence to interface communication does not indicate that a component candidate should be merged. On the contrary, the bypassing of interfaces (and therefore a “negative adherence”) indicates a merge. The value of this strategy is denoted by v_{merge} above.

$$\begin{aligned}
ComponentMerge(C_A, C_B) = & \\
& (-w_1 \times Adh.ToInterfaceComm.(C_A, C_B) \\
& + w_2 \times NameResemblanceAfterCoupling(C_A, C_B) \\
& + w_3 \times PackageMapping(C_A, C_B) \\
& + w_4 \times DirectoryMapping(C_A, C_B) \\
& + w_5 \times SubsystemComponent(C_A, C_B)) \\
& / (w_1 + w_2 + w_3 + w_4 + w_5)
\end{aligned}$$

Component Composition Similarly to Component Merge, Component Composition calculates a weighted sum of other metrics and strategies. In addition to the metrics used in Component Merge, Component Composition also takes the DMS metric into account. The value of this strategy is denoted by $v_{compose}$ above. Similar, to the Component Merge strategy, the Component Composition strategy can also be adapted to the system under analysis by setting the weights w_1 to w_6 .

$$\begin{aligned}
ComponentComposition(C_A, C_B) = & \\
& (w_1 \times Adh.ToInterfaceComm.(C_A, C_B) \\
& + w_2 \times NameResemblanceAfterCoupling(C_A, C_B) \\
& - w_3 \times DistanceFromTheMainSequence(C_{AB}) \\
& + w_4 \times PackageMapping(C_A, C_B) \\
& + w_5 \times DirectoryMapping(C_A, C_B) \\
& + w_6 \times SubsystemComponent(C_A, C_B)) \\
& / (w_1 + w_2 + w_3 + w_4 + w_5 + w_6)
\end{aligned}$$

5.2.3. Dependencies Between Metrics and Strategies

As described in the previous sections, metrics and strategies can (partly) be composed of other metrics. This means that the metric values depend on each other: When the value of a basic metric changes, this also affects the values of all metrics and strategies that depend on it. Figure 5.4 visualises the interdependencies between the metrics and strategies used in SoMoX.

Figure 5.4 is organized in five horizontal layers. The upper three layers depict metrics while the lower two layers contain component detection strategies. In the top layer are seven basic metrics that count types (e.g. Total Types Count) or accesses (e.g. External Accesses Count). The second layer contains eight metrics that either calculate values based on the counting metrics (e.g. Coupling) or that use an algorithm to calculate their respective values (e.g. Name Resemblance). Layer 3 contains only the two metrics Distance from the Main

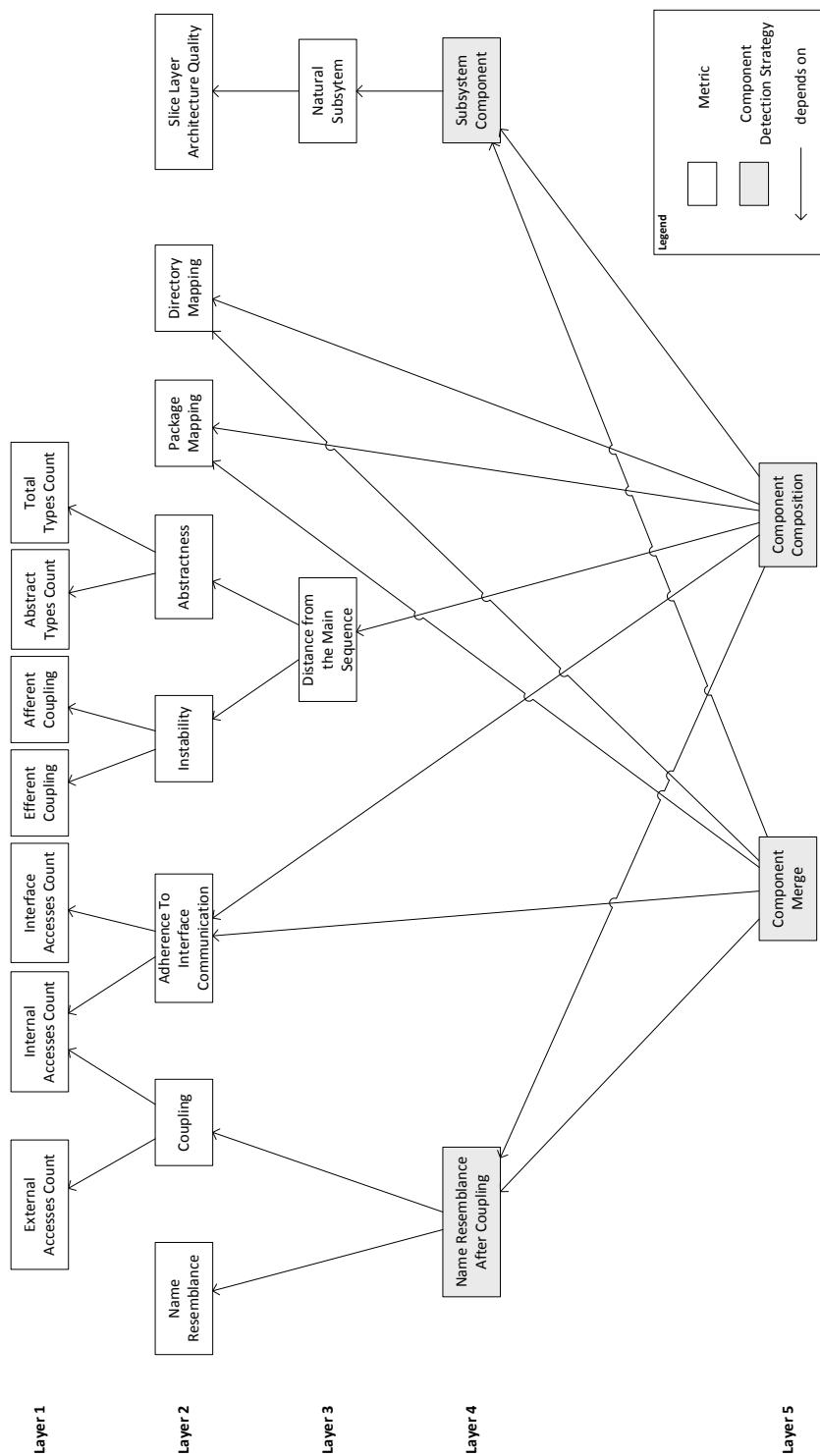


Figure 5.4.: Dependencies between the metrics and strategies

Sequence and Natural Subsystem which are based on metrics from the second layer. The fourth layer contains two strategies that base their calculations on metrics while the fifth layer contains the two most important component detection strategies, Component Merge and Component Composition. The figure shows clearly that both, Component Merge and Component Composition, indirectly depend on several of the counting metrics from the top layer. This means that any change in the accesses or types of a component candidate, for example by introducing a design deficiency, may influence its merge or composition value. The following section analyses exactly which metric values are influenced to which extent by the *Transfer Object Ignorance* deficiency from Chapter 3.3.

5.3. Influence of Design Deficiency Occurrences on the Metrics

In this section, the concrete influence of deficiency occurrences on the metric values is examined. First, I analyse the influence of a single occurrence of the *Transfer Object Ignorance* deficiency on the architecture reconstruction with SoMoX. Afterwards, the degree to which different metrics are susceptible to deficiency occurrences and the influence of other deficiencies are discussed.

5.3.1. Influence of the Transfer Object Ignorance Deficiency

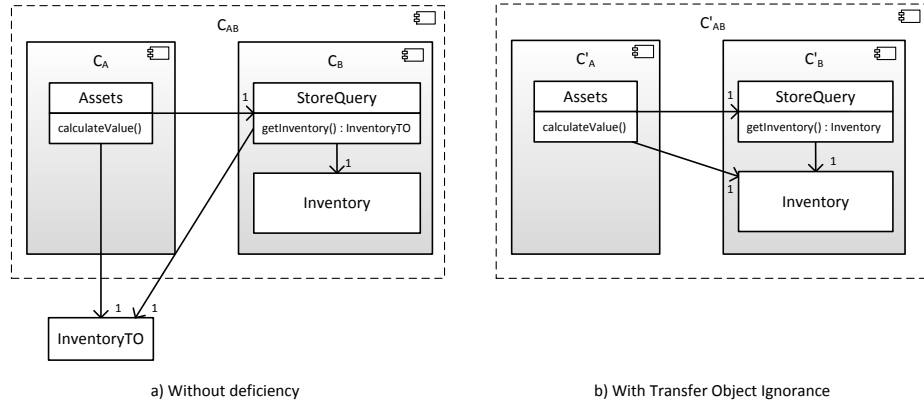


Figure 5.5.: Example of the influence of a *Transfer Object Ignorance* deficiency on the architecture recovery

In order to demonstrate the influence of design deficiency occurrences on the metrics used by SoMoX, I illustrate an example situation that could happen during the architecture reconstruction process. Figure 5.5 shows two component candidates, C_{AB} on the left and C'_{AB} on the right. The interfaces of the components have been omitted in the figure. Similar to Figure 5.3, the compo-

nent candidate C_{AB} comprises the two components C_A and C_B . In comparison, the same situation is visualised a second time, this time with a component candidate C'_{AB} which is comprised of the components C'_A and C'_B . C'_{AB} contains an occurrence of the *Transfer Object Ignorance* deficiency.

Component candidate C_{AB} does not contain a deficiency occurrence. The class *Assets* in component C_A has a reference to the class *StoreQuery* in component C_B and also to the transfer object *InventoryTO* which lies outside of the components because it is ignored by the clustering algorithm (cf. Section 5.2). *StoreQuery* contains a method *getInventory* which is called by *Assets* and which shall return data from the class *Inventory*. To pass data on the current state of the *Inventory* to *Assets*, *StoreQuery* creates an instance of the transfer object class *InventoryTO*, populates it with the necessary data and passes it as a return value to *Assets*. Note that *Inventory* has no reference to either *Assets* or *StoreQuery*, this way.

In contrast, the component candidate C'_{AB} on the right side of Figure 5.5 contains a *Transfer Object Ignorance* deficiency. Instead of using a transfer object, *StoreQuery* directly passes a reference to an instance of *Inventory* to *Assets*. Therefore, *Assets* has a direct reference to *Inventory*.

	Without Deficiency	With Transfer Object Ignorance
External Accesses Count(C_A)	eac_{C_A}	$eac_{C'_A} = eac_{C_A} + 1$
External Accesses Count(C_B)	eac_{C_B}	$eac_{C'_B} = eac_{C_B}$
Internal Accesses Count (C_A, C_B)	iac_{C_A, C_B}	$iac_{C'_A, C'_B} = iac_{C_A, C_B} + 1$
Internal Accesses Count (C_B, C_A)	iac_{C_B, C_A}	$iac_{C'_B, C'_A} = iac_{C_B, C_A}$
Interface Accesses Count (C_A, C_B)	$ifac_{C_A, C_B}$	$ifac_{C'_A, C'_B} = ifac_{C_A, C_B}$
Interface Accesses Count (C_B, C_A)	$ifac_{C_B, C_A}$	$ifac_{C'_B, C'_A} = ifac_{C_B, C_A}$
Efferent Coupling(C_A)	$efcp_{C_A}$	$efcp_{C'_A} = efcp_{C_A}$
Afferent Coupling(C_A)	$afcp_{C_A}$	$afcp_{C'_A} = afcp_{C_A} + 1$
Efferent Coupling(C_B)	$efcp_{C_B}$	$efcp_{C'_B} = efcp_{C_B} + 1$
Afferent Coupling(C_B)	$afcp_{C_B}$	$afcp_{C'_B} = afcp_{C_B}$

Efferent Coupling (C_{AB})	$efcp_{C_{AB}}$	$efcp_{C'_{AB}} = efcp_{C_{AB}}$
Afferent Coupling (C_{AB})	$afcp_{C_{AB}}$	$afcp_{C'_{AB}} = afcp_{C_{AB}}$
Abstract Count (C_A)	Types atc_{C_A}	$atc_{C'_A} = atc_{C_A}$
Abstract Count (C_B)	Types atc_{C_B}	$atc_{C'_B} = atc_{C_B}$
Total Count (C_A)	Types ttc_{C_A}	$ttc_{C'_A} = ttc_{C_A}$
Total Count (C_B)	Types ttc_{C_B}	$ttc_{C'_B} = ttc_{C_B}$
Name Resemblance (C_A, C_B)	nr_{C_A, C_B}	$nr'_{C_A, C_B} = nr_{C_A, C_B}$
Coupling (C_A, C_B)	$\frac{iacc_{C_A, C_B}}{eac_{C_A}}$	$\frac{iacc'_{C'_A, C'_B}}{eac_{C'_A}} = \frac{iacc_{C_A, C_B} + 1}{eac_{C_A} + 1}$
Coupling (C_B, C_A)	$\frac{iacc_{C_B, C_A}}{eac_{C_B}}$	$\frac{iacc'_{C'_B, C'_A}}{eac_{C'_B}} = \frac{iacc_{C_B, C_A}}{eac_{C_B}}$
Adh.To. Interface Comm. (C_A, C_B)	$\frac{ifacc_{C_A, C_B}}{iacc_{C_A, C_B}}$	$\frac{ifacc'_{C'_A, C'_B}}{iacc_{C'_A, C'_B}} = \frac{ifacc_{C_A, C_B}}{iacc_{C_A, C_B} + 1}$
Adh.To. Interface Comm. (C_B, C_A)	$\frac{ifacc_{C_B, C_A}}{iacc_{C_B, C_A}}$	$\frac{ifacc'_{C'_B, C'_A}}{iacc_{C'_B, C'_A}} = \frac{ifacc_{C_B, C_A}}{iacc_{C_B, C_A}}$
Instability (C_{AB})	$ins_{C_{AB}} = \frac{efcp_{C_{AB}}}{efcp_{C_{AB}} + afcp_{C_{AB}}}$	–
Abstractness (C_{AB})	$abs_{C_{AB}} = \frac{atc_{C_{AB}}}{ttc_{C_{AB}}}$	$abs_{C'_{AB}} = abs_{C_{AB}}$
DMS (C_{AB})	$dms_{C_{AB}} = abs_{C_{AB}} + ins_{C_{AB}} - 1 $	–
Package Mapping (C_A, C_B)	pm_{C_A, C_B}	$pm_{C'_A, C'_B} = pm_{C_A, C_B}$
Directory Mapping (C_A, C_B)	dm_{C_A, C_B}	$dm_{C'_A, C'_B} = dm_{C_A, C_B}$

 Table 5.1.: Influence of a *Transfer Object Ignorance* deficiency on the metric values

Table 5.1 shows the effect of this single occurrence of the *Transfer Object Ignorance* deficiency on the metrics that are used during the architecture reconstruction process. The left column contains the name of the respective metric. The parameter names reference the components C_A , C_B , and the component candidate C_{AB} from Figure 5.5. Commutative metrics are listed for both directions, non-commutative metrics only once. The second column contains the

metric values for the component candidate without a deficiency. Because the example only consists of an excerpt of a potentially larger system, no absolute metric values are used. I rather use variables that are then referenced in the third column. This third column contains the metric values for the component candidate with the transfer object deficiency. Here, absolute values are also avoided but the variables from the second column are referenced to show the value changes where necessary.

As Table 5.1 shows, the metric values for External Accesses Count(C_B), Internal Accesses Count (C_B, C_A), Afferent Coupling(C_A), Efferent Coupling(C_B), Coupling(C_B, C_A), and Adherence To Interface Communication(C_B, C_A) change as indicated in the third column⁵. The changes all stem from the reference between Assets and Inventory.

The instability measures the coupling of the whole component candidate to the rest of the system. As the rest of the system is not considered in this example, no statement about this metric can be made. The DMS metric is partially based on the instability. Therefore, no statement about the influence of the example deficiency occurrence on this metric can be made, either.

The metrics Package Mapping and Directory Mapping only change when new elements are created or when old elements are removed. Therefore, they do not change in this concrete example.

The metrics Slice Layer Architecture Quality and Natural Subsystem are omitted in Table 5.1. As they are rather complicated heuristics, it is not easily possible to predict their change in any given situation.

Since the component detection strategies depend directly on the metrics (cf. Section 5.2.3), their values are also changed. The strategy Name Resemblance After Coupling partly depends on the influenced Coupling metric. Component Merge depends, among others, on Name Resemblance After Coupling, Coupling, and Adherence To Interface Communication which are all changed. The Component Composition strategy is dependent on the changed strategy Name Resemblance After Coupling as well as the metric Adherence To Interface Communication.

In summary, the *Transfer Object Ignorance* deficiency leads to the addition of one reference between C_A and C_B compared to the case without the deficiency. This in turn affects six metric values and three strategy values, among them the crucial Component Merge and Component Composition strategies. It stands to reason that the occurrence of several design deficiencies in a system can significantly influence the architecture recovery process of clustering-based reverse engineering approaches like SoMoX.

5.3.2. Susceptibility of Different Metrics and Strategies

The different metrics and strategies are influenced in different ways by deficiency occurrences. Although Section 5.2.3 showed that even high-level component-detection strategies are influenced by the changes of low-level metrics, this

⁵Note that transfer objects like *InventoryTO* are exempt from the clustering (see Section 5.2). They are filtered by SoMoX before the clustering based on their names and neither themselves nor references to them do affect the metric values.

influence is not the same for all metrics.

Table 5.1 shows that the low-level metrics which count relations between elements in the component candidates are clearly influenced. Several values are increased by one. This is true for several access counts and also for some of the coupling metrics.

Which of these counting metrics are increased depends on the specific deficiency occurrence. If for example, the example occurrence would also introduce a reference from *Inventory* to *Assets*, the metrics External Accesses Count(C_B), Internal Accesses Count (C_B , C_A), Efferent Coupling(C_A), and Afferent Coupling (C_B) would increase in addition to the already affected metrics. Similarly, the addition or removal of types caused by a deficiency occurrence would influence the different type count metrics which are not influenced in the above example. This, in turn, might influence the currently unaffected abstractness metric which depends on two of the type count metrics.

Changes of the metrics Name Resemblance, Package Mapping, and Directory Mapping cannot be described for deficiencies in general. It is often the case that changes in a system also include the renaming of classes or packages. If a deficiency is introduced during such a change, the corresponding metrics would also be affected. This however can only be decided on a case-by-case-basis.

The component detection strategies are calculated by combining different metrics, including the very basic ones. Therefore, they are influenced by changes in the basic metric values but the extent of this influence cannot be easily predicted and depends on the specific situation, e.g. on the number of relations of the involved elements, and on the specific occurrence. It is however plausible that the strategies will be affected to a lesser degree than the basic metrics because they combine various different metrics. If only one or two of them change, the overall strategy value may stay more or less the same.

In summary, the more basic metrics are influenced more by deficiency occurrences. However, given the right circumstances, a single deficiency occurrence or the combination of several occurrences may influence the value of the component detection strategies substantially. In some cases, this influence may cause the architecture reconstruction algorithm to merge or compose component candidates which would not have been reconstructed without the deficiency occurrences.

5.3.3. Influence of Other Design Deficiencies Occurrences

The previous section illustrates the impact of a *Transfer Object Ignorance* occurrence on the architecture reconstruction with SoMoX. In part, other deficiency occurrences also influence the metrics and strategies in similar ways. The other deficiencies that have been examined in the course of this thesis are described in Appendix B.

Interface Violation and Unauthorised Call The deficiencies *Interface Violation* and *Unauthorised Call* influence the architecture reconstruction in a way similar to the *Transfer Object Ignorance* deficiency. Both, *Interface Violation*

and *Unauthorised Call*, are concerned with method accesses which are not allowed by the provided interfaces. Therefore, both deficiencies will influence the access count metrics and the coupling metrics similar to *Transfer Object Ignorance* occurrences.

Inheritance between Components The *Inheritance between Components* deficiency is different from the other three deficiencies. Its occurrence indicates that an inheritance relationship exists between two classes that were assigned to different components. Such an inheritance relationship affects a whole range of metrics. As the inheritance counts as an access between the involved classes, the access count metrics are influenced which in turn can alter the coupling (see 5.4). The efferent and afferent coupling of components will lead to a changed instability value. In addition, assigning a class to another component will affect the total types count of the involved components and, in case of super class, maybe also the abstract types count. This then leads to a different abstractness value which, together with instability, influences the distance from the main sequence.

However, a relationship between two classes normally leads to the assignment of these classes to a common component as illustrated in the previous section. Hence, in case of the *Inheritance between Components* deficiency, these classes were assigned to the different components *in spite of the inheritance relationship*. As discussed in Appendix B.3, this deficiency cannot be easily removed by changing the source code as removing the inheritance relationship would have a severe impact on the semantics of the program. Even if the inheritance relationship was removed by a semantics-preserving refactoring, this would rather reinforce the distribution of the classes to different components due to the reduced coupling. Thus, it would not change the reconstructed architecture. Another way to deal with this situation is the consideration of additional knowledge during the architecture reconstruction (“These classes should be assigned to the same component.”). In contrast to the rather subtle influence on the architecture reconstruction which is exerted by the other deficiencies, the *Inheritance between Components* deficiency requires the deliberate influence of the software architect to improve the reconstructed model.

5.4. Result Model

The architecture reconstruction is based on the Generalised Abstract Syntax Tree (GAST) which is extracted from the source code (see Section A.1). SoMoX creates a separate model to store the results of the architecture reconstruction process. This architectural model is an instance of the Service Architecture Meta Model (SAMM). Instances of the SAMM are also referred to as service architecture models (SAM). The SAMM contains elements like basic components, composite components, and connectors between them. It is explained in detail in Section A.2 of the Appendix.

The assignment of existing classes to the reconstructed components is an important result of the architecture reconstruction. This information is stored

in a third model: the Source Code Decorator (SCD) which connects the GAST and the SAMM. Details on the SCD meta model can be found in Section A.3

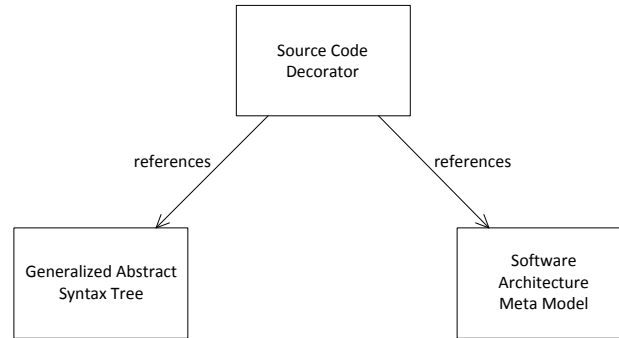


Figure 5.6.: Relationship between Source Code Decorator, GAST, and Service Architecture Meta Model (Figure adapted from [Tra11])

Figure 5.6 illustrates the relationship between the three models. The SCD references both the GAST and the SAMM. This way, no direct reference between GAST and SAMM is necessary.

Figure 5.7 shows an excerpt of an exemplary instance of the three result models. The figure illustrates how the result model instance for the running example might look. An excerpt of the GAST instance is shown on the left while an excerpt of the SAMM instance is shown on the right. Between the two models is an excerpt of the SCD instance which connects them.

The GAST instance shows the three classes `StoreQuery`, `Inventory`, and `Assets` from the running example. The SAMM instance contains two primitive components, `pc1` and `pc2`, and one composite component `cc1`. The SCD model contains three elements of the type `ComponentImplementingClassesLink` which connect the classes from the GAST to the components from the SAMM. In this case, the classes `StoreQuery` and `Inventory` are assigned to `pc1` while `Assets` is assigned to `pc2`.

5.5. Limitations

The architecture reconstruction technique that is the basis for the observations and analysis in this chapter is that of SoMoX. Other clustering-based architecture reconstruction approaches may use different metrics or may combine them in different ways. Thus, the analysis presented in this chapter is only valid for SoMoX.

It can however be argued that the qualitative observations, i.e. that design deficiencies can lead to an alteration of source code metrics which may, in turn, influence the result of the clustering, are valid for clustering-based architecture reconstruction approaches in general. Since metrics like the number of accesses

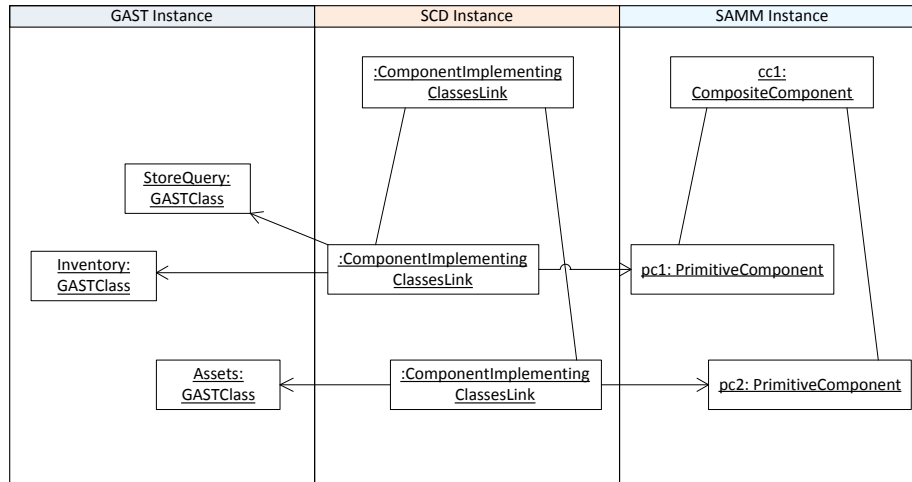


Figure 5.7.: Illustration of the relationship between exemplary result model instances (Figure adapted from [Tra11])

between elements or coupling are used in nearly every clustering-based architecture reconstruction approach [Kos00, DP09], those techniques will probably be similarly influenced by design deficiencies.

5.6. Conclusion

This chapter presented the clustering-based reconstruction of software architectures using the example of SoMoX. I showed how the metrics that are used to calculate the clustering depend on each other. Furthermore, I pointed out how design deficiency occurrences change the basic metric values used in the clustering. The interdependence of the metrics leads to the fact that these value changes are propagated up to the strategies that determine if elements are clustered together. Therefore, the introduction of just one reference may change the metric values such that the reconstructed architecture is significantly influenced.

The following chapter deals with the identification of components which are likely to contain such influencing deficiency occurrences.

6. Component Relevance Analysis

After the architecture has been reconstructed, it can be searched for deficiencies. However, the software architect has to decide where to begin this search. To support the architect in this decision, the *component relevance analysis* identifies components that are particularly relevant for a deficiency detection (Step 2 in Figure 4.1).

This chapter begins with a motivation as to why a component relevance analysis is necessary. The chapter's contributions and assumptions are described afterwards in Sections 6.2 and 6.3. Section 6.4 defines what exactly relevance means in the context of this chapter. Afterwards, the rationale and the calculation of the metrics used in the relevance analysis are discussed in Section 6.5. Finally, it is illustrated how a component's relevance can be determined from its individual metric values in Section 6.6. The limitations of the component relevance analysis are discussed in Section 6.7. Section 6.8 gives an overview of related approaches. The chapter is closed by a conclusion.

6.1. Motivation

After the architecture reconstruction, the architect is presented with an architecture that may contain a possibly large number of interconnected and nested components. In order to detect all deficiencies, the complete system would have to be searched now. However, such a comprehensive detection may be very time-consuming. For example, Simon et al. [SSL01] as well as Bauer and Trifu [BT04a] note that pattern detection does not scale well with the size of the system under study.

In the case of Archimetrix, the pattern detection tool Reclipse [vDMT10a, vDMT10b] is used for the deficiency detection. It uses a graph matching approach based on the system's abstract syntax graph (see Section 7). Searching large systems for deficiencies is a problem for Reclipse because subgraph matching is NP-complete in the general case [Epp95]. Reclipse tries to account for this problem by exploiting structural properties of the pattern specifications to confine the search space [NSW⁺02, NWW03, Nie04]. However, it still suffers from scalability issues.

In addition, searching the complete system for deficiencies may result in a large number of detection results. Being presented with thousands of deficiency occurrences can give the software architect a hard time in deciding which of the results are interesting and which are not.

In order to allow for a more efficient and focused detection of deficiency occurrences, I thus propose to use the reconstructed architecture to limit the search scope of the detection.

6.2. Contributions

The contributions to the reengineering process are as follows.

- The software architect can select an arbitrary subset of the recovered components, from single components to the complete system. The deficiency detection is only carried out for the selected components. On the one hand, this is significantly faster than searching the complete system. On the other hand, it allows the architect to focus on one component or one part of the system at a time. This prevents that he is overwhelmed by the large number of detection results.
- To support the architect in his choice of components, the relevance of the reconstructed components with respect to the deficiency detection can be calculated. To this end, this chapter defines what relevance means for components in the context of Archimatrix's deficiency detection.
- A systematic method to determine relevant components is presented. This allows for a faster and more focused detection of deficiency occurrences.
- The chapter proposes two relevance metrics which can be used for arbitrary systems. The rationale and the calculation of these two example metrics is discussed in detail.
- The proposed relevance metrics are only examples. The presented approach is designed to be easily extensible with new relevance metrics. This allows for a later extension, e.g. in order to adapt the relevance analysis to the architect's specific requirements.

6.3. Assumptions

The concepts presented in this chapter are based on a couple of assumptions:

- All aspects that are of interest for the relevance of a component have to be mapped to a numerical value between 0 and 1. This assumption allows for the definition of relevance metrics which calculate numerical values for a given component. Thus, all aspects that can be characterised on an ordinal scale are possible candidates for relevance metrics.
- The metric values are assumed to be proportional to the relevance of the component in question, i.e. the higher the metric value, the more relevant the component.
- For the calculation of the component relevance described in Section 6.6, all relevance metrics are assumed to be of equal importance.

All these assumptions apply for the metrics presented in Section 6.5.

6.4. Component Relevance

In order to determine the relevance of components, it has to be clarified what the term *relevance* means. In the context of this thesis, a *relevant component* is a software component which was recovered during the architecture reconstruction step (Step 1 in Figure 4.1) and which is a worthwhile input for the deficiency detection (Step 3 in Figure 4.1). This thesis proposes two example relevance metrics which can be used to determine which components are worthwhile in that sense.

Component Complexity A component is a worthwhile input for the deficiency detection if it is likely to contain deficiencies. It is generally in the software architect's interest to detect deficiencies in the system. Therefore, it is sensible to search deficiencies in places where they are likely to be found. Complex components are components which are hard to understand and maintain. When developers change these components, they are arguably more likely to introduce deficiencies in these components [Rie96, BB01, Gla03]. Thus, we consider complex components to be a worthwhile input for the deficiency detection. A study by Kafura and Reddy also suggests that complex components are those that are the best starting points for reengineering activities [KR87].

Closeness to Clustering Thresholds As explained in Chapter 5, deficiency occurrences can influence the architecture reconstruction. An architect who uses Archimetrix is interested in reducing this influence to get a clear picture of the system's concrete architecture. Therefore, components that are likely to change when a contained deficiency is removed can be considered a worthwhile input for the deficiency detection. To determine these components, the thresholds used during the architecture reconstruction have to be considered.

Section 5.2 explains how the decision to merge or compose component candidates is reached: The aggregated metric values v_m and v_c for a component candidate are compared to two threshold values t_m and t_c and the candidate is merged or composed when one of these thresholds is exceeded. However, a deficiency occurrence may influence the metric values such that t_m or t_c are just barely exceeded. Removing this deficiency will then probably prevent this composition or merge. Hence, the resulting reconstructed architecture will differ from the previously reconstructed architecture.

Therefore, components for which v_m or v_c were close to their respective thresholds during the architecture reconstruction are considered as relevant.

6.5. Relevance Metrics

Following the rationale from Section 6.4, I use a combination of two different *relevance metrics*: the *Complexity Metric* and the *Closeness to Threshold (CTT)*

Metric. These metrics were first presented in [Pla11] and can be calculated for any software architecture that was reconstructed as described in Chapter 5. Note that both metrics are heuristics, i.e. they can indicate which components may be relevant but they may also deliver incorrect results.

6.5.1. Complexity Metric

The *Complexity Metric* identifies complex components. Components consisting of many classes, attributes, methods and interfaces are hard to understand and are therefore difficult to maintain and to adapt. Thus, the risk of accidentally introducing design deficiencies when modifying these components is higher than in simple components [Rie96, BB01, Gla03]. Arguably, this problem will get even worse if these components are not thoroughly reengineered. This leads to the following assumption: the more complex a component, the more likely it is to contain design deficiency occurrences. This makes the complexity of a component significant to rate its relevance for the design deficiency detection.

To calculate the complexity, I use a slightly modified version of the Component Plain Complexity (CPC) metric as described by Cho et al. [CKK01]. It is expressed by the following formula:

$$\begin{aligned}
CPC(Comp) := & \#Classes(Comp) \\
& + \#Interfaces(Comp) \\
& + \sum_{c \in Classes(Comp)} \#Methods(c) \\
& + \sum_{c \in Classes(Comp)} \#Attributes(c) \\
& + \sum_{c \in Classes(Comp)} \sum_{m \in Methods(c)} \#Parameters(m)
\end{aligned} \tag{6.1}$$

The complexity of the component is determined by counting all classes and interfaces in the component. In addition, the number of methods and attributes in these classes is counted, as well as the number of parameters in all methods.

In order to obtain a relevance metric value between 0 and 1, the CPC value for each component is normalised by relating it to the complexity of the most complex component.

$$CPC(C)_{norm} = \frac{CPC(C)}{\max_{C_i \in allComponents} (CPC(C_i))} \tag{6.2}$$

This normalised Component Plain Complexity value can then be used during the component relevance calculation described in Section 6.6.

Note that the Component Plain Complexity is a very basic heuristic for the calculation of the component complexity. In the future, the complexity metric could be refined by taking metrics like McCabe’s Cyclomatic Complexity [McC76] or the design evolution metrics proposed by Kpodjedo et al. [KRG⁺11] into account.

6.5.2. Closeness to Threshold Metric

As explained in Section 6.4, components whose aggregated merge or compose value is close to the respective threshold during the clustering, are considered to be relevant. This is determined by the Closeness to Threshold (CTT) Metric. Components with an aggregated merge or compose value close to the respective threshold receive a high CTT value and are therefore rated as very relevant. The farther a component's aggregated metric values are from the threshold, the less relevant it is ranked by the CTT metric.

The calculation of the metric has to be explained in the context of the architecture reconstruction process. The architecture reconstruction is accomplished in a number of iterations (see Section 5.2). During each iteration, all currently reconstructed components are combined in pairs to form component candidates. For each of these component candidates, the aggregated merge and compose values v_m and v_c are calculated. These are then compared to the corresponding threshold values t_m and t_c . If a threshold is exceeded, the component candidate is converted into a component, otherwise it is discarded.

The CTT metric is meant to detect components whose contained candidates were closely above or below a threshold during the architecture reconstruction. For this, it has first to be determined what *close* means in this context. In Archimetrix, the software architect can define an appropriate margin ε . Components whose constituent candidates were closer than ε to a threshold during the clustering are deemed relevant. ε has to be set by the architect based on his experience. The default value which was also used in the validation is 0.2.

For the calculation of the CTT metric, all component candidates that were created during the architecture reconstruction are considered (no matter if they were converted into a component or discarded). For each component candidate, the closeness coefficient is calculated according to Formulas 6.3 and 6.4.

$$IsCTT_{Merge}(cc) := \begin{cases} 1 & \text{if } |v_m(cc) - t_m| < \varepsilon \\ 0 & \text{otherwise} \end{cases} \quad (6.3)$$

$$IsCTT_{Compose}(cc) := \begin{cases} 1 & \text{if } |v_c(cc) - t_c| < \varepsilon \\ 0 & \text{otherwise} \end{cases} \quad (6.4)$$

cc is the component candidate in question and ε is the margin in which the candidates merge or compose value must lie to be considered “close” to the threshold. If the candidate's aggregated merge or compose value is closer to the threshold than ε , its closeness coefficient is 1. Otherwise, it is 0.

v_m and v_c are calculated for all component candidates that are created during the reconstruction process. However, the architect is only interested in the

relevance of the components that are the results of the finished clustering process. To the architect, the component candidates are intermediary artefacts that he is not directly concerned with. In order to make a statement about the relevance of a concrete component $Comp$, it has to be determined which component candidates ended up being assigned to $Comp$ at the end of the reconstruction process. As shown in Figure 5.3, a component candidate consists of two sets of classes C_A and C_B . If these classes are – at least in part – assigned to $Comp$ is determined by a component's mapping coefficient for a given component candidate. Formula 6.5 shows the calculation of this mapping coefficient.

$$Mapping(cc, Comp) := \begin{cases} 0 & \text{if } \#(cc.C_A.Classes \cap Comp.Classes) = 0 \\ & \wedge \#(cc.C_B.Classes \cap Comp.Classes) = 0 \\ 1 & \text{if } \#(cc.C_A.Classes \cap Comp.Classes) \geq 1 \\ & \oplus \#(cc.C_B.Classes \cap Comp.Classes) \geq 1 \\ 2 & \text{if } \#(cc.C_A.Classes \cap Comp.Classes) \geq 1 \\ & \wedge \#(cc.C_B.Classes \cap Comp.Classes) \geq 1 \end{cases} \quad (6.5)$$

If $Comp$ contains neither classes from C_A nor from C_B , the mapping coefficient of component candidate and final component is 0. If at least one class from either of both sets C_A or C_B is assigned to the component, this is rated with a coefficient of 1. Finally, if classes from both C_A or C_B are found in the final component, the mapping 2.

Formula 6.6 shows the calculation of the Closeness to Threshold metric.

$$CTT(Comp) := \sum_{i \in Iterations} \left(\sum_{cc \in CCands_i} isCTT_{Merge}(cc) \cdot Mapping(cc, Comp) \right) + \sum_{i \in Iterations} \left(\sum_{cc \in CCands_i} isCTT_{Compose}(cc) \cdot Mapping(cc, Comp) \right) \quad (6.6)$$

The components closeness to threshold is calculated as a sum over all iterations. For all candidates in every iteration, the product of the closeness coefficient and the mapping coefficient is calculated. This way, candidates that are not close enough to the threshold are filtered out as their closeness coefficient is 0. Similarly, component candidates whose classes are not part of the component in question are filtered out because their mapping coefficient is 0.

In order to obtain a relevance value between 0 and 1, the CTT metric value

has to be normalised. For this, the greatest possible CTT value for all the component candidates is calculated as shown in Formula 6.7.

$$AllCCands_{max} := \sum_{i \in Iterations} \#CCands_i \cdot 2 \quad (6.7)$$

The number of component candidates for each iteration is multiplied by 2 because this is the maximum value a component candidate can achieve during the CTT calculation (The mapping coefficient is 2 when the final component contains classes from both parts of the component candidate. A candidate can either be composed *or* merged during one iteration but not both.). The CTT value is then normalised by dividing it by the maximum value as shown in Formula 6.8.

$$CTT_{norm}(Comp) := \frac{CTT(Comp)}{AllCCands_{max}} \quad (6.8)$$

6.6. Relevance Calculation

In order to calculate the relevance of a component, the values of the relevance metrics are mapped to an n-dimensional coordinate system. Each relevance metric represents one dimension. Thus, for the two relevance metrics presented in this thesis, a two-dimensional coordinate system suffices. Figure 6.1 illustrates fictional results of a component relevance analysis for the example system from Section 1.7.

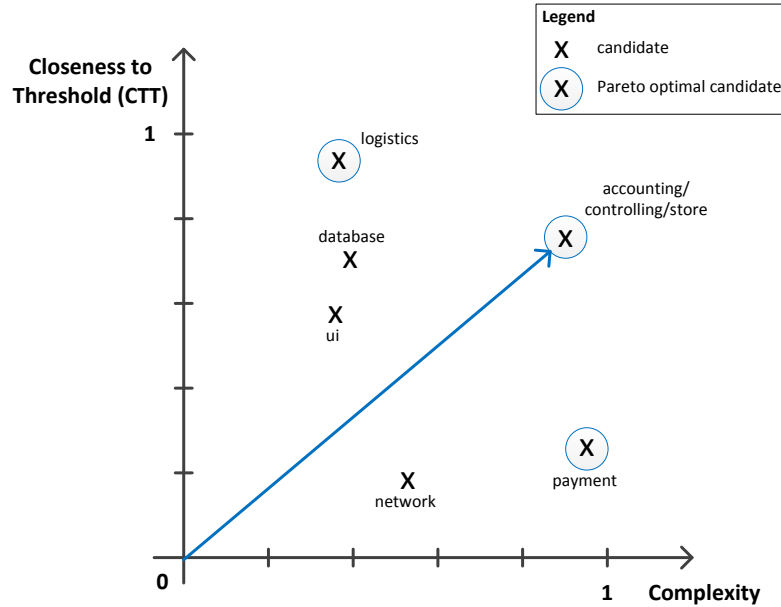


Figure 6.1.: Relevance analysis result calculation

The relevance values are visualised in a coordinate system. The x-axis represents the component complexity and the y-axis represents the Closeness to Threshold. Six components are positioned in the coordinate system. The component accounting/controlling/store represents the combination of the three components introduced in the running example (see Figure 3.3). They were reconstructed as one component because of the *Transfer Object Ignorance* occurrences contained therein.

To identify the most relevant components, the Pareto optimality [CCDJ10] is calculated for all components. A Pareto optimal set contains solutions that represent the best possible trade-off among a number of objectives. A solution is called Pareto optimal if and only if there is no solution that dominates it. Here, we use the *dominates* relation in a maximization context: A solution x dominates a solution y if and only if $\forall i \in [1..n], f_i(x) \geq f_i(y)$ and $\exists i \in [1..n]$ such that $f_i(x) > f_i(y)$. The components that are Pareto optimal with respect to all the relevance metrics are assumed to be good subjects for a design deficiency detection because they represent the best available combination of relevance values.

In Figure 6.1, the Pareto optimal components accounting/controlling/store, logistics and payment are marked with circles. They build up a Pareto front. Each component below the Pareto front (here: ui, database, and network) is dominated by the other components, hence, it is not Pareto optimal and therefore less interesting.

If several Pareto optimal solutions exist, as in the example in Figure 6.1, a further criterion to identify the most relevant component is required. For this purpose, the geometric distance to the origin is used as a heuristic: The higher this distance, the more relevant the corresponding component. Note that this is possible due to the proportionality between the metric values and the component relevance (see Section 6.3). The resulting distance value is normalised to a value between 0 and 1 in order to simplify the comparison.

Thus the relevance of a component is computed according to Formula 6.9.

$$Relevance(C) := \frac{\sqrt{\sum_{i=1}^n value_i^2}}{\sqrt{n}} \quad |value_i \in [0; 1] \quad (6.9)$$

where C is the component whose relevance is calculated, n is the number of ranking strategies, and $value_i$ is the value of ranking strategy i for C . In our example, accounting/controlling/store is the component with the largest relevance value and thus it is the most relevant component for the design deficiency detection.

This approach to calculate an overall relevance value is easily extensible. The Pareto optimality as well as the normalised geometric distance are computable for arbitrarily many dimensions. Thus, any number of metrics can be added to rate the relevance of a component given they meet the assumptions described in Section 6.3.

6.7. Limitations

When considering the results of the component relevance analysis, a number of limitations have to be kept in mind:

- The complexity metric is based on the assumption that large components contain many deficiencies. Although this heuristic is based on a lot of studies on this subject, e.g. [Rie96, BB01, Gla03], there may be cases where it produces wrong results. Sometimes, short algorithms may be more complex to understand and therefore harder to maintain than larger routines for simple tasks. Similarly, a component may be a maintenance nightmare due to missing documentation while a much larger component is easier to understand. There may also be cases where large components coincidentally do not contain any deficiencies but smaller components do.
- One motivation to confine the deficiency detection to single components is that it does not scale to large systems. However, one heuristic to find components that are good targets for the deficiency detection is the complexity metric. This may result in components that are complex and are therefore deemed relevant but which are on the other hand so large that the deficiency detection still takes a very long time.
- The relevance analysis points out components in which a deficiency detection is worthwhile according to the rationale presented in Section 6.4. However, the software architect may want to select components based on different considerations. For example, he could be interested in components that are related to the system's user interface or he may want to find deficiencies that are easy for him to remove. In these cases, the current relevance metrics are not particularly helpful for him. It may however be a challenge to create specific relevance metrics that tackle these problems as these scenarios are strongly dependent on the architect's particular goals and abilities.
- One assumption in Section 6.3 states that the relevance metrics are regarded to be equally important. However, if one metric was found to be more or less important in comparison to others (e.g. due to one of the other limitations discussed here), a possibility to assign weights to certain metrics could be useful. This would allow the architect to flexibly adjust the relevance rating according to his requirements.
- Focussing the deficiency detection on a subset of components may lead to deficiency occurrences being missed by the detection. Design deficiencies often involve multiple classes. The *Transfer Object Ignorance* deficiency introduced in Chapter 3, for example, involves three classes. The classes belonging to a given deficiency occurrence D may be assigned to different components during the architecture reconstruction. If not all of these components are then selected as an input for the deficiency detection, D will not be detected. On the other hand, the purpose of the Archimetrix

process is not necessarily to detect and remove all deficiency occurrences but to find the relevant ones.

6.8. Related Approaches

Arthur suggests to use a Pareto analysis to “identify the 20% of the programs that consume 80% of the resources” [Art88]. For that, he suggests to create a table that lists all the modules in a program and assigns to each module the number of failures, the number of defects, the number of enhancements, and the time that the module has been worked on. From this table, it can be determined, which modules should be subject to perfective maintenance efforts, e.g. to reduce the failure density. In contrast to Archimatrix, which can automatically determine the relevance of the reconstructed components, the process described by Arthur is a manual one.

A lot of research went into the identification of defect-prone classes. If a class is known to be prone to the introduction of defects, it can be tested more thoroughly, for example. Basili et al. used object-oriented metrics to predict defects [BBM96]. More recently, Kpodjedo et al. [KRG⁺11] as well as Khomh et al. [KDGA12] have investigated the influence of anti patterns on the defect-proneness of classes. However, none of these approaches considers components as first-class entities in their analyses.

6.9. Conclusion

This chapter presented a concept for the rating of a software component’s relevance. This rating can support the software architect in selecting components from the reconstructed software architecture in which he can perform a design deficiency detection. On the one hand, this selection will limit the search scope of the deficiency detection and therefore allow for a faster search. On the other hand, it allows the architect to examine parts of the system in which a deficiency detection will probably yield meaningful results.

The component relevance analysis is designed to be an extensible mechanism. Relevance metrics that fulfil the assumptions given in Section 6.3, can be added to the analysis without changing the general concept.

7. Design Deficiency Detection

When the relevant components in the reconstructed architecture are identified, the detection of deficiencies in these components can begin (Step 4 in Figure 4.1). In Archimetrix, the deficiency detection is accomplished by using the Reclipse Tool Suite which was developed by the Software Engineering Group at the University of Paderborn in earlier work [NSW⁺02, Nie04, Wen07] [vDMT10a, vDMT10b, vDT10].

Reclipse bases its detection both on structural and behavioural aspects of a pattern. The static analysis is completely based on the static structure of the system. The dynamic analysis collects traces of the system behaviour at run-time and analyses them. Archimetrix was developed and validated with a focus on the static analysis. There are however first ideas to improve the dynamic analysis which have also been developed in the course of this thesis. These ideas are also sketched in this chapter.

The chapter is structured as follows. Section 7.1 sums up the contributions of this chapter while Section 7.2 discusses the assumptions of the deficiency detection. An overview of the complete pattern detection process with Reclipse, including static and dynamic analysis, is given in Section 7.3. Section 7.4 explains how the structural analysis is integrated with the architecture reconstruction such that it can be focused on a selection of relevant components. Section 7.5 then shows why the current trace collection for the dynamic analysis is insufficient and sketches a solution to this problem. The limitations of the pattern detection are detailed in Section 7.6 while Section 7.7 concludes the chapter.

7.1. Contributions

- This chapter illustrates how the pattern detection approach of Reclipse can be integrated with the architecture reconstruction of SoMoX. This is accomplished through a number of auxiliary patterns referencing different parts of the result model.
- This chapter also sketches a new method to systematically and comprehensively obtain traces for the dynamic analysis. Normally, the behavioural traces are generated by executing the system under analysis in a more or less systematic manner. Instead, I propose to use symbolic execution to generate a whole set of traces which comprises the complete behaviour.

7.2. Assumptions

- For the explanations in this chapter, I assume that the architecture reconstruction (Step 1 in Figure 4.1) has been successfully executed. In particular, I assume that a structurally consistent result model of the analysed system consisting of GAST, SAM, and SCD is available.
- Another prerequisite for the deficiency detection is the existence of a number of deficiency formalisations for the system under analysis. The formalisations may have been created following the process proposed in Section 4.4 or they may be reused from a pre-existing catalogue. The underlying type graphs of the formalisations are assumed to be the GAST meta model, the SAMM, and the SCD meta model (see Appendix A).

7.3. Pattern Detection with Reclipse

This section briefly explains the complete pattern detection process with Reclipse. Reclipse was developed in earlier work [NSW⁺02, Nie04, Wen07] [vDMT10a, vDMT10b, vDT10] and is not a contribution of this thesis. Nevertheless, this process is important for the explanation of the actual contributions of this thesis in this area.

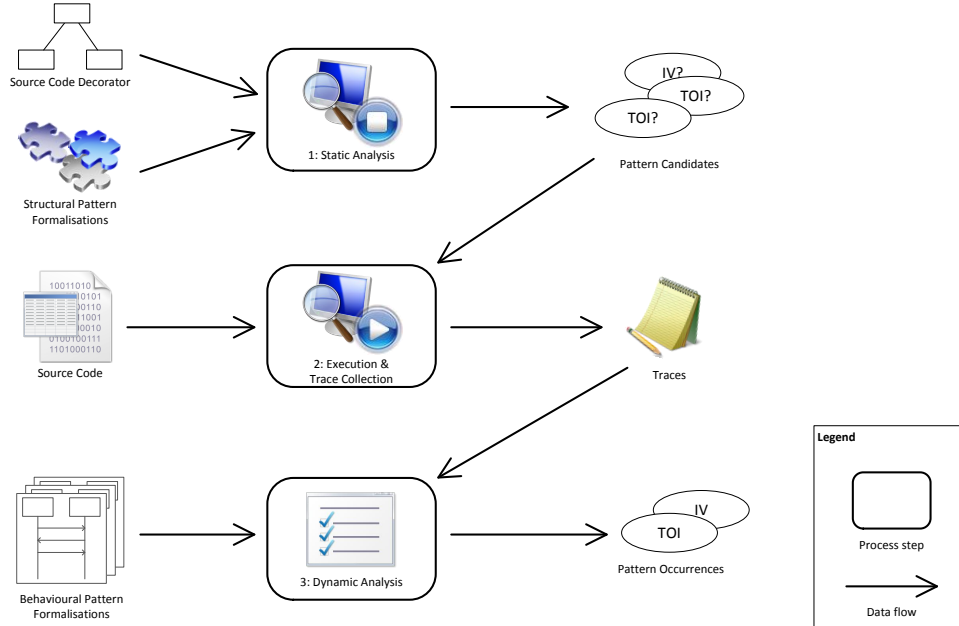


Figure 7.1.: Process for the pattern detection with Reclipse

Figure 7.1 gives an overview of the pattern detection process. The process comprises three steps.

First, a static analysis is executed to identify pattern candidates. The static analysis employs graph matching to identify pre-defined object patterns in a given host graph. In Archimetrix, the source code decorator of the system is the host graph of the pattern matching. The second input of the static analysis is a catalogue of structural pattern formalisations. The static analysis uses only the static information available from the SCD (and the GAST and SAM which are referenced by it, see Section 5.4). The program under analysis is not executed. The static analysis yields a set of pattern candidates, i.e. parts of the input model whose structure matches the pattern formalisations.

The second step of the Reclipse process is the execution of the system under analysis. For this, the source code and the previously detected pattern candidates are necessary. The source code is executed and the behaviour of the system is traced. Traces are only collected for methods that play a role in one of the pattern candidates, i.e. for methods that are annotated as a part of the pattern candidate during the static analysis. The collected traces are stored in a file.

During the third step, the dynamic analysis, the traces of all candidates are compared to the respective behavioural pattern formalisations. Candidates whose behaviour conforms to the expected pattern are accepted as pattern occurrences. Candidates who violate the expected behaviour are rejected. If the trace of a candidate contains no conclusive evidence about its behaviour, the candidate is neither rejected nor accepted. In this case, the software architect has to decide if the candidate is a true or a false positive.

Archimetrix was developed and validated with a focus on the static analysis. The extension of the dynamic analysis which is presented later in this chapter is still in its early stages. It has therefore been exempt from the validation presented in Chapter 10.

The following subsections explain the three steps in more detail.

Structural Analysis Reclipse employs a graph matching approach for the detection of structural patterns [NSW⁺02]. The structural patterns are graph patterns which formalise the structural properties of the design patterns or design deficiencies to be detected. Graph matching aims at finding occurrences of one or more graph patterns in a host graph. In Reclipse, the host graph of the matching can for example be the abstract syntax graph of a software system. In Archimetrix, the abstract syntax graph is not sufficient because the reconstructed components have to be considered during the graph matching. Therefore Archimetrix uses the source code decorator as a host graph which references both, the GAST and the SAM. This topic is discussed in Section 7.4.

Reclipse detects occurrences of a given pattern formalisation – which is also a graph – by searching an isomorphic matching between that pattern formalisation and the host graph. When a pattern occurrence in the host graph is detected, Reclipse creates an annotation which marks that pattern occurrence.

On a technical level, the detection is accomplished by generating story diagrams [vDHP⁺12] from the structural pattern formalisations. These story diagrams are then executed through interpretation [Foc10]. The result of the

static analysis is a set of pattern candidates.

If Reclipse was to detect deficiencies in the example system introduced in Chapter 1, it would detect a number of candidates. Reclipse offers different visualisations of the detected candidates [PvDT11]:

- A *class diagram view* which shows the different classes and methods that are participating in an occurrence.
- A *pattern view* which shows the formalisation that corresponds to the occurrence and hides all elements from the formalisation that were *not* detected for the occurrence.
- An *host graph view* which shows only that part of the host graph that in which the occurrence was detected.

These different views can help the software architect in deciding if a detected occurrence is a true positive or a false positive. Figure 7.2 gives an example of the class diagram view.

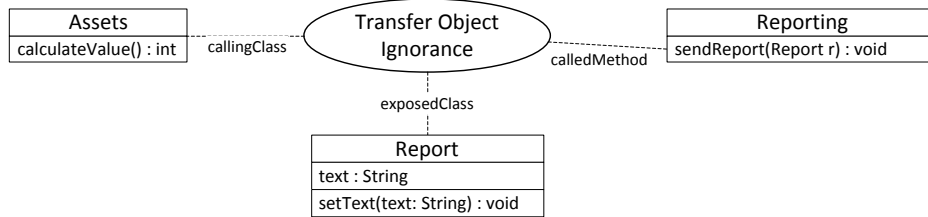


Figure 7.2.: Candidate for an occurrence of the *Transfer Object Ignorance* deficiency

The figure shows a candidate for an occurrence of the *Transfer Object Ignorance* deficiency which was introduced in Section 3.3. The specific occurrence in the figure is the occurrence no. 2 from page 40. It consists of the class *Assets* which is identified to play the role of the calling class. The class *Report* appears to be the class that is exposed by the deficiency. Finally, the method *send Report* is the method that is called by the calling class.

This would be one of the candidates that would be observed during the next step, the trace collection.

Trace Collection During the execution of the system under analysis, the candidates that were detected during the static analysis are observed. The execution can either be accomplished by executing a given test suite for the system or by executing the program manually and triggering typical functionality. The behaviour of the candidates is logged and stored in traces which are afterwards analysed in the dynamic analysis step. Tracing the complete behaviour of a system is, of course, impractical due to the huge amounts of data that would

be created. Instead, Reclipse only traces the relevant behaviour of the detected candidates. Only methods which are marked by the static analysis and which appear in the behavioural formalisation of the pattern candidates are traced [MW05].

Behavioural Analysis For the behavioural formalisation, Reclipse uses a DSL that is syntactically similar to UML sequence diagrams [Obj10]. The DSL was developed by Wendehals [Wen04, Wen07]. It was later extended in a Bachelor's thesis by Platenius [Pla09] [vDP09].

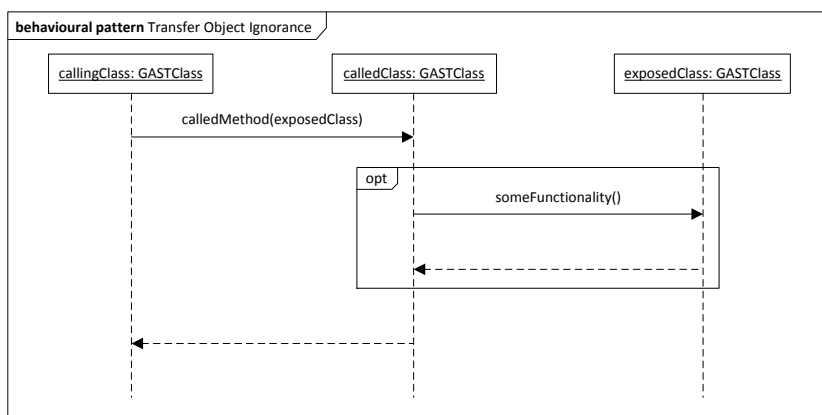


Figure 7.3.: Behavioural formalisation of the *Transfer Object Ignorance* deficiency

Figure 7.3 shows the behavioural formalisation of the *Transfer Object Ignorance* deficiency's general form. The object types and message names refer to identifiers from the structural formalisation. It contains the three objects `callingClass`, `calledClass`, and `exposedClass` which are all of the type `GASTClass` (as specified in the structural formalisation). The expected interaction between these objects would be as follows.

The `callingClass` calls the `calledMethod` of the `calledClass` and passes an instance of the `exposedClass` as an argument. Afterwards, the `calledClass` invokes the method `someFunctionality` on the `exposedClass`. This interaction is contained in a combined fragment with the *opt* operator signifying that the interaction is optional. If the method `someFunctionality` is not invoked, a detected candidate may still be a true positive occurrence of the *Transfer Object Ignorance* deficiency.

Note that Reclipse makes no assumptions about methods that are not used in the behavioural formalisation. Between the invocation of `calledMethod` and `someFunctionality`, arbitrary methods may be called during the execution. This has no influence on the interaction that Reclipse expects from the pattern candidate.

The expected behaviour of a particular candidate at run-time can be derived from its general behavioural formalisation. For each candidate, the general class

and method names from the formalisation are replaced with names of elements that play the corresponding roles in the candidate.

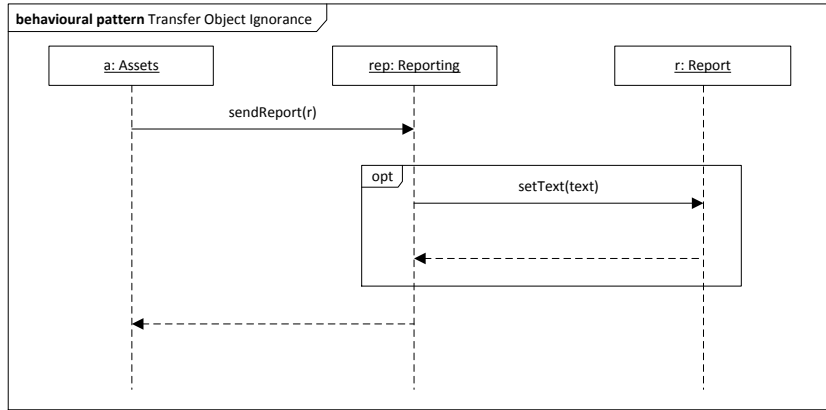


Figure 7.4.: Expected behaviour for the candidate from Figure 7.2

If the candidate from Figure 7.2 is indeed an occurrence of a *Transfer Object Ignorance* deficiency (instead of being just structurally similar), it is expected to behave as shown in Figure 7.4. An instance *a* of the class *Assets* is expected to call the method *sendReport* of the instance *rep* of the *Reporting* class and pass an instance *r* of the class *Report* in the process. Thereby, *rep* would obtain a reference to *r* which it should not have according to the system architecture. Afterwards, *rep* could invoke arbitrary methods of *r*. In this formalisation, *rep* could call the *setText* method of *r*. In accordance with the general behavioural formalisation, the latter call is optional. For this candidate, calls of the method *sendReport* of all the instances of *Reporting* and calls of *setText* of *Report* instances would be traced.

The dynamic analysis checks if the candidates behave according to their behavioural patterns. This analysis is based entirely on the collected traces. If the traces contain evidence that a candidate behaves only as specified by the pattern, i.e. the relevant method calls occur in an order that matches the pattern, the candidate is accepted as a true positive. If the traces show that a candidate behaves not in accordance with the behavioural pattern, the candidate is rejected, i.e. it is assumed to be a false positive. In the example, *rep* might call *setText* on *r* without *sendReport* being called earlier. This would for example be an indicator that the candidate is a false positive. If the trace does not contain conclusive evidence of either of these behaviours, e.g. because neither *sendReport* nor *setText* are being called, the dynamic analysis cannot determine if the candidate is a true or a false positive. Such inconclusive behaviour might, for example, occur when no methods of the candidate are called during execution. In this case, the dynamic analysis cannot reach a decision if the candidate is a true or a false positive and the software architect may have to make this decision on his own, e.g. by manually inspecting the candidate.

An in-depth explanation of the pattern detection with Reclipse is given in [Wen07].

7.4. Integrating the Deficiency Detection with the Architecture Reconstruction

In this thesis, Reclipse’s pattern detection is used for the detection of design deficiencies in a reconstructed software architecture. Archimetrix increases the scalability of the deficiency detection by confining it to certain components (see Chapter 6).

Travkin developed a concept for the integration of the deficiency detection with the architecture reconstruction in his master’s thesis [Tra11]. This section describes how a scalable deficiency detection which can take architectural elements into account can be realised.

7.4.1. Input Model for the Deficiency Detection

Chapter 5 explains the architecture reconstruction process of SoMoX. SoMoX bases its analysis on the GAST of the system and creates a separate architecture model, the Service Architecture Model (SAM). A third model, the source code decorator (SCD) is used to connect the two models and relate, e.g. classes to components. In Section 5.4, this result model of the architecture reconstruction is explained in more detail.

Until now, the input model for the pattern detection with Reclipse has always been an abstract syntax tree. However, this is not sufficient in Archimetrix because the deficiencies also contain elements from the reconstructed architecture. As the SCD references both, the GAST and the architectural model, it is the ideal input model for the deficiency detection. By connecting the GAST and the SAM, it creates one large host graph for the pattern matching. This allows to use Reclipse for the deficiency detection without modifying the matching algorithms.

7.4.2. Auxiliary Component Patterns

The formalisation of design deficiencies for component-based architectures often necessitates statements about the relationship between components and the elements contained therein. For example, it may be necessary to express that two classes are contained in different components. This is the case in the formalisation of the *Transfer Object Ignorance* deficiency (see Figure 4.3). It specifies that the `callingClass` and the `exposedClass` should reside in the same component (c1) while the `calledClass` should be contained in a different component (c2). This formalisation uses the *Component* auxiliary pattern described below.

In order to simplify the specification of such properties, a number of auxiliary patterns has been defined. They formalise the composition of components and the containment of classes in these components. They can then be easily used in deficiency formalisations.

Component Composition Patterns

This section presents three patterns which formalise the composition of components. The first pattern is an abstract pattern that defines the roles that have to be annotated if a component composition is detected. The other two patterns are concrete extensions of this patterns and formalise direct and indirect composition, respectively.

Figure 7.5 shows the *abstract pattern Component Composition*. An abstract pattern is a pattern which is not meant to be matched itself but which defines elements that shall be annotated by concrete patterns that extend it. If an abstract pattern is used as a sub pattern in a formalisation, every pattern extending it can be substituted for it during the matching process (similar to polymorphism in object-oriented programming languages).

The *Component Composition* pattern defines that a component composition exists between one `ComponentImplementingClassesLink` and a set of these links. They are elements of the Source Code Decorator and reference a component from the SAM on the one hand and number of classes from the GAST on the other hand. The single `ComponentImplementingClassesLinks` is annotated as the parent component while the set of links represents all the `ComponentImplementingClassesLinks` which point to the subcomponents of the parent component. Because it is an abstract pattern, it does not specify if the subcomponents are directly or indirectly contained in the parent component. This is specified by the two concrete sub patterns *Direct Composition* and *Indirect Composition*.

The *Direct Composition* pattern is shown in Figure 7.6. Because it is a sub pattern of *Component Composition*, it annotates exactly the same roles. The only difference to *Component Composition* is the direct connection between parent and the subComponents. Occurrences of this pattern will be created everywhere in the SCD where components are directly contained in other components (represented by the `ComponentImplementingClassesLinks`, here)¹.

In contrast to the *Direct Composition* pattern, the *Indirect Composition* pattern shown in Figure 7.7 represents situations where components are *not* directly contained within each other. Instead, it covers all cases where at least one intermediate component exists between the parent and the subComponents. The subComponents are directly contained in an intermediate middleComponent as represented by the *Direct Composition* pattern. The middleComponent is in turn contained in the parent component. Note that this containment is modelled by the abstract *Component Composition* pattern. Therefore, the containment of the middleComponent in the parent component may either be direct or indirect. This way, indirect composition hierarchies of arbitrary size can be recognised and annotated by this pattern.

¹The semantics of annotating a set object in Reclipse are as follows: When an occurrence of this pattern is detected, one link from the annotation to the `ComponentImplementingClassesLink` representing the **parent** component will be created. In addition, links from the annotation to *all* `ComponentImplementingClassesLinks` that are referenced via the subComponents link will be created.

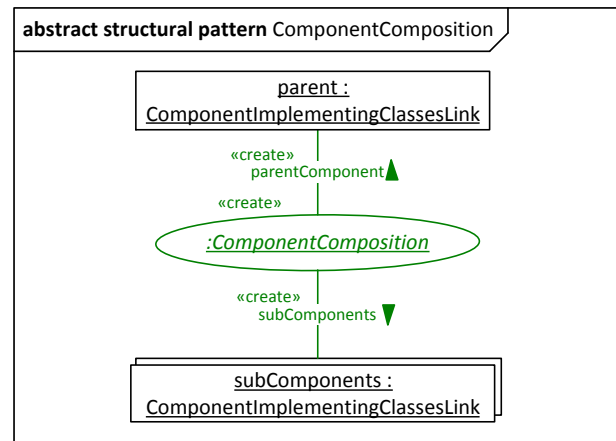


Figure 7.5.: Structural formalisation of the abstract *Component Composition* pattern

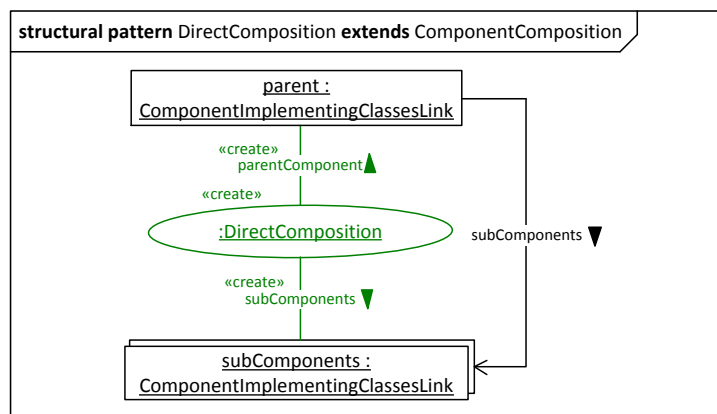


Figure 7.6.: Structural formalisation of the *Direct Composition* pattern

Class Containment Patterns

This section presents three patterns which formalise the containment of classes in components. Similar to the component composition patterns, they consist of one abstract and two concrete patterns. The abstract patterns defines the roles and the concrete patterns formalise the direct and indirect containment of classes in components, respectively.

The abstract *Component* pattern is shown in Figure 7.8. It specifies two roles: *classes* which annotates a set of classes from the GAST and *component* which is a *ComponentType* from the SAM.

The *Direct Component Classes* pattern (see Figure 7.9) extends the *Component* pattern. It formalises the direct containment of a set of classes in a component. This is achieved by matching a *ComponentImplementingClassesLink* from the SCD as well as the *ComponentType* and all *GASTClasses* it references.

A class is always directly contained in exactly one component. However, if that component is the subcomponent of a parent component, the class is also indirectly contained in the parent component. The *Indirect Component Classes* pattern depicted in Figure 7.10 captures this situation, annotating all indirect containments of classes in components. The outermost parent component is represented by the component object. Its corresponding *ComponentImplementingClassesLink* is matched as *parentComponentLink*. The *parentComponentLink* references the directly contained classes of the component which are not of interest here. Instead it is used to navigate to the contained subcomponents of the parent component. The *subComponent* link references the classes which are indirectly contained in the parent component. The composition between the parent component and its subcomponents is represented by a *Component Composition* pattern which again allows for direct or indirect composition. The set fragment caters for the possibility that the parent component contains several subcomponents.

7.5. Improved Trace Collection through Symbolic Execution

In order to improve the detection results of the static analysis, the run-time behaviour of the system under analysis can be taken into account. Reclipse provides the means to analyse a system's behaviour and relate this dynamic analysis to the static analysis results [Wen07]. For the most part, this thesis is focused on the static analysis for the detection of design deficiencies. However, in the future, the dynamic analysis could also be integrated into the Archimetric process. Section 7.3 presented a possible behavioural formalisation for the *Transfer Object Ignorance* deficiency.

This section presents an idea for an improved trace collection which could lead to better dynamic analysis results. Since this extension is still in an early conceptual phase, the approach is not yet implemented as a part of the prototype presented in Chapter 10. Therefore, the improved trace collection is also exempt from the validation of Archimetric. Nevertheless, the ideas sketched in

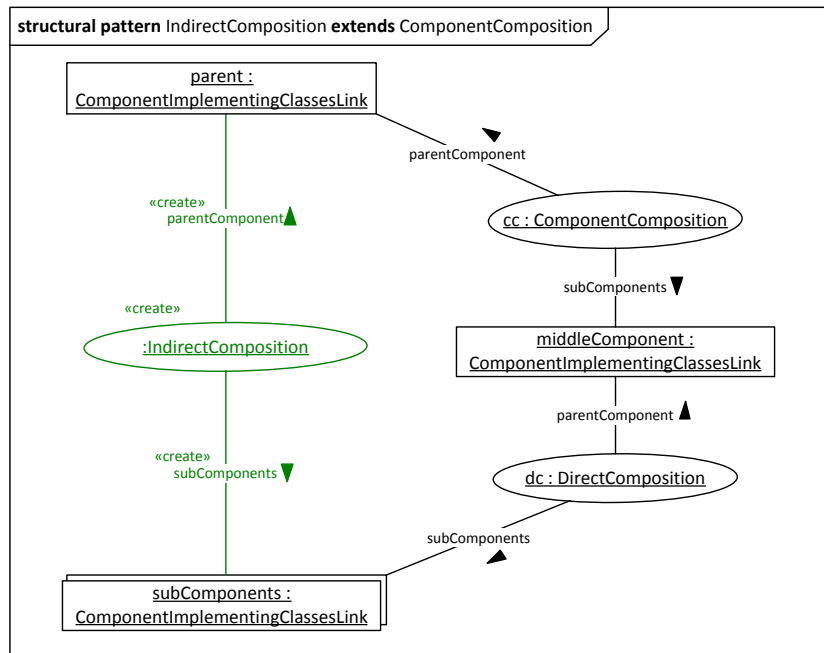


Figure 7.7.: Structural formalisation of the *Indirect Composition* pattern

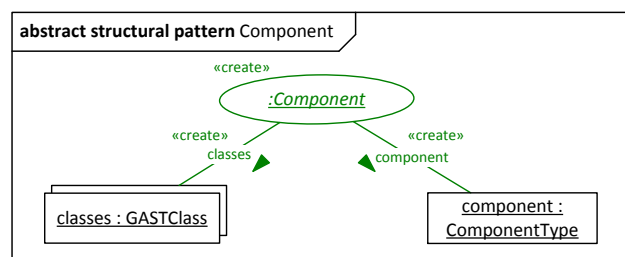


Figure 7.8.: Structural formalisation of the abstract *Component* pattern

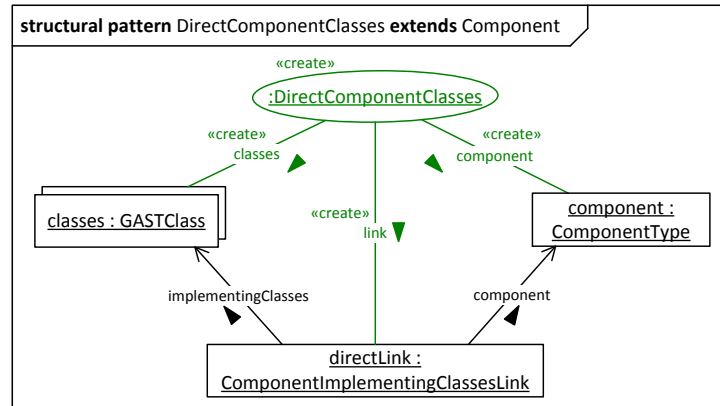


Figure 7.9.: Structural formalisation of the *Direct Component Classes* pattern

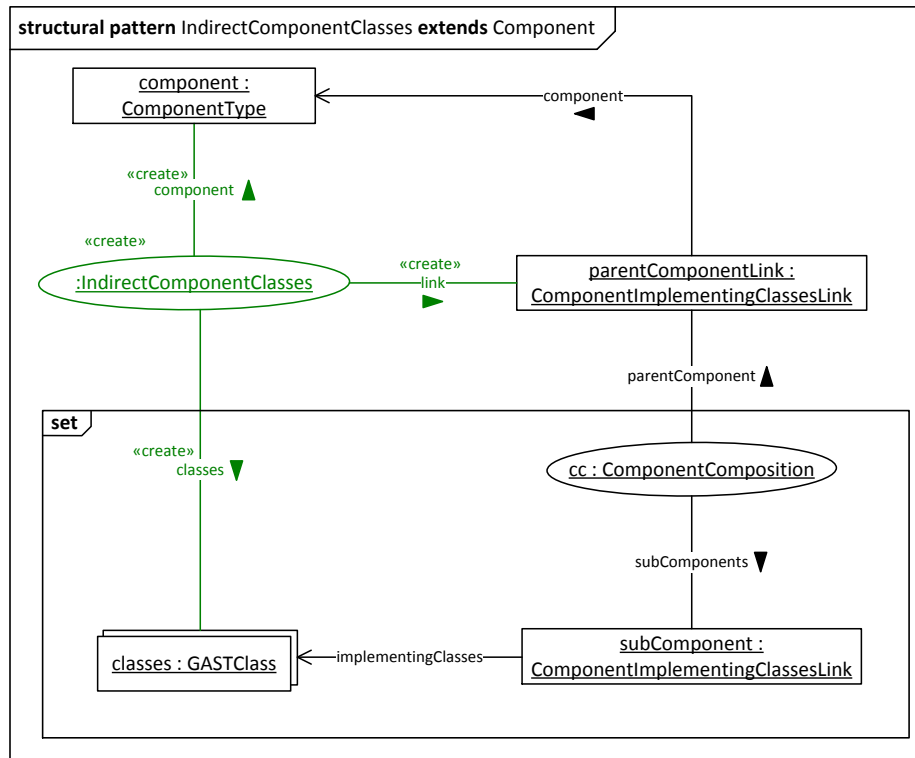


Figure 7.10.: Structural formalisation of the *Indirect Component Classes* pattern

this section can be useful for future extensions of Archimetric. They are based on a Bachelor's thesis by Volk [Vol10] and the resulting publication [vD11].

7.5.1. Shortcomings of the Trace Generation through Concrete Execution

To analyse the run-time behaviour of a system, it has to be traced during execution. These traces are an input for the dynamic analysis. In Reclipse, traces are either collected through the execution of test cases (if available) or through performing typical user interaction on the running system. This method of collecting traces has some major drawbacks: It can only cover part of the system's behaviour. This part is determined in a non-transparent and, in case of user interaction, possibly random way through the execution of the system.

Because the program execution can only cover part of all possible behaviour, the collected traces do not necessarily contain behavioural data on the interesting candidates. In this case, it is very difficult to actively obtain that data because there is no straightforward way to get the system to execute certain methods. The actual system behaviour can normally be influenced by the input or by direct interaction with the system, e.g. via a graphical user interface. The software architect on the other hand only knows the classes and methods which play a part in the corresponding pattern candidate. Without an in-depth analysis of the system's source code, it is not possible to determine which interaction with or input into the system will trigger the execution of a particular candidate.

If the collected trace does not contain behavioural data for a certain candidate, the architect has no choice but to collect more data. He can either interact randomly with the system to trigger the desired behaviour by chance or he systematically tries to cover as much interaction as possible. Neither of these heuristics guarantees to produce the desired data. In addition, the former approach does not allow to repeat the data collection in a systematic way. The latter approach requires meticulous logging of the user interaction and input data in order to be repeatable.

Furthermore, the current way of collecting traces does not allow definitive conclusions about the candidates. The only conclusion the architect can draw is that a given candidate behaves according to the behavioural pattern *for the part of the program that was actually executed*. The candidate might violate the pattern for another execution. The software architect, however, would never know because he can only base his judgement on the collected trace data.

Therefore, a more systematic and comprehensive way to collect the traces for the dynamic analysis could greatly improve the dynamic analysis results.

7.5.2. Systematic and Comprehensive Trace Generation through Symbolic Execution

The basic idea for a more comprehensive trace generation is to use symbolic execution. Symbolic execution as proposed by King allows to reason about

classes of program executions instead of individual, concrete executions [Kin76]. This is accomplished by assigning symbolic values instead of concrete values to variables and evaluating the different possible control flows. Monitoring a symbolic execution of a program can therefore yield a number of traces that cover the complete possible program behaviour. Consider this simple example program.

```
public void m1(int x) {  
    if (x == 5)  
        m2();  
    else  
        m3();  
}
```

A concrete execution of `m1` would lead to either `m2` or `m3` being called, depending on the value of `x`. Hence, the collected trace would either contain `m1` and `m2` *or* `m1` and `m3`. Executing `m1` symbolically instead (with `x` as a symbolic variable) would generate two traces: One with `x = 5` and one with `x ≠ 5`. In the first case `m1` and `m2` would appear in the trace, in the second case `m1` and `m3` would be traced.

I propose to use symbolic execution to generate traces for pattern candidates detected by Reclipse. First experiments which integrate Symbolic PathFinder, an extension for the Java Source Code model checker Java PathFinder (JPF) [PR10], into Reclipse show the general feasibility of the idea [Vol10] [vD11].

7.5.3. Interpreting the Generated Traces

Currently, the concrete execution of programs only allows for very imprecise conclusions about the pattern candidates. Whether a candidate was a true or false positive can only be answered with regard to the actually executed part of the system. The use of symbolic execution enables the software architect to get a much clearer picture.

Because all possible behaviour of the system is executed (at least theoretically, see Section 7.6.2) and the relevant parts are traced, a more definite statement can be made for each candidate. Even if abstraction and inaccuracy of the symbolic execution may miss some execution paths, the data is much more comprehensive than for concrete execution. The following three cases can be distinguished:

1. If all traces of a candidate only show behaviour which complies with the corresponding pattern, it is a true positive.
2. If at least one trace violates the behavioural pattern, a candidate has the potential to behave incorrectly. This means it is either a false positive or at least a flawed implementation of the pattern in question.
3. In the third case, no trace complies with the behavioural pattern but no trace violates it either. This can happen when, e.g. only one of the methods of a candidate is executed at all. This may be allowed by the

pattern but because no other relevant methods are executed afterwards, the pattern behaviour is neither completed nor violated. In this case, the candidate can also be regarded as a false positive because it never shows the complete correct behaviour.

7.5.4. Related Approaches

Symbolic execution is applied for a multitude of problems in different domains. While the method was originally conceived for the generation of test input [Kin76], it is now also applied in fields like the behavioural verification of safety-critical, embedded systems [LMV05]. The Java PathFinder extension Symbolic PathFinder [PR10] was developed for test generation and correctness checking of multi-threaded programs.

Most dynamic pattern detection approaches rely on the concrete execution of programs for the collection of traces [HHHL03, SH08]. De Lucia et al. [dLDGR10] apply a systematic approach for the checking of behavioural patterns. Similar to the idea sketched here, they begin with a static analysis to identify pattern candidates. Then, they use the model checker SPIN to analyse whether the candidates can possibly show the desired behaviour. This way, false negatives can be removed from the set of candidates. For this step, the authors translate the behavioural patterns into LTL expressions and the system into a Promela specification. Afterwards, a straight-forward dynamic analysis is carried out for the remaining candidates to verify if they actually show the expected behaviour at run-time. This last step has the same disadvantages as other approaches which employ concrete execution for the collection of traces.

7.6. Limitations

In this section, first the limitations of the deficiency detection in general are discussed. The subsequent section then deals with the limitations of the improved trace generation.

7.6.1. Limitations of the Deficiency Detection in Software Architectures

- The restriction of pattern detection to a subset of components has the consequence that the sum of detected pattern occurrences for this subset and for its complement may not be equal to the number of detected occurrences for a detection run on the whole system. Patterns that would be detected between classes which are assigned to different components are not detected if not all these components are part of the selected subset. As a consequence, some deficiencies might be overlooked if the software architect investigates different parts of the system one after the other. On the other hand, the deficiency detection might be significantly faster due to this restriction. The software architect has to be aware of this trade-off. If he wants to be sure to detect all deficiency occurrences in the system, a detection run on the system as a whole is inevitable.

7.6.2. Limitations of the Improved Trace Generation

- The most straightforward approach to the generation of execution traces is the symbolic execution of the system's main method. This way, all possible paths through the system are evaluated and all possible traces for the detected pattern candidates can be collected. This approach obviously suffers from a high computational complexity and also generates extremely large amounts of trace data. In addition, the symbolic execution of programs which use frameworks like Eclipse is problematic because the framework code is not of interest for the analysis.
- During the static analysis, a large number of pattern candidates can be detected by Reclipse. As explained in Section 7.3, only method calls that are part of the behavioural patterns are recorded in the traces. Still, generating all possible traces for every candidate can yield an impractically large amount of data. To alleviate this problem, the software architect could select a subset of the statically detected pattern candidates. During the symbolic execution, only methods of these selected candidates are recorded in the traces which would reduce the trace files to manageable sizes.
- User interaction constitutes a challenge for symbolic execution. While textual interaction via a command line can still be covered by treating the textual input as a symbolic variable, graphical user interfaces cannot be simulated easily. JPF offers the library `jpf-awt` to abstract away the rendering aspects of the GUI and still preserve the control flow of AWT- and Swing-based GUIs. The implementation is still rudimentary though, so the library does not work for all applications. However, it generally allows to apply the approach to GUI-based systems.

7.7. Conclusion

This chapter gave an overview of the pattern detection capabilities of Reclipse. It also described how the result model of SoMoX can be used as an input model for Reclipse such that architecture-related design deficiencies can be detected. Although Archimetrix mainly focuses on the static analysis for the deficiency detection, I also explained the dynamic analysis and sketched an approach to improve its trace collection.

The result of the deficiency detection is a set of deficiency occurrences. Archimetrix can support the software architect in the decision which of the detected occurrences should be removed first. This design deficiency ranking is presented in the next chapter.

8. Design Deficiency Ranking

The design deficiency detection yields a list of detected deficiency occurrences. However, depending on a number of factors such as the size of the system under study, this list may be very long [GD12].

Often, the limitation of resources like time, money, or manpower only allows the deficiency occurrences with the highest impact to be tackled [BK07]. In order to support the software architect in the decision which of the detected deficiency occurrences should be removed, Archimetric can rank them based on their severity (Step 4 in Figure 4.1). Similar to the component relevance analysis, the design deficiency ranking uses several metrics to determine the rank of a design deficiency occurrence.

Section 8.1 details the contributions of this chapter while the assumptions of the deficiency ranking are explained in Section 8.2. The ranking metrics are presented in Section 8.3. Afterwards, the calculation of the overall rank is covered in Section 8.4 and Section 8.5 discusses the limitations of this approach. Related work is reviewed in Section 8.6 while Section 8.7 concludes the chapter.

8.1. Contributions

- Building on the example ranking metrics, I propose a method to rank the detected deficiency occurrences based on the captured metric values. This rank is represented by a number between 0 and 1 and supports the architect in judging the severity of the detected deficiency occurrences.
- In this chapter, I present a number of example ranking metrics which measure several severity-related aspects of a deficiency occurrence.
- Similar to the component relevance analysis, the deficiency ranking is designed to be extensible with new ranking metrics in the future. This is especially relevant as most of the ranking metrics presented in this thesis are deficiency-specific. If Archimetric were to be applied in a scenario in which different deficiencies would be detected, new ranking metrics would be needed.

8.2. Assumptions

The assumptions on which the ranking metrics are based, are similar to those of the component relevance metrics presented in Chapter 6.

- The severity-related aspects have to be mapped to a numerical value between 0 and 1. Similar to the component relevance metrics, all aspects

that can be mapped to an ordinal scale can be considered as a basis for a relevance metric.

- The metric values are assumed to be proportional to the severity of the deficiency occurrence, i.e. the higher the metric value, the more severe the corresponding aspects of the deficiency occurrence.
- For the calculation of the overall deficiency rank, all ranking metrics are assumed to be of equal importance. In the future, the deficiency ranking could be extended with the possibility to assign weights to the different ranking metrics in order to increase the configurability of the approach.

8.3. Ranking Metrics

For the ranking of deficiency occurrences in Archimetric, I use general and deficiency-specific metrics. General ranking metrics can be applied to all design deficiencies.

In many cases, different occurrences of the same deficiency are not all equally critical. For example, an *Interface Violation* between classes which reside in the same package may be more acceptable than an *Interface Violation* between completely unrelated classes. Therefore, deficiency-specific ranking metrics can be used to measure such factors and differentiate occurrences of the same deficiency better. Deficiency-specific ranking metrics can only be applied to certain deficiencies.

In the following, the general and deficiency-specific ranking metrics used in this thesis are explained. There is one general rankings metric and three deficiency-specific ranking metrics. For the deficiency-specific metrics, the deficiencies to which they are applicable are listed. As this thesis focuses on the structural analysis for the detection of deficiencies, metrics for the ranking of behavioural detection results are not discussed here. If a behavioural analysis would be used, additional general ranking metrics which take, e.g. the number of matching traces into account, could be used.

8.3.1. Structural Accuracy Metric

The structural accuracy metric is a general deficiency ranking metric and is therefore applicable to all deficiencies.

Structural formalisations can contain mandatory and additional elements. Mandatory elements have to be matched in order for a deficiency occurrence to be detected. Additional elements can be matched during the detection process but are not necessary for the detection of a deficiency occurrence.

The exemplary formalisation of the *Transfer Object Ignorance* occurrence in Figure 4.3 contains one additional element: the Method `someFunctionality` of the `exposedClass`. It represents the fact that a *Transfer Object Ignorance* occurrence is more critical if the `exposedClass` contains functionality that is made available to the `callingClass`. If the `exposedClass` contains no methods, the occurrence may either be a false positive or it may be less critical.

The structural accuracy of a pattern occurrence o is calculated as the ratio between the elements of a deficiency formalisation that are actually matched for a given occurrence and all elements in the formalisation [Tra07] (see Equation 8.1).

$$Rank_{StructuralAccuracy}(o) = \frac{\#matchedElements}{\#mandatoryElements + \#additionalElements} \quad (8.1)$$

This value hints at the confidence that can be placed in a detected deficiency occurrence. If only the absolutely mandatory core of a formalisation was matched for a pattern occurrence but none of the additional elements were detected, then the occurrence might arguably be a false positive. If an occurrence, on the other hand, conforms to the structural formalisation in all the details, it is probably a true positive. Therefore, I suggest to use this structural accuracy of a detected deficiency occurrence as a metric in the deficiency's severity ranking. While it is no definitive indicator of the deficiency occurrence being a true or false positive, it can point to the occurrences that are more likely to be true positives.

In the example occurrence no. 1 from Figure 3.3, the occurrence consists of six matched objects (the callingClass Assets, the call in the method calculateValue, the calledClass Reporting, the calledMethod sendReport, the param r , and the exposedClass Report), two matched component sub patterns ($c1 = \text{Accounting}$ and $c2 = \text{Controlling}$), and one matched object constraint (the name constraint of the exposedClass Report). The method someFunctionality cannot be matched for this occurrence because Report does not contain any methods. In this case the structural accuracy metric would be calculated as follows:

$$Rank_{StructuralAccuracy}(TOI_1) = \frac{6 + 2 + 1}{9 + 1} = 0.9$$

For example occurrence no. 2, the someFunctionality object would be matched to the method checkStock. Therefore, the occurrence's ranking value would be 1.

8.3.2. Deficiency-Specific Ranking Metrics

The metrics presented in this section can only be applied to specific deficiency occurrences. Therefore, the deficiencies to which the given ranking metric can be applied are indicated at the beginning of the following sections. In the calculation of the ranking metrics, several clustering metrics from the architecture reconstruction are reused. These are clustering metrics which have been identified as being especially susceptible to the influence of deficiency occurrences (see Section 5.3.2), e.g. the different Access Count metrics or the Package Mapping metric.

Class Locations Metric

Applicable to Interface Violation, Transfer Object Ignorance, Unauthorised Call

The idea behind the *Class Locations Metric* is that classes which reside in the same part of the system are intended to collaborate closely with each other. For Java systems, these are classes that lie in the same branch of the package tree or even belong to the same package. Consequently, an occurrence of the *Transfer Object Ignorance* deficiency (or of one of the other deficiencies listed above) between classes which are located far away from each other (in terms of the package structure) is a more serious design problem than an occurrence between classes in the same part of the package tree. For this metric, the value of the *PackageMapping* metric that is calculated during the architecture reconstruction is used (see Section 5.2.1).

The calculation of the class locations metric is shown in Formula 8.2.

$$Rank_{ClassLocations}(D) := 1 - PackageMapping(CC_D) \quad (8.2)$$

Here, D is the design deficiency occurrence and CC_D represents the component candidate that contains the classes that are involved in the design deficiency occurrence D . The higher the *PackageMapping* value, the lower the occurrence is ranked.

In the running example in Figure 3.3, the classes *Assets* and *Report* lie in the package `app.accounting` while the class *Reporting* belongs to the package `app.controlling`. Since the classes belong to different sub packages of the `app` package, they have a package mapping value of 0.5. Thus, the rank of the *Transfer Object Ignorance* deficiency no. 1 would be $1 - 0.5 = 0.5$.

Deficiency occurrence no. 2 in the example exists between the classes *Assets*, *StoreQuery*, and *Inventory*. As stated above, *Assets* belongs to the package `app.accounting`. In contrast, *StoreQuery* and *Inventory* are part of the `data.storage` package. Thus, the classes have a package mapping value of 0 and the rank of the deficiency occurrence is 1, i.e. it is more relevant than deficiency occurrence no. 1.

Communication via Data Classes Metric

Applicable to Transfer Object Ignorance

The *Transfer Object Ignorance* deficiency describes situations in which objects that are not transfer objects are passed between components. There are several degrees of severity depending on the class that is used for communication. For example, a class that only contains fields and access methods but does not adhere to the specific naming convention for transfer objects does not constitute a major problem. It may be that the developer intended to use a transfer object and only forgot to name the class correctly. If, on the other hand, a class adheres to the naming convention but contains several methods with application logic, the deficiency is more severe.

Hence, the heuristic I suggest for this metric is as follows: The more non-accessor methods occur in the class, the worse is the deficiency. Therefore, we

use the *Communication via Data Classes Metric* (see Formula 8.3).

$$Rank_{DC}(D) = 1 - IsDataClass(c) \quad (8.3)$$

$$IsDataClass(c) = \begin{cases} 0 & \text{if } \#Fields(c) = 0 \\ 1 - \left(\frac{\#NonAccessors(c) + \#MissingAccessors(c)}{\#NonAccessors(c) + \#PotentialAccessors(c)} \right) & \text{else} \end{cases} \quad (8.4)$$

The metric value of the Communication via Data Classes metric for *Transfer Object Ignorance* occurrences is 1 minus the similarity of the transfer object's class to a data class. Formula 8.4 shows how this similarity is calculated:

- If the class does not contain any fields, it is definitely not a data class. Therefore, the rank of such a deficiency occurrence with respect to the Communication via Data Classes metric is 1.
- If the class contains fields, the metric value depends on the methods in the class. On the one hand, the number of methods that are not accessors ($\#NonAccessors(c)$) is counted. The number of potential accessor methods in a class $\#PotentialAccessors(c)$ is 2 times the number of fields. By counting the number of actual accessor methods and subtracting the count from the number of potential accessor methods, we get the number of missing accessor methods $\#MissingAccessors(c)$. The method counts are related to one another as shown in the formula. The more non-accessor methods exist in the class and the more accessor methods are missing, the more the class deviates from a data class. This leads to a higher ranking with respect to this metric.

In the running example from Section 3.3, the class *Inventory* is exposed to the Accounting component although it is not a data transfer object. *Inventory* has one non-accessor method: *checkStock*. Assuming that the *Inventory* class possesses two access methods for the field *items* (e.g. *getItems* and *setItems*), no accessor methods are missing. The rank calculation for the *Transfer Object Ignorance* occurrence between *Assets*, *StoreQuery*, and *Inventory* would be as follows:

$$IsDataClass(Inventory) = 1 - \left(\frac{1 + 0}{1 + 2} \right) = \frac{2}{3}$$

$$Rank_{DC}(TOI_1) = 1 - \frac{2}{3} = \frac{1}{3}$$

In contrast, *Report* does not contain non-accessor methods. Therefore, the *Transfer Object Ignorance* occurrence between *Assets*, *Reporting*, and *Report* would be ranked like this:

$$IsDataClass(Report) = 1 - \left(\frac{0 + 0}{0 + 2} \right) = 1$$

$$Rank_{DC}(TOI_2) = 1 - 1 = 0$$

Hence, *Transfer Object Ignorance* occurrence no. 1 is more critical than occurrence no. 2 with respect to the Communication via Data Classes metric.

Number of External Accesses Metric

Applicable to Interface Violation, Transfer Object Ignorance, Unauthorised Call

During the architecture reconstruction, high coupling is an indicator of component merging and composition (see Chapter 5). As explained in Section 5.2.1, the coupling metric is calculated by relating the number of internal accesses of a component to the number of its external accesses. The occurrence of deficiencies like *Transfer Object Ignorance* or *Interface Violation* changes these metrics. In the example presented in Section 5.3, the metrics *Internal Accesses Count* and *External Accesses Count* are increased by 1 each.

If a component has many external accesses, an increase by 1 does not make a big difference. In a component with few external accesses however, a slight increase could greatly affect the coupling. Consider the following example: In Figure 5.5, component C_A has one internal access to C_B (from *Assets* to *StoreQuery*). Assume that C_A had three external accesses in total. With the deficiency occurrence depicted on the right in Figure 5.5, internal and external accesses would both increase by 1.

The coupling between C_A and C_B would thus be calculated as follows.

$$Coupling(C_A, C_B)_{withoutTOI} = \frac{1}{3} \approx 0.33$$

$$Coupling(C_A, C_B)_{withTOI} = \frac{2}{4} = 0.5$$

However, if we assumed that C_A had ten external accesses, the difference between the two coupling values would be much lower:

$$Coupling(C_A, C_B)_{withoutTOI} = \frac{1}{10} = 0.1$$

$$Coupling(C_A, C_B)_{withTOI} = \frac{2}{11} \approx 0.18$$

Hence, the number of external accesses is indicative of the impact that a deficiency can have on the coupling and thereby the merging and composition of components. If a component has a lot of external accesses, the impact is significantly lower than for components with few external accesses. Therefore,

deficiency occurrences in components with a high external accesses count are less severe than those in components with a low external accesses count. The deficiency rank with respect to this metric is calculated according to Formula 8.5.

$$Rank_{No.OfExt.Accesses}(D) := 1 - \frac{ExternalAccessesCount(C_D)}{\max_{C_i \in allComponents} (ExternalAccessesCount(C_i))} \quad (8.5)$$

Here, C_D is the component containing the deficiency occurrence.

8.4. Rank Calculation

For the calculation of the overall rank of a deficiency occurrence, all applicable general and deficiency-specific ranking metrics are taken into account. As stated in Section 8.2, all ranking metrics are considered to be equally important.

The overall ranking result for the design deficiency occurrences is determined by calculating the Pareto optimal design deficiency occurrences with respect to the applicable ranking metrics. This is similar to the calculation of the component relevance defined in Formula 6.9 with the applicable ranking metrics in place of the relevance strategies. Thus, the design deficiency ranking is equally easy to extend. The normalisation of the rank value even allows for the comparison of deficiency occurrences for which different ranking metrics are applicable.

Assuming that both deficiency occurrences from the running example are detected with a structural accuracy value of 100%, and that the Number of External Accesses metric evaluates to 0.5 for both occurrences, their severity is ranked as follows.

$$Rank_{overall}(TOI_1) = \frac{\sqrt{1^2 + 0.5^2 + 0^2 + 0.5^2}}{\sqrt{4}} = \frac{\sqrt{1.5}}{2} = 0.75$$

$$Rank_{overall}(TOI_2) = \frac{\sqrt{1^2 + 1^2 + \frac{1}{3}^2 + 0.5^2}}{\sqrt{4}} \approx \frac{\sqrt{2.611}}{2} \approx 0.808$$

The second deficiency occurrence is ranked as more severe with respect to the employed metrics. Thus it would probably be better to reengineer this deficiency occurrence first.

8.5. Limitations

This section lists the limitations of the proposed deficiency occurrence ranking method.

- One drawback of the proposed method is that only objectively quantifiable aspects of the deficiency occurrences can be considered in the ranking. In the running example, the second deficiency occurrence would be ranked

the more severe one. While this is correct with respect to the employed metrics, several other aspects can play a role in an architect's decision which deficiency to remove. For example, the first deficiency occurrence in the example can be easily removed by renaming the Report class (see Section 3.3). An architect might want to prioritise these simple reengineerings over more complicated ones. On a related note, an architect may want to remove those deficiencies first that he is familiar with (or with the part of the system that the deficiency occurs in). This cannot easily be covered by metrics because it is a very subjective, architect-dependent heuristic.

8.6. Related Approaches

This section presents a number of approaches which either identify the need to rank detected pattern occurrences or which propose methods to do so.

Simon et al. were among the first to argue that tool support is necessary to point out where refactorings can be applied [SSL01]. They suggest to use metrics and an appropriate visualisation to help developers find worthwhile refactoring locations. However, they admit that their approach does not scale for large applications. The execution of the refactoring and the impact on the system are not in the focus of their work.

Marinescu adds 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. He does not address the removal of detected bad smells.

Bourquin and Keller present an approach that is focused on manual refactorings on the architecture level [BK07]. They argue that bad smells “cannot be quantified easily, and therefore are hard to prioritize”. They manually analyse the relevance of their refactorings on the architecture after their application. In order to analyse the refactoring results, they use code metrics and a comparison between the number of detected bad smells before and after the refactoring.

Liu et al. point out that the removal of one bad smell occurrence can facilitate or complicate the removal of other occurrences [LYN⁺09]. Hence, they suggest that a proper resolution order should be found before the refactoring is begun. This does not necessarily mean that the ‘most critical’ bad smells are removed first. However, finding the best resolution order does either require static rules (“Smells of type A should always be removed before smells of type B.”) or human judgement by a software architect.

Niere et al. suggest to assign a “fuzzy belief” to the formalisation of design patterns which expresses the quality of the formalisation [NSW⁺02, Nie04]. An occurrence of a pattern with a low belief would be ranked lower than occurrences with a higher belief. This means that occurrences of the same pattern are always assigned the same rank. This system was incorporated in early versions of Reclipse. It was later refined, by Travkin, Wendehals, and Meyer

[Tra07, Wen07, Mey09]. They allow the specification of “mandatory” and “additional” elements in pattern formalisations. If an occurrence contains some or all additional elements, it is ranked higher than an occurrence which contains only the mandatory elements. This method is currently implemented in Reclipse. In this thesis, I reuse this technique in the Structural Accuracy Metric presented in Section 8.3.1.

8.7. Conclusion

In this chapter, I presented an extensible approach to the ranking of detected deficiency occurrences. I also described four example metrics and illustrated their evaluation for the running example. Finally, the calculation of the overall deficiency rank for the two example occurrences from the running examples was shown.

In future applications of Archimetrix, the presented metrics can be reused if the same deficiencies as in this thesis are detected. If different deficiencies were formalised and searched for, new ranking metrics for them would have to be devised. This is however supported by the extensibility of the deficiency ranking.

9. Deficiency Removal

The previous chapters presented the reverse engineering capabilities of Archimetric. They consist on the one hand of the architecture reconstruction and on the other hand of the detection of design deficiencies. However, Rosik et al. point out that “inconsistency identification is not of itself sufficient to ensure architectural consistency” [RLGB⁺11]. They note that often developers do not remove deficiencies even if they know about them. Therefore, Archimetric offers support for the removal of deficiencies.

This chapter begins with an overview of the contributions of Archimetric in the area of deficiency removal. Afterwards, the assumptions of the deficiency removal are discussed and the deficiency removal process is explained (Section 9.3). Archimetric provides support for the manual removal of deficiencies as well as for the automated removal. The manual removal is the subject of Section 9.4 while Section 9.5 deals with the automated removal. The chapter ends with a discussion of the limitations and the conclusion.

9.1. Contributions

- This chapter presents a dedicated process to support the software architect in removing the detected design deficiency occurrences. This process refines Step 5 of the Archimetric process presented in Chapter 4.
- This chapter also introduces the concept of guide templates which can be used to generate concrete removal guides for the manual removal of individual deficiency occurrences.
- For the automated removal of deficiency occurrences, this chapter discusses how automatic removal strategies can be specified. In addition, a method to preview the effects of the application of automatic removal strategies is introduced.

9.2. Assumptions

- A basic assumption of this chapter is that the detection and ranking of deficiency occurrences have been executed successfully before the deficiency removal. The explanations in this chapter assume that the software architect has chosen a particular design deficiency occurrence which he now wants to remove.
- I assume that the deficiency occurrences are unrelated, i.e. that they do not share common elements. Therefore, I do not consider cases in which

the removal of one deficiency occurrence invalidates other occurrences. This issue is discussed in more detail in Section 9.6

9.3. Deficiency Removal Process

Detecting deficiency occurrences in the system goes a long way towards supporting the software architect. As described in Chapter 3, design deficiency descriptions often also point out one or more ways to remove a given occurrence of that deficiency. In order to really accomplish the removal, however, the architect still has to perform two steps.

- 1. Role mapping** In the deficiency description, the removal is described in terms of the general form of a deficiency. For example, in Section 3.3, the removal strategy *Move calling method* talks about 'moving the called-Method' to the callingClass. These are, of course, the pattern roles that are played by concrete elements in a given deficiency occurrence. The architect has to map these roles to the concrete elements in order to understand, that, e.g. for the removal of *Transfer Object Ignorance* occurrence no. 2, the method `calculateValue` should be moved to the `StoreQuery` class.
- 2. Removal strategy application** Once the architect has identified what has to be done with which element, he still has to execute the prescribed removal. Depending on the complexity of the deficiency and the removal strategy, this task can be anything from straightforward to very complicated. Still, manually removing a design deficiency occurrence is – like any programming activity – error-prone. Thus, it is helpful to automate this task wherever this is possible.

In order to support the software architect in the removal of a deficiency occurrence, I propose the process that is illustrated in Figure 9.1. The process refines Step 5 “Architecture Preview & Deficiency Removal” from the Archimetrix process presented in Chapter 4. Therefore, it begins after the deficiency ranking.

Archimetrix allows for the specification of two kinds of supportive artefacts: *Guide templates* and *automated removal strategies*.

Guide templates On the one hand, there are guide templates which textually enumerate the steps that are necessary to remove a given deficiency occurrence (Step 1.1 in Figure 9.1). They are comparable with the “refactoring mechanics” used by Fowler [Fow99, p. 111] and Kerievsky [Ker04, p. 48]. The template aspect of these guide templates is such that Archimetrix can replace the pattern role names used in the guide templates with the corresponding object names from a given deficiency occurrence. This instance of a guide template for a specific deficiency occurrence is called a *removal guide* for this occurrence. By generating removal guides for all the deficiency occurrences he wants to remove, the software architect does not have to perform the role mapping by himself. He can concentrate on the removal of the deficiency occurrences without constantly keeping in mind which element of the occurrence is playing which role.

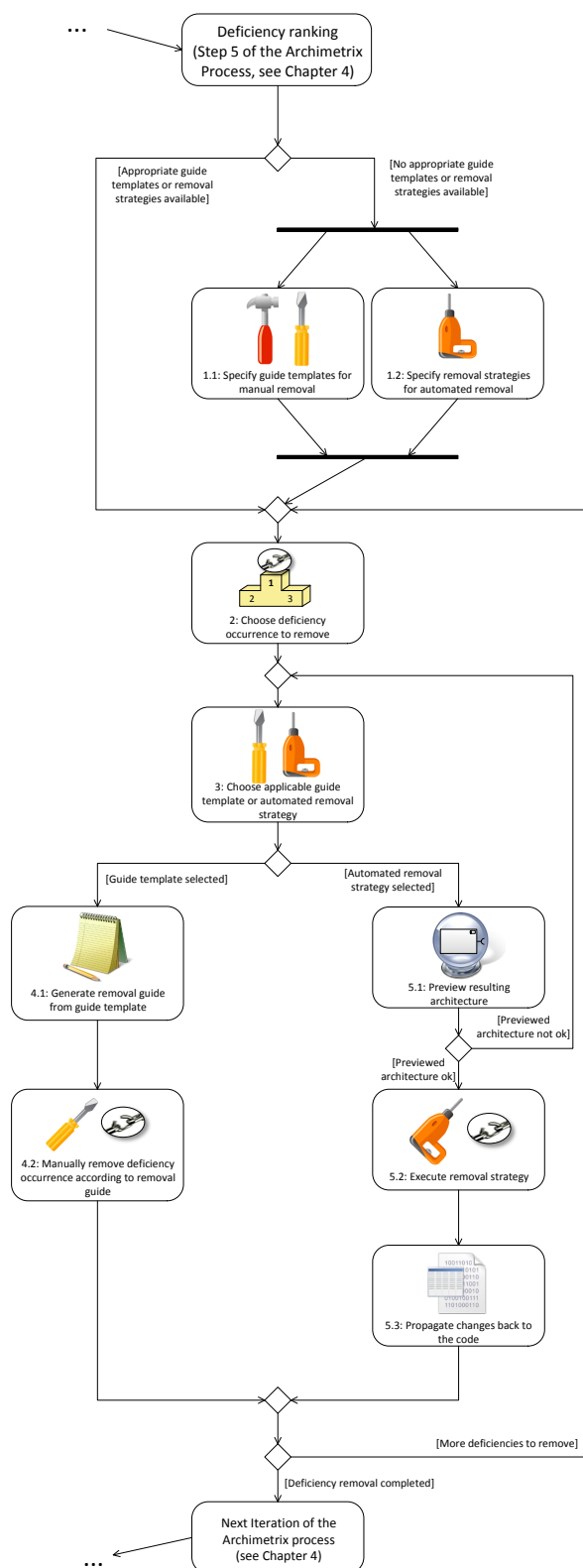


Figure 9.1.: Process for the removal of deficiency occurrences

Automated removal strategies Archimetrix also allows for the specification of automated removal strategies (Step 1.2). These describe transformations of the software system which automatically remove a given deficiency occurrence. This completely relieves the architect of the role mapping and of the manual removal of deficiency occurrences.

Guide templates and removal strategies can of course be saved and reused. Thus, Steps 1.1 and 1.2 of the process are only necessary when no such artefacts exist or when additional ones are needed.

When the necessary guide templates and removal strategies are available, the architect can select a deficiency occurrence which shall be removed (Step 2). This choice will be based on the deficiency occurrences detected by Reclipse (see Chapter 7). It can be supported by the deficiency ranking described in Chapter 8.

The guide templates and removal strategies are only applicable to specific deficiencies. Therefore, Archimetrix filters the available templates and strategies based on the deficiency occurrence selected by the software architect. From these, the architect has to choose one solution (Step 3). Depending on this choice, the process continues in two possible ways.

If the architect chooses a guide template for the removal of the deficiency, a removal guide for the selected deficiency occurrence is generated (Step 4.1). The removal guide resembles a cooking recipe that contains the specific role names of the selected deficiency occurrence in place of the more general names from the deficiency description. The architect can then remove the deficiency occurrence manually according to the removal guide (Step 4.2). Section 9.4 describes the specification of guide templates and the generation of removal guides.

If, on the other hand, the software architect chooses an automated removal strategy, he can first preview the architecture that will result from the execution of the strategy (Step 5.1). He can then either go back to Step 3 and choose another solution or he can execute the removal strategy (Step 5.2). Because the automated removal strategies are executed on the GAST of the system, the changes that are made by the removal strategy then have to be propagated back to the source code (Step 5.3). The specification and execution of automated removal strategies is described in detail in Section 9.5.

After the removal of one or more deficiency occurrences, the main process can continue. The next step in the Archimetrix process is a new architecture reconstruction which will then be based on the reengineered system. If the deficiency occurrences were removed manually, the system has to be parsed again so that a new GAST is created. Otherwise, the reengineered GAST can be directly used as an input for the new architecture reconstruction.

9.4. Manual Deficiency Removal

As explained in Section 9.3, not all deficiency occurrences are removable automatically. Sometimes, not all information necessary for a deficiency removal is available (e.g. how to name a new class or where to move a method). At other

times, a deficiency occurrence is so complicated that it cannot be removed by pre-specified removal strategies because not all possible cases can be covered by the removal strategy with reasonable specification effort. In these cases, the software architect has to remove the deficiency occurrence manually.

Carrying out refactorings and reengineerings manually can be supported by providing the architect with a guide to the different refactoring steps. Fowler [Fow99] and Kerievsky [Ker04] call these step-by-step guides “mechanics” of the refactoring. However, these mechanics use the terminology from the general form of a pattern. Therefore, such a “mechanic” for the removal of a *Transfer Object Ignorance* occurrence might talk about an ‘exposed class’ or a ‘called method’. The architect still has to perform the role mapping on-the-fly in order to relate this description to the occurrence at hand (see Section 9.3). Thus, he has to keep in mind that the ‘exposed class’ is called, for example, Report and that the ‘called Method’ is the method `sendReport` in the class `Reporting`. When the involved elements bear similar names as in the example, or when a complicated pattern consists of many elements in different roles, the role mapping can become quite tedious and error-prone.

9.4.1. Removal Guides

Archimetrix can support the software architect by generating a removal guide for a given deficiency occurrence from the provided guide templates. The guide template contains the necessary (manual) steps for the removal of a given deficiency occurrence but it uses the generic pattern role names of the elements from the general form of the deficiency. The general succession of removal steps is the same for every deficiency occurrence but the concrete objects names differ. The generated removal guides use the concrete object names from a given deficiency occurrence instead of the generic pattern role names. This saves the architect the tedious task of the role mapping and facilitates the deficiency removal for him.

On a technical level, the guide generation can be achieved by a simple template mechanism. The basic text is always the same. It is captured in a guide template. The template engine substitutes the role names with the concrete element names from the deficiency occurrence.

9.4.2. Example

In Section 3.3, the removal strategy *Introduce transfer object* was presented for the *Transfer Object Ignorance* deficiency. Because this removal strategy requires a complex data analysis, it is hard to automate. A removal guide template could look as follows.

1. Analyse the methods `#calledMethod` and `#callingMethod` and find out which data from the class `#exposedClass` is used therein.
2. Create a new class named `#exposedClassTO`.
3. In method `#calledMethod`, create an new instance of `#exposedClassTO`.

4. Then populate the newly created instance of `#exposedClassTO` with the data used in `#calledMethod` and `#callingMethod`.
5. Change the parameter list of `#calledMethod`, so that it accepts instances of `#exposedClassTO` instead of `#exposedClass`.
6. Adapt the corresponding interface, if necessary.
7. Adapt all other classes that implement that interface, if necessary.
8. In `#callingMethod`, pass the instance of `#exposedClassTO` to `#calledMethod` instead of the instance of `#exposedClass`.

Listing 9.1: Removal guide template for the *Transfer Object Ignorance* deficiency

Suppose, the architect chose to remove *Transfer Object Ignorance* occurrence no. 1 from the example in Figure 3.3 and he selected to remove it by applying the *Introduce transfer object* removal strategy. Archimetric would instantiate the template shown in Listing 9.1 and produce the removal guide shown in Listing 9.2

1. Analyse the methods `sendReport` and `calculateValue` and find out which data from the class `Report` is used therein.
2. Create a new class named `ReportTO`.
3. In method `calculateValue`, create an new instance of `ReportTO`.
4. Then populate the newly created instance of `ReportTO` with the data used in `sendReport` and `calculateValue`.
5. Change the parameter list of `sendReport`, so that it accepts instances of `ReportTO` instead of `Report`.
6. Adapt the corresponding interface, if necessary.
7. Adapt all other classes that implement that interface, if necessary.
8. In `calculateValue`, pass the instance of `ReportTO` to `sendReport` instead of the instance of `Report`.

Listing 9.2: Removal guide for *Transfer Object Ignorance* occurrence no. 1 from Figure 3.3

9.5. Automated Deficiency Removal

The automated removal of deficiency occurrences can be achieved by means of automatic transformation of the software. This transformation can either take place directly in the source code or in the software's GAST representation. In the latter case, the source code has to be evolved synchronously with the GAST in order to correctly reflect the removal of the deficiency occurrence. However, it also allows the software architect to explore different reengineering alternatives without touching the source code.

As the architecture reconstruction as well as the deficiency detection in Archimetric both take place on the level of the GAST, I decided to also execute the automated deficiency removal directly in the GAST. To this end, story diagrams [vDHP⁺12], are used to describe the model transformations necessary

to accomplish the removal. Story diagrams are an in-place model transformation language developed at the University of Paderborn. Since, on a conceptual level, Archimatrix is not dependent on a specific model transformation language, other in-place model transformation languages like Henshin [ABJ⁺10], MOLA [KBC05], QVT [Obj11], or VIATRA2 [VB07] could be integrated, as well.

Similar to the deficiency formalisations, the removal strategies can be specified by the software architect or by a deficiency expert. Therefore they can also be adapted to consider project- or company-specific guidelines like naming conventions.

9.5.1. Removal Strategies

This section shows two exemplary specifications of the removal strategies *Mark exposed class as transfer object* and *Move called method* introduced in Section 3.3. Both removal strategies are possibilities to remove occurrences of the *Transfer Object Ignorance* deficiency. A formalisation of a more complex removal strategy for the design deficiency *Interface Violation* can be found in Appendix B.1.

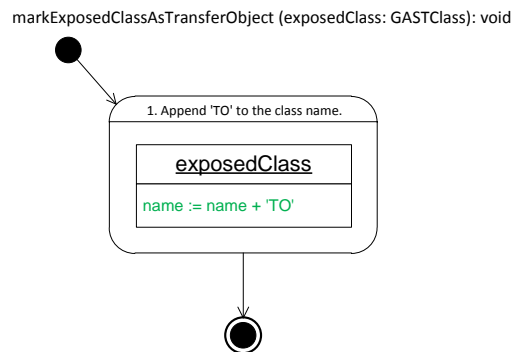
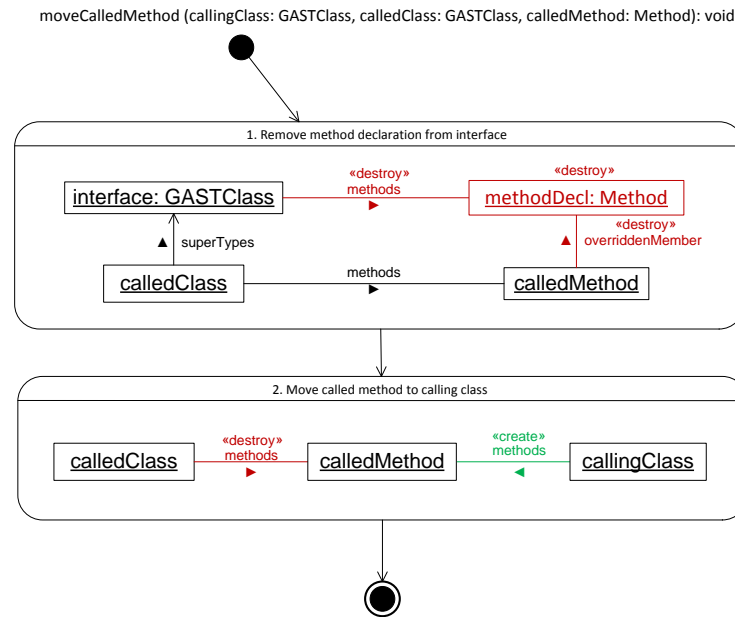


Figure 9.2.: Removal strategy: *Mark exposed class as transfer object*

Mark exposed class as transfer object Figure 9.2 shows an extremely simple transformation which realises the *Mark exposed class as transfer object* removal strategy. It only consists of one story node which contains only the object variable `exposedClass`. The `exposedClass` is passed to the removal strategy as a parameter as shown in the signature at the top of Figure 9.2. The only modification that the removal strategy applies is the addition of the suffix 'TO' to the name of the exposed class in order to mark it correctly as a transfer object.

The removal strategy assumes that the exposed class is really intended to be a transfer object and that it is just named incorrectly. This is an example of a project-specific removal strategy that enforces a specific naming convention.

Figure 9.3.: Removal strategy: *Move called method*

Move called method Figure 9.3 shows the slightly more complex specification of the *Move called method* removal strategy. As described in Section 3.3, it moves the called method from the called class to the calling class. This makes the use of transfer object unnecessary and therefore removes the deficiency.

The removal strategy has three parameters, `callingClass`, `calledClass`, and `calledMethod`, and consists of two story nodes. According to the deficiency description, the `calledMethod` is available via an interface of the the `calledClass`. The first story node specifies that the declaration of the `calledMethod` is to be removed from that interface. The second story node removes the `calledMethod` from the `calledClass` and puts it into the `callingClass` by destroying and creating the corresponding containment links.

Obviously, this specification of the removal strategy is rather simplistic. While it takes care of removing the method declaration from the interface, it does not consider other classes that may implement the same interface. Similarly, methods other than the `callingMethod` that may have called the moved method are ignored. All these things can be modelled with story diagrams but are omitted here for the sake of brevity.

9.5.2. Behaviour Preservation

A software architect has many degrees of freedom when removing a deficiency occurrence manually. He may even decide to remove method calls and to move or reimplement functionality. However, an automated transformation should not change the behaviour of the system under analysis. There are cases in which an adaptation of behaviour may be necessary. For example, in the CoCoME

system which served as one of the case studies in the validation of Archimetric (see Chapter 10), one deficiency occurred because a persistence mechanism was called from a component which was forbidden to access it. In such a case, the mechanism could be made “legally” available to the accessing component, it could be extracted to a separate component, or it could be reimplemented in the accessing component. In a strict sense, the last solution would not preserve the behaviour because different methods would be called after the deficiency removal (although they act similar to the originally called methods).

However, in most cases behaviour preservation is a desired property of a deficiency removal, especially if the removal is performed automatically. At the moment, Archimetric does not offer support to analyse if an automatic removal strategy preserves the system behaviour. To enforce this property, a suitable analysis approach, e.g. the one by Meyer [Mey09], could be integrated into the deficiency removal process. Meyer uses inductive invariant verification to prove that automatic transformations of a system do preserve certain properties (e.g. that variable accesses are not removed) and to prevent the introduction of illegal constructs (e.g. accesses to members which are not visible). This way, it could be ensured that an automatic deficiency removal does not alter the system behaviour.

9.5.3. Propagating the Removal back to the Source Code

Carrying out the deficiency removal on the level of the GAST has advantages and disadvantages. On the one hand, the transformed GAST can directly be used for a new clustering without the need to parse the source code again. In large systems, this can save valuable time. It also allows the software architect to explore different alternatives for the removal of deficiencies without changing the source code. Only when a suitable removal strategy has been found, he could choose to propagate the removal to the code. In addition, it is advantageous to specify the deficiency’s structural formalisations and the deficiency removal in languages which are conceptually and syntactically similar. The software architect can use similar syntactic constructs in the specifications and does not have to learn two completely different languages. It also facilitates the understanding of the specifications for other stakeholders.

On the other hand, if changing the source code is desired, the transformation of the GAST incurs the problem that the source code has to be kept in sync. Archimetric handles this problem by providing a code generation for the GAST. In the prototype implementation of Archimetric, this code generation is realised in a proof-of-concept manner. However, to fully support the synchronisation of GAST and source code, an incremental code generator would be needed (see Section 9.6).

9.5.4. Architecture Preview

Usually, the software architect has at most a vague idea about the implications of a given refactoring or reengineering. For example, the design patterns by Gamma et al. [GHJV95] and the from POSA books [BMR⁺96, SSRB00, KJ04]

all feature a section on the consequences of applying that pattern. This section consists of a prose description of what the pattern’s application means for the rest of the system. Many of Fowler’s refactorings end with the line “Compile and test.” [Fow99]. This not only suggests that the correct application of the refactoring has to be ensured, but that it is necessary to validate the program behaviour.

In Chapter 5, I showed that the introduction of design deficiency occurrences can influence the reconstruction of a software architecture. Similarly, the architecture can change when such deficiency occurrences are removed. In fact, those components that are likely to change on the removal of a deficiency occurrence are considered to be relevant for the deficiency detection (see Section 6). But on the one hand, this is not the only factor that determines the relevance of a component. And on the other hand, there is neither a guarantee that removing a deficiency occurrence will lead to a change of the component in question nor is it clear how exactly the component will change. So the effects of a deficiency removal are as unclear as the consequences of a pattern application or a refactoring. This can especially be a problem when multiple deficiency occurrences with similar or identical ranking values are detected.

Therefore, Archimetrix provides the software architect with a means to preview the architectural consequences of automated removal strategies. This allows the architect to explore the effect of the different removal strategies and to determine which strategy is most suited to remove a given design deficiency occurrence. For that purpose, the architecture that will result from the automatic removal of a given deficiency occurrence is calculated and presented to the architect. This preview helps the architect in judging how the selected removal strategy affects the architecture and lets him decide if this is in line with his requirements. Note that the architecture preview is only possible for pre-defined removal strategies. If the deficiencies are removed manually, no preview can be presented.

The architecture resulting from the initial clustering (referred to as *original architecture*, here) is compared to the anticipated architecture from the preview (referred to as *previewed architecture*, here). To execute the architecture preview, the architect selects the design deficiency occurrence to be removed and one of the applicable removal strategies. The selected removal strategy is applied to a copy of the original architecture by executing an in-place model transformation. The previewed architecture is then calculated by executing a new clustering on the modified copy of the model. As the clustering scales well for large systems (Krogmann reports a clustering time of 14 seconds for a system with 50,000 LOC [Kro10]), the execution of a complete clustering for producing the previewed architecture is justifiable.

To simplify the comparison for the user, the differences between the original architecture and the previewed architecture are visualised. Figure 9.4 depicts a possible visualisation of an architecture preview for the removal of *Transfer Object Ignorance* occurrence 1 from the running example. The components of the original architecture are visualised on the left side and the components of the previewed architecture are visualised on the right side. In this example, the original architecture only consists of one component, named *comp 1*. It

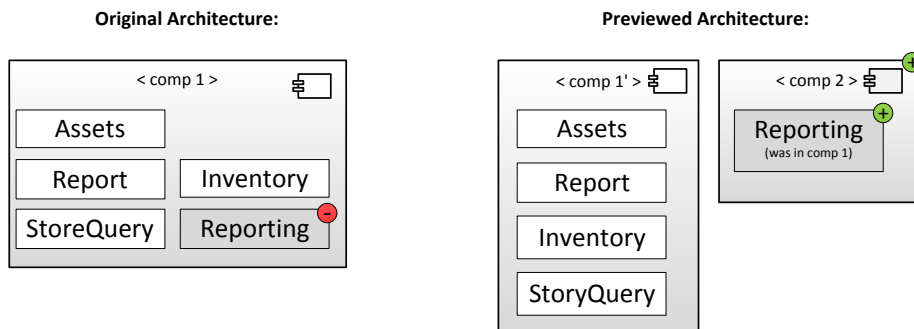


Figure 9.4.: Architecture preview for the removal of *Transfer Object Ignorance* occurrence no. 1

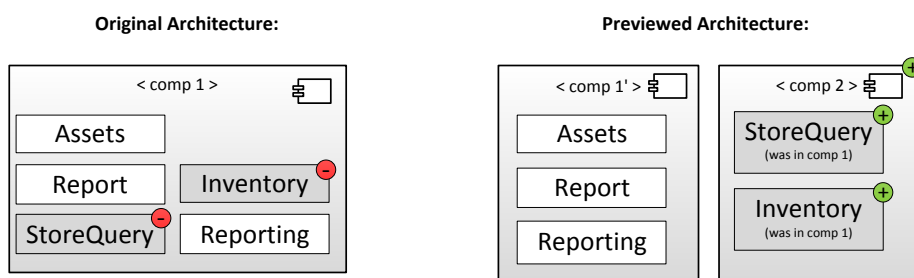


Figure 9.5.: Architecture preview for the removal of *Transfer Object Ignorance* occurrence no. 2

contains all five classes from the running example. Assuming that the removal of *Transfer Object Ignorance* occurrence 1 indeed has an effect on the architecture, the previewed architecture will consist of two components, `comp 1'` and `comp 2`. Classes that have been assigned to other components than in the original architecture (in this case `Reporting`) are visualised with a darker background in this figure. In the original architecture, classes that will no longer be assigned to a given component after the deficiency removal are marked with a red minus sign in the corner of the classes. Elements that are new in the previewed architecture in comparison to the original architecture are marked with a green plus. In this case, this applies to the component `comp 2` and the class `Reporting` contained therein. For moved classes, in the previewed architecture, the label “was in comp 1” indicates their former location.

Figure 9.5 visualises the software architecture that will result from the removal of *Transfer Object Ignorance* occurrence 2 instead of occurrence 1. If that deficiency occurrence was removed, a new component `comp 2` would be created, too. But instead of the class `Reporting`, it would contain the classes `StoreQuery` and `Inventory`.

Based on these two previews of the running example, a software architect could decide which of the two previewed architectures would fit his purpose best. Therefore, instead of having reengineering decisions based on a hunch, Archimatrix allows the architect to make a better, informed decision.

As explained in Section 9.3, not all deficiencies are automatically removable. Sometimes, not all information necessary for a deficiency removal is available (e.g. how to name a new class or where to move a method). At other times, a deficiency is so complicated that it cannot be removed by pre-specified removal strategies. In these cases, the architect has to remove the deficiency manually.

9.5.5. Related Approaches

Related work on the removal of bad smells has already been described in Section 2.5. In this section, I present a number of approaches which allow to preview the effects of a code change on the software architecture.

The visualisation of recovered architectures in general has received a lot of attention. Either specialised views of an architecture are constructed (e.g. [SSL01, vDHK⁺04]) or reconstruction results are described in terms of an architecture description language (ADL) and can be displayed accordingly (e.g. with the X-ray tool [MK01]). Pacione provides a comparison of different visualisation approaches [Pac03, PRW03]. These approaches are concerned with the visualisation of the status quo of an architecture. The effect of changes on this status quo is not addressed.

Many IDEs, e.g. Eclipse [Ecl12] or IntelliJ IDEA [Int11], support refactorings and previews of the resulting changes in the source code. For example, all lines of code which are affected by the renaming of a variable can be shown. However, the impact on the architecture is not analysed.

Zhao et al. carry out a change impact analysis on software architecture level [ZYXX02]. Starting from a formal architecture specification, they use slicing and chopping to identify those parts of a software architecture that are related to

a given change. On the one hand, this implies that a formal architecture model is required for their approach which may not always be the case. On the other hand, they identify which parts of the system are related and therefore should be considered when a given element of the system is changed. In contrast, the architecture preview of Archimetric precisely shows how the architecture will change if a specific change is executed automatically.

Göde and Deissenboeck present an approach called “Delta Analysis” [GD12]. They state that the usually large number of problems reported by static analysis tools decreases the manageability of these problems and the motivation of software architects. Instead they suggest that the architect should always remove some problems from that file that he has to work on anyway, thereby gradually improving the system. To capture this improvement they compare the file before the modification (the “baseline”) with the file after the modification and show which problems have been removed, which remain, and which have been introduced. This idea is related to the architecture preview presented in this chapter in that two states of a system are compared to each other. However, in Archimetric two versions of the reconstructed system architecture are compared in contrast to two versions of a source file.

9.6. Limitations

The removal of deficiencies in the Archimetric process is subject to a number of limitations.

- At the moment, a completely new architecture reconstruction has to be executed for the architecture preview. A copy of the system’s GAST is transformed by the automatic removal strategy and the result is given to SoMoX. SoMoX has to start from scratch and calculate all metric values for the transformed system to finally reconstruct the changed architecture. This can then be compared to the previous architecture.

Although this approach is straightforward, it is also computationally inefficient. Because the exact impact of design deficiencies on the clustering can be calculated (see Chapter 5), the metric values for the transformed GAST do not necessarily have to be re-measured. Instead the value changes could be calculated and then be fed into SoMoX to allow for a faster architecture reconstruction. For systems like those that were used in the validation (< 10 KLOC), for which the architecture reconstruction takes only seconds, this improvement may be insignificant. Reengineering realistically-sized applications with over one million lines of code may however be sped up substantially.

- On a related note, the architecture reconstruction that is part of every iteration of the Archimetric process (see Chapter 4) could be improved. Currently, the complete architecture is reconstructed at the beginning of a new iteration. This is done although only a small part of the system may be affected by the previous deficiency removal. Because the affected parts of the system are known, the architecture reconstruction could be

adapted to work incrementally. This way, only the changed parts of the system would have to be reconstructed at the beginning of a new iteration. Again, this may not be necessary for small systems but it will ensure the scalability of the approach for larger systems.

- The current approach assumes that deficiency occurrences are unrelated and can be removed independently of each other. In reality, this is too strong an assumption. Deficiency occurrences are often related or dependent on each other. If one occurrence is removed, others may become invalid, for example, because a previously bypassed interface has been extended. Therefore, it could be worthwhile to determine a good resolution order for the detected deficiency occurrences as suggested by Liu et al. [LYN⁺09]. This could be taken into account when advising the software architect.
- In the limitations of the Archimetrix process discussed in Chapter 4, I suggest that architecture quality metrics [SKR08] could be used to determine when to end the process. Similarly, such metrics could already be considered during the architecture preview step. For example, the architect could be advised if and how much the removal of a given deficiency occurrence would increase the quality metrics. Similar to the architecture preview, this would only be possible for automated removal strategies, of course.
- For the automated removal of deficiency occurrences, the current approach synchronises the transformed GAST and the source code by generating code for the complete system. This is, of course, impractical. Transformations from the code to the GAST and back almost invariably lead to the loss of information at some point. For example, the generated code may be formatted slightly differently. These differences can lead to problems with versioning systems because the complete code seems to have changed although only a small part of the GAST was actually transformed. It would be better to only generate code for those system elements that really have changed.
- The manual and automated deficiency removal are executed on two different artefacts, i.e. the source code and GAST, respectively. For this reason, the architect can only remove deficiency occurrences manually *or* automatically without synchronising these artefacts. If an occurrence is removed manually, the code has to be parsed into a new GAST representation before an automated removal can be executed. In the other case, code from the automatically reengineered GAST has to be generated before a manual removal is executed. Otherwise the modifications from the first removal will be overwritten by the second one.

9.7. Conclusion

This chapter explained how Archimatrix supports the software architect in removing detected deficiency occurrences. On the one hand, the manual removal of deficiency occurrences is supported by the generation of individualised removal guides for every occurrence. If automatic removal strategies are used, on the other hand, a preview of how the removal affects the software architecture can be computed and presented to the software architect. This dedicated support can encourage architects to actually remove architecturally relevant deficiency occurrences instead of refraining from it [RLGB⁺11].

10. Validation

In this chapter, I present the validation of Archimetrix. In the context of this thesis, a prototype implementation of Archimetrix was created which is described in Section 10.1. This prototype was used to carry out a Level-I-Validation, i.e. to compare the assumptions and predictions described in this thesis with the measurements obtained by running the prototype [BR08]. The setup of this validation is described in Section 10.2 while the validation questions are presented in Section 10.3. Section 10.4 gives an overview of the case studies to which the prototype implementation was applied. Section 10.5 lists the threats to the validity of this validation. Sections 10.6 to 10.8 report on the results that were obtained during the validation and discuss them with respect to the validation questions. The time and effort that was needed for the application of Archimetrix in the case studies is documented in Section 10.9. Due to time and resource constraints, a level-II-validation, i.e. a validation of the applicability of Archimetrix in practice, could not be carried out within the scope of this thesis. However, Section 10.10 sketches an approach to such a validation. Section 10.11 concludes the chapter by discussing the lessons learned.

A major part of the validation results presented here was also published in [vDPB13].

10.1. Prototype Implementation

The concepts presented in this thesis have been implemented in a research prototype which also goes by the name of Archimetrix. The description of the prototype in this section is based on a tool demonstration paper about Archimetrix [vD12]. The prototype is a collection of plug-ins for Eclipse which is freely available for download on the Archimetrix web page [Arc12]. All the models that are created and processed in the Archimetrix process are based on the Eclipse Modeling Framework (EMF). This section briefly shows the software architecture of the prototype implementation of Archimetrix and then presents an example session of the tool.

10.1.1. Software Architecture

Archimetrix reuses existing tools for several of the process steps described in Chapter 4. Figure 10.1 gives an overview of the general software architecture of Archimetrix.

Parsing the source code into a GAST is accomplished by the parser Sissy [Sis11]. Sissy allows for the analysis of Java, C++, and Delphi code. For the architecture reconstruction, Archimetrix relies on SoMoX [CKK08, Kro10] which uses a combination of software metrics to heuristically create a model of

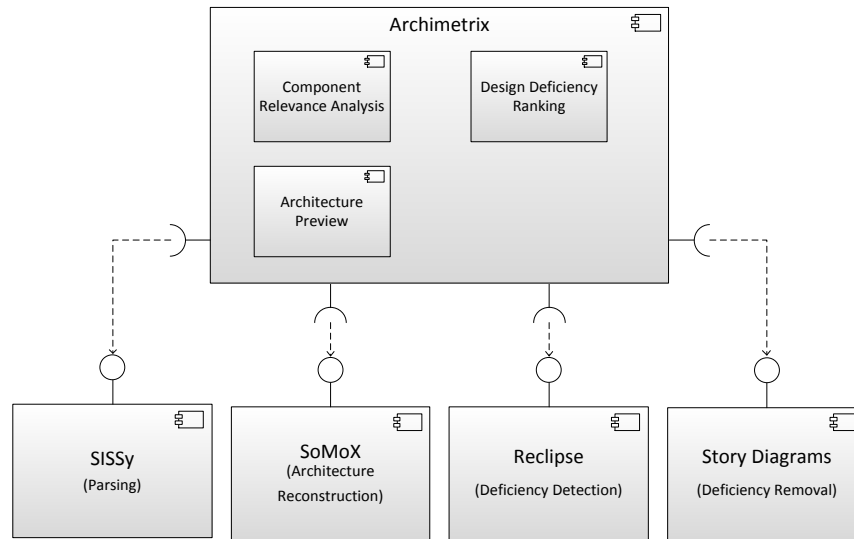


Figure 10.1.: Illustrative overview of the software architecture of Archimetrix

the software architecture of the system under analysis (see Section 5.2). The deficiency detection is accomplished by Reclipse [vDMT10a, vDMT10b, vDT10], a pattern detection tool which employs graph matching to identify patterns in a GAST (see Section 7.3). The automated removal of detected deficiency occurrences is realised by transformations of the GAST. For this step, Story Diagrams, a graphical in-place model transformation language [vDHP⁺12], are used in the prototype (see Section 9.5). The remaining steps of the process (component relevance analysis, deficiency ranking, and architecture preview) are implemented within Archimetrix itself. Archimetrix is also responsible for the coordination and execution of the process steps.

10.1.2. Example Session

This section gives an impression of the research prototype by describing an example session with Archimetrix. It uses the Store Example system introduced in Section 1.7. The Store Example system is also analysed in case study 1 in Section 10.6. The session follows the process steps presented in Chapter 4 and illustrates them with snapshots from the tool.

1. Parse the source code of the system under analysis First, the source code of the system under analysis has to be parsed. As described above, Archimetrix uses SISSy for this task. SISSy is able to parse Java, C++, and Delphi code and creates an instance of the GAST meta model (see Appendix A.1) from it.

2. Reconstruct the software architecture In this step, the GAST is the input for the initial architecture reconstruction step. The architecture reconstruction

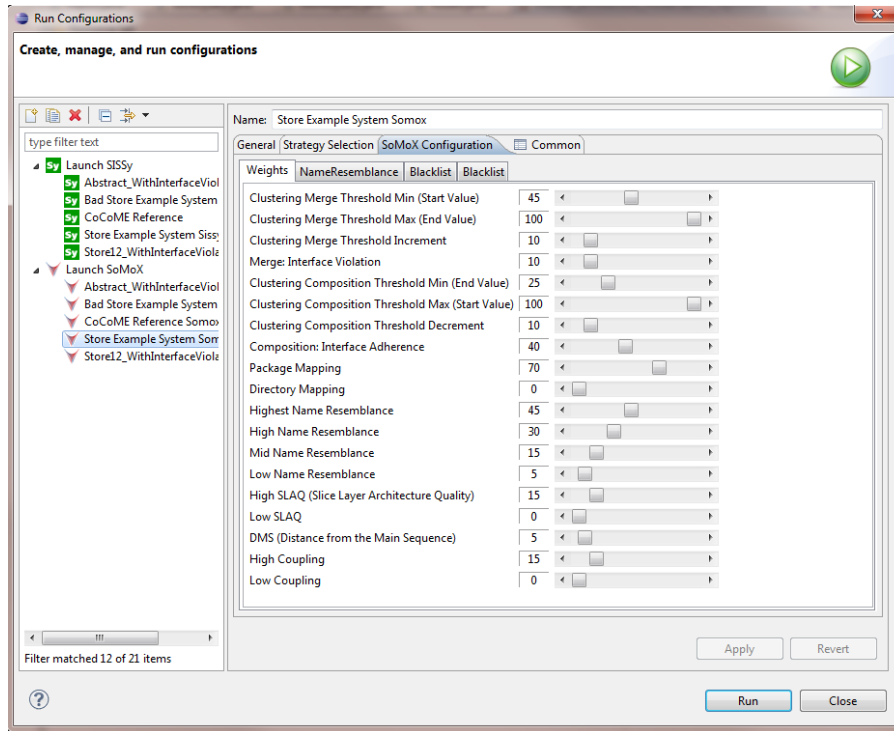


Figure 10.2.: Configuration of the metric weights for the architecture reconstruction

is accomplished by SoMoX. Before the reconstruction is started, SoMoX has to be configured.

Figure 10.2 shows the configuration of metric weights for the reconstruction. These weights allow, for example, to put a greater emphasis on the coupling or the name resemblance during the architecture reconstruction. The concrete values are dependent on the system under analysis and have to be determined by the software architect. SoMoX also provides a set of empirically determined default values.

As a result of the architecture reconstruction, several models are created. Two of these models are shown here ¹.

One model shows the reconstructed components, the interfaces with their operations and the detected data types (see Figure 10.3). For example, the diagram shows in the upper left that the reconstructed component <PC No. 18 ProductsListView> provides the IListview interface with its createListEntry operation. This interface is required by component <CC No. 3>. In addition, PC No. 18 requires the interface ISearch.

Another reconstructed model places an emphasis on the connection of the reconstructed components by visualising the model as a component diagram (see Figure 10.4). In this diagram, the interface operations and data types are omitted. Instead, the connectors between the components are shown. The

¹In fact, the models in Figure 10.3 and Figure 10.4 were layouted manually. At the moment, Archimetric does only provide very basic auto-laying.

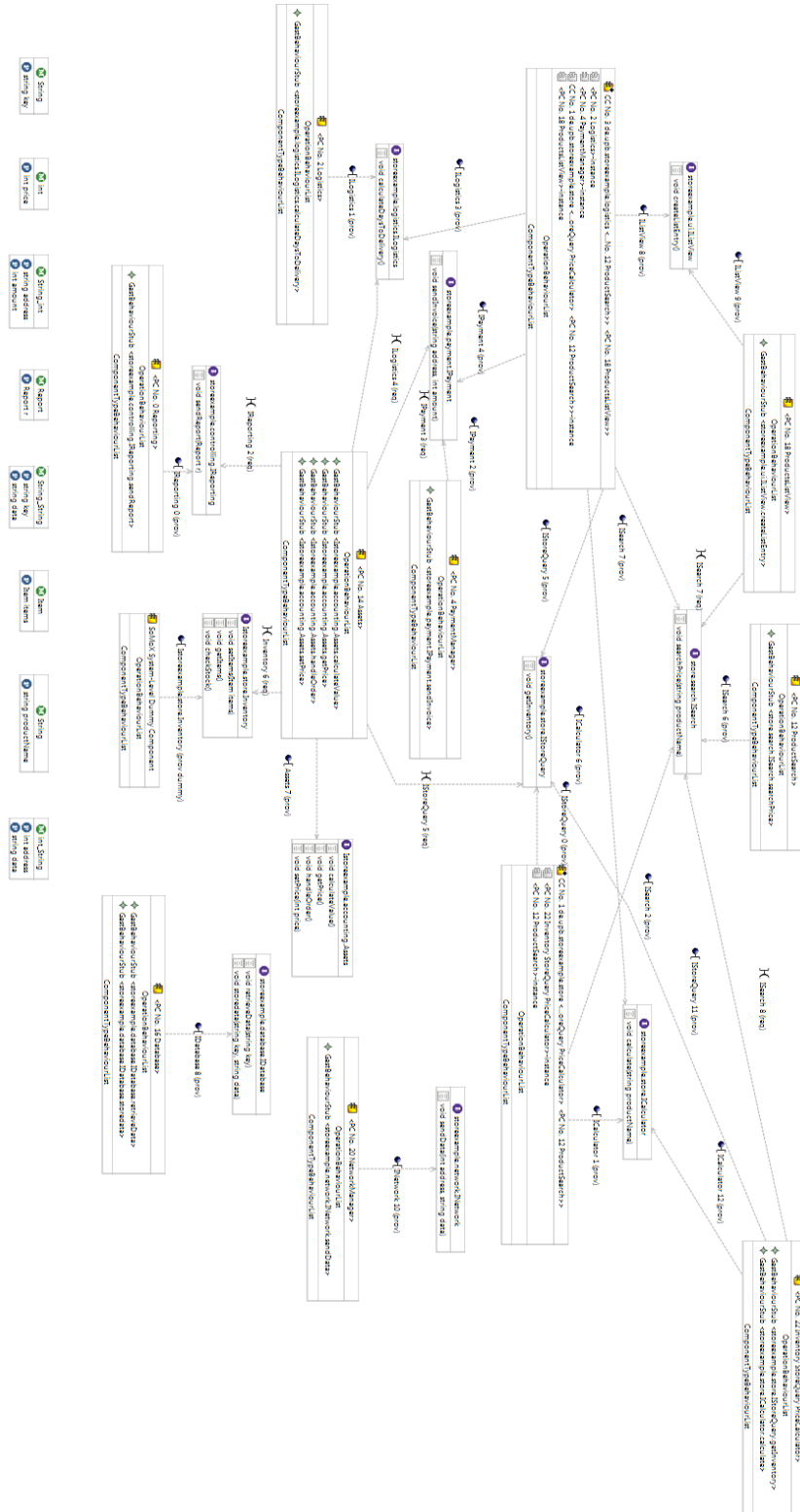
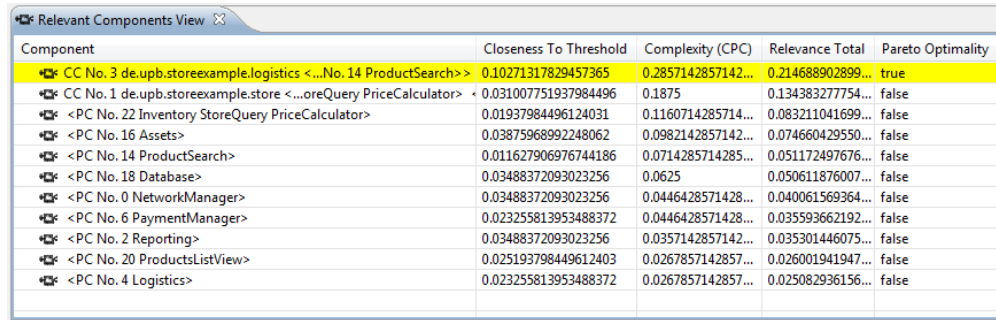


Figure 10.3.: Reconstructed components

reconstructed components are contained in one large component called SoMoX Reverse Engineered System representing the system boundary.

3. Execute relevance analysis After the architecture reconstruction, the component relevance analysis is executed. The results of this analysis are shown in a specialised tabular overview (Figure 10.5).



Component	Closeness To Threshold	Complexity (CPC)	Relevance Total	Pareto Optimality
CC No. 3 de.upb.storeexample.logistics <...No. 14 ProductSearch>	0.10271317829457365	0.2857142857142...	0.214688902899...	true
CC No. 1 de.upb.storeexample.store <...oreQuery PriceCalculator>	0.031007751937984496	0.1875	0.134383277754...	false
<PC No. 22 Inventory StoreQuery PriceCalculator>	0.01937984496124031	0.1160714285714...	0.083211041699...	false
<PC No. 16 Assets>	0.03875968992248062	0.0982142857142...	0.074660429550...	false
<PC No. 14 ProductSearch>	0.011627906976744186	0.0714285714285...	0.051172497676...	false
<PC No. 18 Database>	0.03488372093023256	0.0625	0.050611876007...	false
<PC No. 0 NetworkManager>	0.03488372093023256	0.0446428571428...	0.040061569364...	false
<PC No. 6 PaymentManager>	0.023255813953488372	0.0446428571428...	0.035593662192...	false
<PC No. 2 Reporting>	0.03488372093023256	0.0357142857142...	0.035301446075...	false
<PC No. 20 ProductsListView>	0.025193798449612403	0.0267857142857...	0.026001941947...	false
<PC No. 4 Logistics>	0.023255813953488372	0.0267857142857...	0.025082936156...	false

Figure 10.5.: Result of the component relevance analysis

The view consists of five columns. The first column contains the name of the component in question. (The names are automatically generated and assigned by SoMoX.) The following columns show the concrete metric values that were calculated for that component during the relevance analysis. These are the values of the single relevance metrics (Closeness To Threshold, Complexity) as well as of the aggregated value (Relevance Total, see Section 6.6). The last column indicates if the relevance value of the component is Pareto optimal with respect to the other components. Pareto optimal (i.e. very relevant) components are also highlighted in yellow.

4. Execute deficiency detection Before the detection of deficiencies is started, the deficiencies have to be formalised. The editor for deficiency formalisations is shown in Figure 10.6. It shows a formalisation of the *Interface Violation* deficiency (see Appendix B.1).

The components for which the deficiency detection is executed can be selected by the software architect. Typically, he will choose a subset of those components that were identified as the most relevant in the relevance analysis. Figure 10.7 shows the selection dialogue for the reconstructed components.

The detected deficiency occurrences are presented in a simple list (see Figure 10.8).

The detected deficiency occurrences can also be inspected individually. Reclipse offers three different views for the visualisation of detection results (see Section 7.3) Figure 10.9 shows the host graph view of a detected occurrence of the *Interface Violation* deficiency. It shows the section of the host graph which is involved in this deficiency occurrence. It is layouted similar to the deficiency formalisation (cf. Figure 10.6) in order to facilitate its interpretation by the software architect. In addition, it also shows selected attribute values of the elements from the host graph, e.g. the names of the objects.

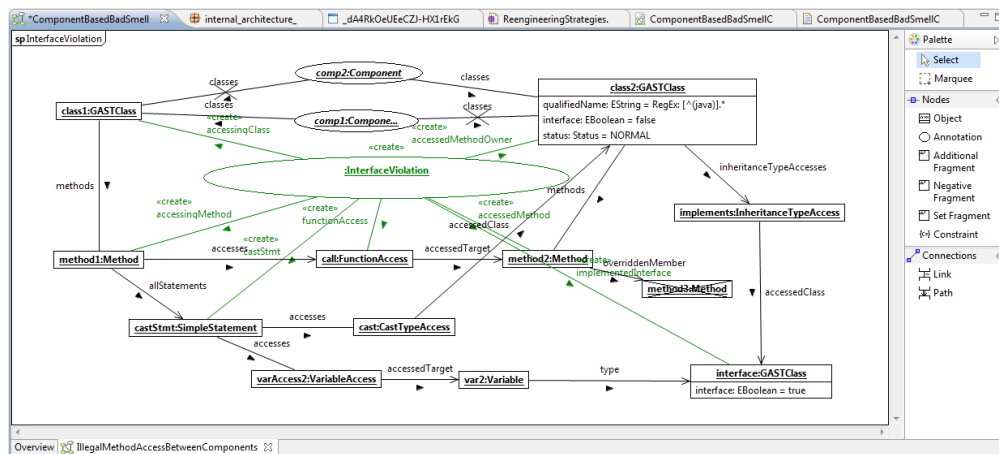
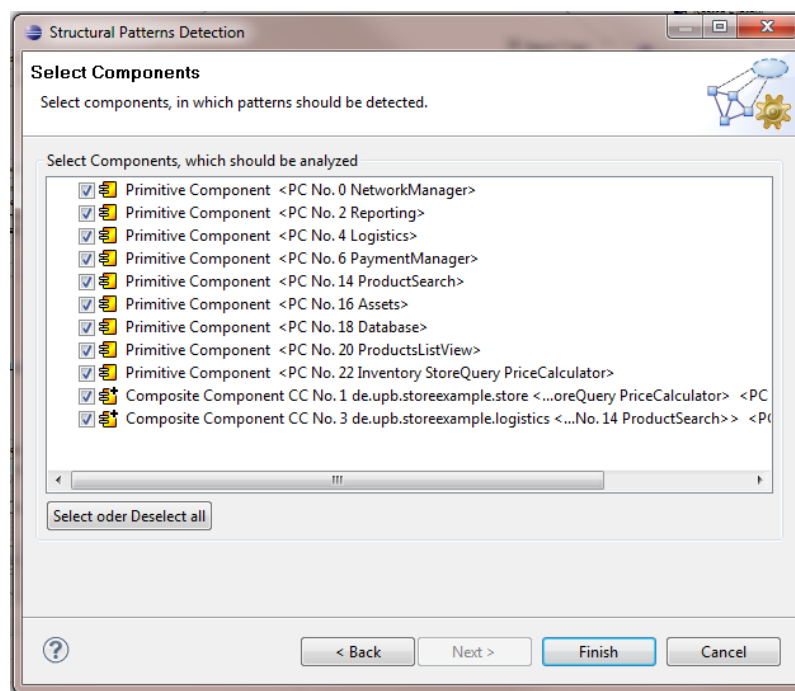
Figure 10.6.: Example deficiency specification: *Interface Violation*

Figure 10.7.: Dialogue for the selection of components for the deficiency detection

10. Validation

Annotation	Rating	Annotated Elements
DirectComponentClasses (58 annotations)		
DirectComposition (5 annotations)		
IllegalMethodAccess (11 annotations)		
IllegalMethodAccessBetweenComponents (11 annotations)		
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	96,85%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	97,20%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalMethodAccessBetweenComponents	96,85%	accessedMethod=getEntityManager, implementedInterface=PersistenceContext, castStmt=...
IllegalVariableAccessBetweenComponents (80 annotations)		
IndirectComponentClasses (5 annotations)		
IndirectComposition (2 annotations)		
NonTOCommunication (31 annotations)		
NonTOCommunication	96,27%	functionAccess=fillProductWithStockItemTO, calledClass=FillTransferObjects, nonTO=Stoc...
NonTOCommunication	94,54%	functionAccess=update, calledClass=LightDisplayController, nonTO=ExpressModeEnabled...

Figure 10.8.: List view of the detected deficiency occurrences

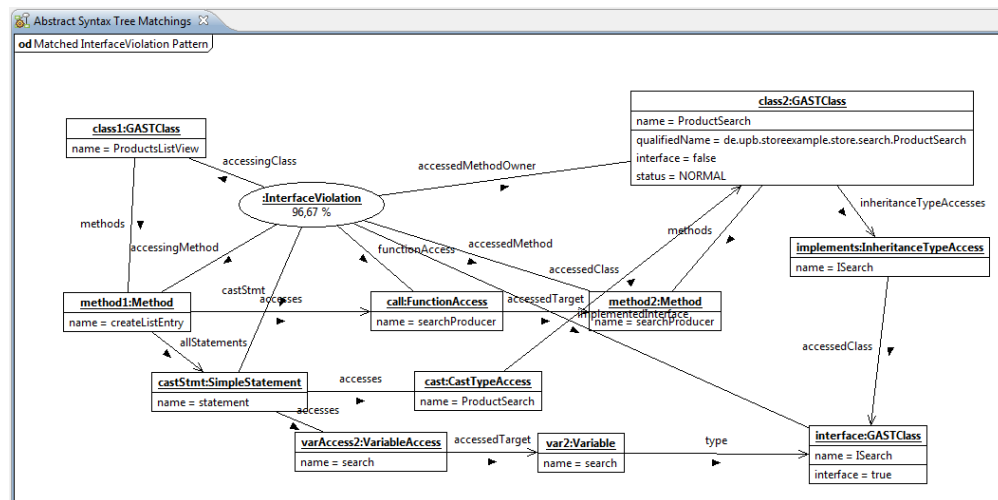
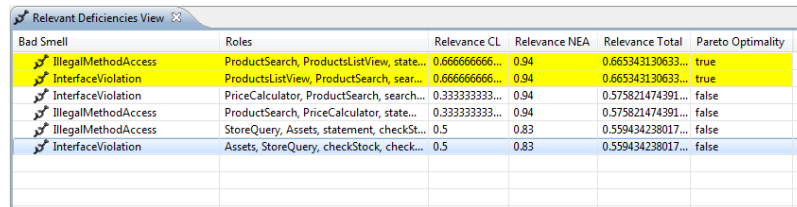


Figure 10.9.: Host graph view of a detected *Interface Violation* occurrence

5. Identify critical deficiencies In the next step, the detected deficiency occurrences are ranked in order to identify the occurrences that are the most critical. This is accomplished by the deficiency ranking (see Figure 10.10).



Bad Smell	Roles	Relevance CL	Relevance NEA	Relevance Total	Pareto Optimality
IllegalMethodAccess	ProductSearch, ProductsListView, state...	0.666666666...	0.94	0.665343130633...	true
InterfaceViolation	ProductsListView, ProductSearch, sear...	0.666666666...	0.94	0.665343130633...	true
InterfaceViolation	PriceCalculator, ProductSearch, search...	0.333333333...	0.94	0.575821474391...	false
IllegalMethodAccess	ProductSearch, PriceCalculator, state...	0.333333333...	0.94	0.575821474391...	false
IllegalMethodAccess	StoreQuery, Assets, statement, checkSt...	0.5	0.83	0.559434238017...	false
InterfaceViolation	Assets, StoreQuery, checkStock, check...	0.5	0.83	0.559434238017...	false

Figure 10.10.: Result of the design deficiency ranking

The results of the deficiency ranking are presented in a tabular view similar to the relevance analysis results. The first column shows the name of the deficiency while the second column states the elements (classes and methods) that play the different roles of the deficiency. Columns three and four show the particular ranking values while the fifth column presents the aggregated rank of the deficiency occurrence. The last column states if the deficiency occurrence is Pareto optimal with respect to the other occurrences. Again, the Pareto optimal deficiencies are highlighted in yellow. Those are the most critical deficiency occurrences that should be removed first.

6. Select automated removal strategy for the critical deficiency After the detected deficiency occurrences have been ranked, the software architect can remove them according to their severity. This example session presents the automatic removal of one deficiency occurrence through the application of a removal strategy. For this, the software architect first has to select the deficiency occurrence that is to be removed. Archimetric then only shows those removal strategies that are applicable to this type of deficiency (see Figure 10.11).

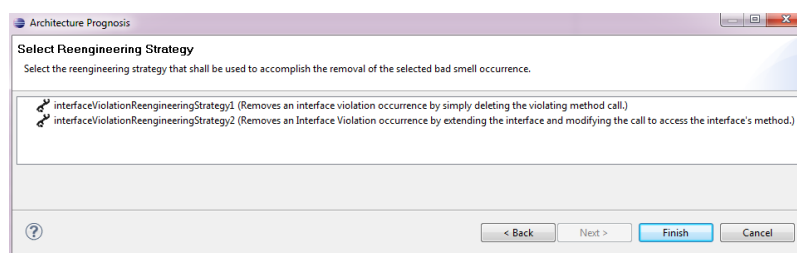


Figure 10.11.: Selection of an automated removal strategy

Before the strategy is executed and the system is changed, however, Archimetric can first calculate the effect of the reengineering on the architecture reconstruction. The changes that are to be expected are shown in the architecture preview in Figure 10.12.

The architecture preview consists of two parts. The upper part contains general information about the original architecture and the previewed architecture. This information includes the number of components in the architecture,

	Original Architecture	Predicted Architecture
Total Number of Components	12	11
Number of Primitive Components	10	10
Number of Composite Components	2	1
Total Number of Interfaces	11	11
Total Number of Messages	8	9

Original Architecture Components:

- <PC No. 2 Reporting> (1 classes)
- <PC No. 4 Logistics> (1 classes)
- <PC No. 6 PaymentManager> (1 classes)
- <PC No. 14 ProductSearch> (1 classes)
- <PC No. 16 Assets> (1 classes)
- <PC No. 18 Database> (1 classes)
- <PC No. 20 ProductsListView> (1 classes)
- <PC No. 22 Inventory StoreQuery PriceCalculator> (3 classes)
- CC No. 1 de.upb.storeexample.store <...oreQuery PriceCalculator> <PC No. 14 ProductSearch> (1 classes)
- <PC No. 22 Inventory StoreQuery PriceCalculator> (3 classes)
- <PC No. 14 ProductSearch> (1 classes)
- CC No. 3 de.upb.storeexample.logistics <...No. 14 ProductSearch>> <PC No. 20 ProductSearch> (1 classes)

Predicted Architecture Components:

- <PC No. 2 Reporting> (1 classes)
- <PC No. 4 Logistics> (1 classes)
- <PC No. 6 PaymentManager> (1 classes)
- <PC No. 14 ProductSearch> (1 classes)
- <PC No. 16 Assets> (1 classes)
- <PC No. 18 Database> (1 classes)
- <PC No. 20 ProductsListView> (1 classes)
- <PC No. 22 Inventory StoreQuery PriceCalculator> (3 classes)
- CC No. 1 de.upb.storeexample.store <... No. 14 ProductSearch> <PC No. 20 ProductSearch> (1 classes)
- <PC No. 22 Inventory StoreQuery PriceCalculator> (3 classes)
- <PC No. 14 ProductSearch> (1 classes)
- <PC No. 20 ProductSearch> (1 classes)

Figure 10.12.: Architecture preview

divided into primitive and composite components, and the number of interfaces. Value changes between the two architectures are emphasised in yellow.

The lower part lists all the components of the original architecture (left) and the previewed architecture (right), respectively. Components that exist unchanged in both versions of the architecture are not marked. Components that existed in the original architecture but no longer exist in the previewed architecture are marked red. These components are no longer reconstructed as they were before the reengineering. Components which still exist but which contain different components and classes than before are marked yellow. Finally, components which did not exist in the original architecture but are newly reconstructed after reengineering are marked green (not shown in Figure 10.12).

After the architecture preview is shown, the reengineering can be executed by applying the automated removal strategy. This will remove the selected deficiency occurrence by transforming the GAST of the system.

10.2. Experiment Setup

The validation was performed by myself and a student, mainly in the context of her master's thesis [Pla11]. In preparation for the case studies, we obtained the source code of the systems under study. The CoCoME reference implementation is freely available for download [CoC12]. The source of Palladio Fileshare and the SOFA implementation of CoCoME was made available to us by the respective developers. We studied the available documentation to familiarise ourselves with the systems.

We created deficiency formalisations for the four deficiencies described in this thesis, namely: *Transfer Object Ignorance*, *Interface Violation*, *Unauthorised Call*, and *Inheritance between Components*. We also formalised two automated removal strategies for the removal of *Interface Violation* occurrences. For the other deficiencies, no automated removal strategies were formalised.

The duration of the analysis steps was measured on a machine with an Intel Core i7-2620M processor with 2.7 GHz and 6 GB RAM.

10.3. Validation Questions

I evaluated Archimetrix with respect to the following validation questions:

- VQ1 Are the deficiency formalisations sufficiently precise to detect actual deficiency occurrences (instead of false positives)?
- VQ2 Do the defined design deficiencies occur in real-life systems, even if the systems were developed in a strictly component-based way?
- VQ3 Is the calculated component relevance value a good indicator of components in which the detection of design deficiencies is worthwhile?
- VQ4 How does the limitation of scope by the relevance analysis improve the deficiency detection compared to pure pattern matching?
- VQ5 Does the removal of the deficiencies that receive a high ranking value lead to architectural changes, and does the removal of deficiencies with a low ranking value leave the architecture unchanged, i.e. do the deficiency ranking heuristics work?
- VQ6 Is the recovered architecture after the removal of a relevant deficiency occurrence closer to the documented architecture?

10.4. Case Studies

In order to answer the validation questions, I selected three case studies in which we applied Archimetrix. The first case study is the simple store example system presented in Chapter 1. Second, we analysed Palladio Fileshare, a client-server file sharing platform. In the third case study, two implementations of the Common Component Modeling Example CoCoME [RRMP08] were compared. The remainder of this section describes the analysed systems and relates them to the validation questions.

Case Study 1: Store Example This system was built as a toy example. Due to its small size, the basic functionality of Archimetrix could be evaluated quickly. It also allowed the deliberate introduction of deficiencies. Therefore, it served for the validation of the deficiency formalisations (cf. Step 4.1 and 4.2 in Figure 4.2). The example system shows the basic feasibility of the Archimetrix process and demonstrates that deficiency occurrences have an effect on the architecture reconstruction.

Case Study 2: Palladio Fileshare Palladio Fileshare realises a client-server file sharing platform [KKR10]. It was developed in a strictly component-based way and was already used in the validation of SoMoX [Kro10]. Therefore, we did expect to find few design deficiencies in it, if any. Thus, the analysis of Palladio Fileshare is mainly targeted at validation question VQ2.

Case Study 3: CoCoME The Common Component Modeling Example CoCoME was created as a benchmark system for the comparison of different component frameworks [RRMP08]. It is meant to represent a typical component-based business information system. It was designed in a joint effort of several software engineering research groups and the design was thoroughly documented. Afterwards, a number of implementations of the system were created. On the one hand, a Java reference implementation was created by a group of students. On the other hand, a number of component frameworks, e.g. SOFA or FRACTAL, were used to create different implementations of the system.

Hence, the CoCoME system lends itself especially well to this validation. Its deliberate design as an exemplary, component-based business information system allows for the generalisation of the validation results to arbitrary business information systems. Additionally, in contrast to the general assumption of Archimatrix that no conceptual architecture is available, the conceptual software architecture of CoCoME is well-documented. This allows to compare the reconstructed architecture to the conceptual architecture and also to evaluate the changes in the reconstructed architecture.

As the reference implementation was created by a group of students and as it was written in Java which does not directly support component-oriented design, I expected to find a number of deficiencies in the code. In contrast, the SOFA implementation was created by researchers with the help of the SOFA component framework. This framework enforces good component-oriented design, e.g. by prohibiting the external access to methods which are not part of a component's interface. I did expect to find significantly fewer deficiencies in this implementation. Therefore, especially the reference implementation can help to answer validation questions VQ1 to VQ6 while the SOFA implementation is primarily targeted at questions VQ1 and VQ2. Furthermore, the results for both implementations can be compared with each other.

10.5. Threats to Validity

This section discusses the threats to the validity of this validation. It is split into a discussion of the internal threats to validity in Section 10.5.1 and of the external threats to validity in Section 10.5.2.

10.5.1. Threats to Internal Validity

In a controlled experiment, there are independent and dependent variables. The independent variables are changed during the course of the experiment and the behaviour of the dependent variables is observed. In order to draw valid conclusions about the behaviour of the dependent variables, the circumstances in which the experiment is conducted have to be controlled. This ensures that the

experiment yields the same, correct results every time it is repeated. The internal validity of an experiment describes of an experiment the degree to which the relevant interfering variables have been controlled [Pre01, p. 50ff]. Accordingly, threats to the internal validity are factors which may have influenced the outcome of the conducted experiment unintentionally.

Reuse of existing tools One important point to consider in this regard is the reuse of existing tools in Archimatrix. On the one hand, these tools can contain defects which produce incorrect results. On the other hand they can be applied incorrectly and therefore influence the outcome of the case studies.

One crucial factor in the use of SoMoX is the configuration of the metric weights for the architecture reconstruction (see Chapter 5). The software architect has to select a set of metric weights that is used in the architecture reconstruction. Different sets of weights can lead to drastically different reconstruction results. The selection of appropriate weights is a difficult task which can require a lot of trial-and-error work [BHT⁺10, Kro10]. The selection of metric weights in the case studies below was always based on the provided default values. These were then altered if necessary so that the reconstructed architecture contained a reasonable amount of components (not too few, not too many; our experience shows that an architecture with ten to twenty components has a reasonable degree of granularity to be useful). Because the conceptual architecture was known beforehand, the selection of metrics weights was influenced by it in that we tried to select weights for which the reconstructed architecture was “reasonably similar” to the conceptual one. Selecting different weights may have yielded different architectures. This could have influenced the subsequent process steps considerably.

Likewise, the use of Reclipse can have influenced the validation results. First, the pattern detection algorithm may contain defects and may therefore yield incorrect results. Second, the deficiency formalisations were completely created by two students and myself. This incurs the risk that deficiencies which actually exist in the systems under analysis were not detected due to too restrictive specifications (false negatives). Third, the detected deficiency occurrences were inspected manually in order to ensure that they were indeed true positives. Human misjudgement may lead to the acknowledgement of occurrences as true positives even if they are false positives, objectively.

Selection of design deficiencies The design deficiencies that were searched for in this thesis were selected based on well-known component-based design principles (see Chapter 3 and Appendix B). They are not meant to be an exhaustive representation but rather a sample of possible deficiencies. Obviously, a different sample might have yielded different results, possibly leading to different conclusions.

10.5.2. Threats to External Validity

A controlled experiment is conducted in order to draw general conclusions from the observed situation. The external validity is concerned with the generalisability of the experiment results [Pre01, p. 54ff]. Threats to the external validity are a factors which may prevent to generalise the observed effects.

Selection of design deficiencies The design deficiency formalisations used in the validation were in part tailored to the systems under analysis. For example, for case study 3 (CoCoME), the *Transfer Object Ignorance* deficiency formalisation was adapted to consider the project-specific naming convention for transfer objects (the suffix ‘TO’). When applying Archimetric to other systems, the formalisations would have to be adapted accordingly. There may be also other deficiencies which are more suitable for other systems. These could be discovered, documented, and formalised with the process proposed in Section 4.4.

Selection of case studies Naturally, the generalisability of validation results strongly depends on the selected case studies. In this regard, CoCoME has been specifically selected as a case study because it was designed as a benchmark system [RRMP08]. CoCoME is intended to represent an example of a typical business information system which can be used in the evaluation of analysis techniques for component-based systems. Palladio Fileshare on the other hand is a typical client-server system and is therefore also a good candidate for a case study. Therefore, I am confident that the results of this validation can be generalised.

10.6. Case Study 1: Store Example

In this section, the validation of the store example is presented. First, an overview of the system is presented. Then results of the validation are shown.

10.6.1. System Overview

The store example is a simplified, component-based system that was created exclusively for the validation of Archimetric. It has no real functionality but instead consists of a number of classes which represent a mock-up of a real system. The conceptual architecture of this system is shown and explained in Section 1.7.

During the implementation, a number of design deficiencies were deliberately introduced into the system. I introduced three *Interface Violation* occurrences and two *Transfer Object Ignorance* occurrences. Thereby, I was able to validate if the deficiency formalisations are appropriate for detecting these deficiency occurrences.

The implementation of the store example consists of 10 interfaces and 14 classes which are distributed to eight components. Because the system does only consist of mock-up code it comprises only 150 LOC.

Conceptual Architecture	
# Composite Components	0
# Primitive Components	8
Implementation	
# LOC	150
# Classes	14
# Interfaces	10

Table 10.1.: Properties of the Store Example system

10.6.2. Validation Results

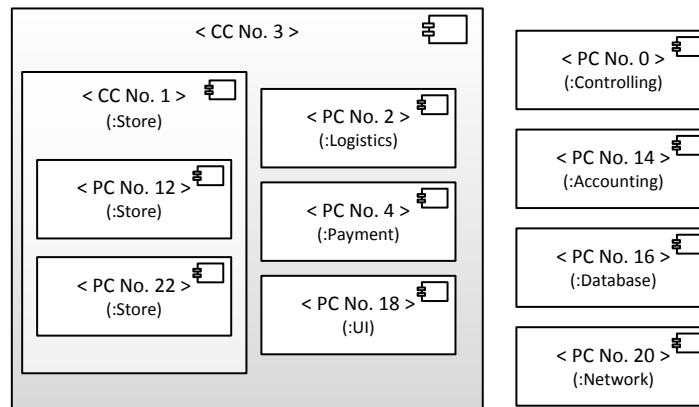


Figure 10.13.: Initially reconstructed architecture of the Store Example

Initial Architecture Reconstruction The result of the initial architecture reconstruction is visualised in Figure 10.13. The primitive components are labelled with numerical tags starting with PC, the composite components are labelled with tags starting with CC. The numerical tags are generated and assigned by the architecture reconstruction algorithm during the creation of the components of the architecture model. Interfaces and connectors are omitted in this figure for a better readability, although they are part of the formal model.

Nine primitive and two composite components were recovered. I compared the classes from the reconstructed components to the conceptual architecture and named the reconstructed components accordingly. The component names are displayed in parentheses in Figure 10.13.

Compared to the conceptual architecture, the Store component has been reconstructed as two separate primitive components, albeit inside the common

composite component CC No. 1. In addition, CC No. 1 has been encapsulated inside composite component CC No. 3 together with the primitive components Logistics, Payment, and UI.

Component	Relevance
CC No. 3	0.215
CC No. 1	0.134
PC No. 22	0.083
PC No. 16	0.075
PC No. 14	0.051
PC No. 18	0.051
PC No. 0	0.040
PC No. 12	0.036
PC No. 2	0.035
PC No. 20	0.026
PC No. 4	0.025
Duration	<1s

Table 10.2.: Component relevance analysis results for the Store Example

Component Relevance Analysis The results of the component relevance analysis are shown in Table 10.2. The most complex component, CC No. 3, was identified as the most relevant one.

Deficiency Detection and Ranking According to the results of the relevance analysis, I performed the deficiency detection in component CC No. 3. The detected deficiency occurrences are documented in Table 10.3.

All of the three deliberately introduced *Interface Violation* occurrences were detected. One of the two *Transfer Object Ignorance* occurrences was also detected. The other *Transfer Object Ignorance* occurrence exists between the two components Accounting and Controlling. Since neither of those components is contained in CC No. 3, it was not detected in this step.

Table 10.4 shows the ranking of the three detected *Interface Violation* occurrences. One of the occurrences, located between the classes `ProductsListView` and `ProductSearch`, was ranked to be the most relevant deficiency occurrence. It is located between the two primitive components PC No. 12 and PC No. 18

Design deficiency	Detected occurrences	
	CC No. 3	Whole System
Transfer Object Ignorance	1	2
Interface Violation	3	3
Unauthorised Call	0	0
Inheritance Between Components	0	0
Duration	6s	19s

Table 10.3.: Detected design deficiencies in the Store Example

Deficiency Removal and Subsequent Architecture Reconstruction The most critical *Interface Violation* occurrence was removed by applying an automatic removal strategy which extends the violated interface (see Section B.1.4). Then, the resulting architectural changes were previewed.

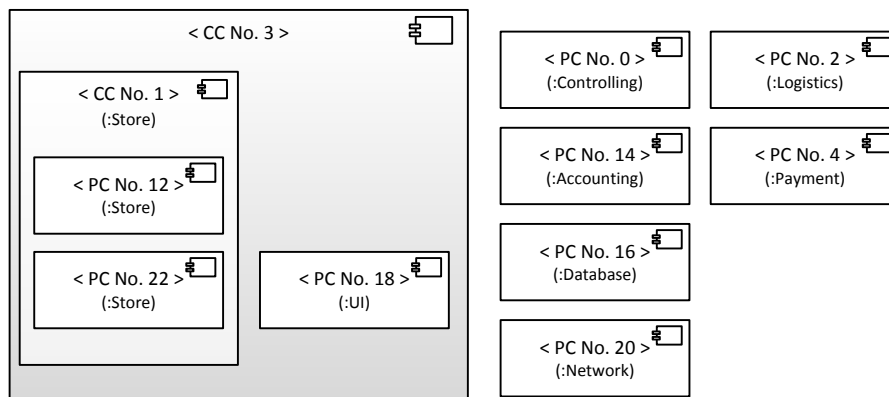
Figure 10.14.: Reconstructed architecture after the removal of the most relevant *Interface Violation* occurrence

Figure 10.14 shows the reconstructed architecture after the removal of the most critical deficiency occurrence. Compared to the originally reconstructed architecture, the two primitive components *Logistics* and *Payment* are no longer assigned to CC No. 3. Hence, the newly reconstructed architecture is now closer to the conceptual architecture.

10.6.3. Discussion

Although, the Store Example is a manually constructed, trivial system, it demonstrates that the prototype implementation of Archimetric can be used to apply the complete Archimetric process. The example exhibits the central

	Roles			Ranking	Corresponding Components
	interface	accessedMethod	accessingClass	accessingMethod	
#1	ISearch	searchProducer	ProductListView	createListViewEntry	0.6653 (opt.) PC No. 12 & PC No. 18
#2	ISearch	searchDiscount	PriceCalculator	calculate	0.5758 PC No. 12 & PC No. 22
#3	IStoreQuery	checkStock	Assets	checkStock	0.5594 PC No. 14 & PC No. 22

Table 10.4.: Detected *Interface Violation* occurrences in the Store Example

properties which are the motivation of this thesis: The source code of this system contains deficiencies which influence architecture reconstruction. The deficiency occurrences are detected in the component which was identified as the most relevant one. Removing the occurrence which was ranked as the most critical one leads to the reconstruction of an architecture which is closer to the conceptual architecture.

10.7. Case Study 2: Palladio Fileshare

This section presents the case study of the Palladio Fileshare system. First, an overview of the system under analysis is given. Then, the validation results are documented and discussed.

10.7.1. System Overview

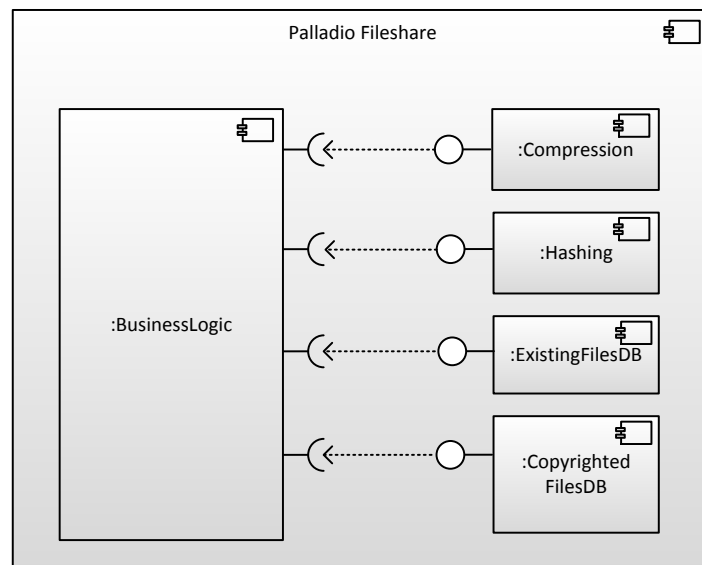


Figure 10.15.: Conceptual architecture of Palladio Fileshare (adapted from [KKR10])

Palladio Fileshare realises a server-based file sharing platform. As such, it represents a typical business information system and its component-based architecture is well-documented [KKR10]. The conceptual architecture of the system is shown in Figure 10.15. According to the documentation, Palladio Fileshare consists of five primitive components. The central component is the one labelled Business Logic. It contains all the relevant program logic and uses the four other primitive component Compression, Hashing, ExistingFilesDB, and CopyrightedFilesDB. These five components are encapsulated by one composite component representing the system boundary.

Table 10.7.1 lists some technical properties of the implementation of Palladio Fileshare. The system consists of 87 classes and six interfaces. The implementation comprises 7869 lines of code.

10.7.2. Validation Results

Initial Architecture Reconstruction For the validation, I first executed an architecture reconstruction with SoMoX. The clustering configuration is described

Conceptual Architecture	
# Composite Components	1
# Primitive Components	5
Implementation	
# LOC	7869
# Classes	87
# Interfaces	6

Table 10.5.: Properties of the Palladio Fileshare system

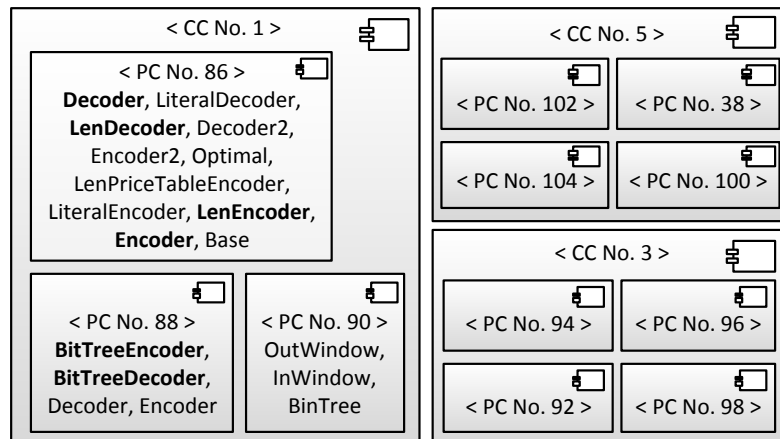


Figure 10.16.: Reconstructed architecture for Palladio Fileshare

in Appendix C. The architecture reconstruction took 9 seconds.

Figure 10.16 shows the reconstructed architecture. For the primitive components inside CC No. 1, Figure 10.16 also visualises the contained classes because they are relevant in the next validation step. Three composite components (labelled with CC) and eleven primitive components (labelled with PC) were detected in Palladio Fileshare.

Component	Relevance	Component (cont.)	Relevance (cont.)
CC No. 5	0.234	PC No. 88	0.036
CC No. 1	0.150	PC No. 90	0.029
PC No. 86	0.086	PC No. 92	0.021
PC No. 104	0.063	PC No. 94	0.014
CC No. 3	0.054	PC No. 98	0.008
PC No. 102	0.051	PC No. 96	0.008
PC No. 100	0.047	PC No. 38	0.006
Duration	1s		

Table 10.6.: Component relevance analysis results for Palladio Fileshare

Component Relevance Analysis In the next step, the component relevance analysis was performed. The results are shown in Table 10.6. It only took one second and identified the component CC No. 5 as the most relevant and CC No. 1 as the second most relevant component for design deficiency detection.

Deficiency Detection and Ranking Subsequently, I executed the design deficiency detection. The detection in component CC No. 5 yielded no results.

Design deficiency	Detected occurrences	
	CC No. 1	Whole System
Transfer Object Ignorance	11	11
Interface Violation	0	0
Unauthorised Call	0	0
Inheritance Between Components	0	0
Duration	2m 52s	8m 18s

Table 10.7.: Detected design deficiencies in Palladio Fileshare

Therefore, the second-most relevant component, CC No. 1, was searched next. The detection for CC No. 1 took 2 minutes and 52 seconds.

The detection results are presented in Table 10.7. Column 1 lists the different design deficiencies. The second column shows the results for the analysis of component CC No. 1 and the third column shows the results for the analysis on the whole system. The deficiency detection found eleven occurrences of the *Transfer Object Ignorance* deficiency in CC No. 1. Some of them are located in the class `LenEncoder` which was assigned to the primitive component PC No. 86. There, methods of the class `BitTreeEncoder` in the component PC No. 88 are called and objects of the type `Encoder` are passed as parameters. Classes that are involved in the detected deficiency occurrences are marked in boldface in Figure 10.16. A manual inspection verified that `Encoder` is not a typical data class since it contains several methods that are neither getters nor setters. Instead, they contain more complex application logic.

The same situation occurs for the call of the class `BitTreeDecoder` and a parameter of the type `Decoder`. Consequently, the communication between these components violates the component-oriented design principle that transfer objects have to be used for interactions between different components. In the deficiency removal step, we removed the detected deficiencies one by one through manual reengineering. Subsequent architecture reconstructions showed that the reconstructed architecture did not change. Therefore, we conclude that in case of this case study, the detected deficiencies did not influence the metric values enough to influence the architecture reconstruction.

10.7.3. Discussion

This section answers the validation questions in terms of the validation results for Palladio Fileshare.

Are the deficiency formalisations sufficiently precise to detect actual deficiency occurrences (instead of false positives)? (VQ1) The deficiency detection found eleven occurrences of the *Transfer Object Ignorance* deficiency. All of them were manually inspected and found to be true positives. At least for Palladio Fileshare, the formalisation of the *Transfer Object Ignorance* deficiency seems to be appropriate.

Do the defined design deficiencies occur in real-life systems, even if the systems were developed in a strictly component-based way? (VQ2) Palladio Fileshare was developed by experienced researchers and was intended to be developed in a strictly component-based way [KKR10]. I did not analyse the system manually before and had no prior knowledge of contained deficiencies. Instead, I derived the deficiency from component-oriented design principles in literature [ACM01, Fow02]. Still, the implementation contained eleven occurrences of the *Transfer Object Ignorance* deficiency. This shows that the *Transfer Object Ignorance* deficiency does occur in practice.

Is the calculated component relevance value a good indicator of components in which the detection of design deficiencies is worthwhile? (VQ3) The relevance analysis identified composite component CC No. 5 as the most relevant component followed by CC No. 1. Despite its high relevance value, however, CC No. 5 did not contain any of the searched deficiencies. CC No. 1 on the other hand did contain all deficiencies that were found in the system. Since the deficiencies occur between the two primitive components PC No. 86 and PC No. 88, a detection that is just focused on PC No. 86, the third-highest-ranked component also yields no results.

This result stresses the heuristic nature of the component relevance analysis. A high relevance value does not guarantee the existence of deficiency occurrence. On the contrary, even in systems without any deficiencies, one component would be ranked as the most relevant.

How does the limitation of the scope for the deficiency detection by the relevance analysis improve scalability with respect to pure pattern matching? (Q4) Focusing the deficiency detection on CC No. 1 increases the performance of the analysis considerably. The deficiency detection in the whole system was performed in eight minutes and 18 seconds. In contrast, the detection in CC No. 1 only took one third of the time, i.e. two minutes and 52 seconds.

10.8. Case Study 3: CoCoME

This section reports on the case study of the Common Component Modeling Example CoCoME. In this case study, two implementations of the same system were analysed. First, an overview of the conceptual architecture of CoCoME is given which is identical for both implementations. Then, the validation results are presented, first for the reference implementation and then for the SOFA implementation. The section closes with a discussion of the results.

10.8.1. System Overview

CoCoME represents a component-based trading system. Its well-documented architecture is intended to illustrate good component-oriented design.

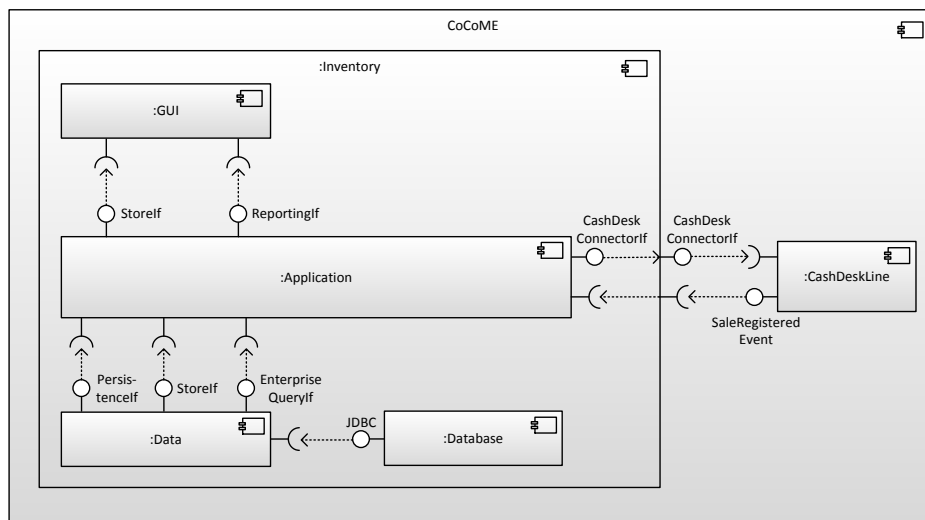


Figure 10.17.: An overview of the conceptual architecture of CoCoME (adapted from [HKW⁺08])

Figure 10.17 shows an overview of CoCoME’s conceptual architecture as documented in [HKW⁺08]. The system consists of two parts: an Inventory component and a CashDeskLine component. The former is meant to handle the management of the store inventory while the latter is responsible for the functionality of the cash desks. These two components can communicate via the interfaces CashDeskConnectorlf and SaleRegisteredEvent. Both, Inventory and CashDeskLine, are composite components. Inventory contains the four components GUI, Application, Data, and Database. As the results of this case study are mainly concerned with the Inventory component, and especially its subcomponents Application and Data, the other components are not presented in detail, here.

Figure 10.18 shows a more detailed view of the components Application and Data. The component Data consists of the three sub components Enterprise, Persistence, and Store. They all provide interfaces with corresponding names but are

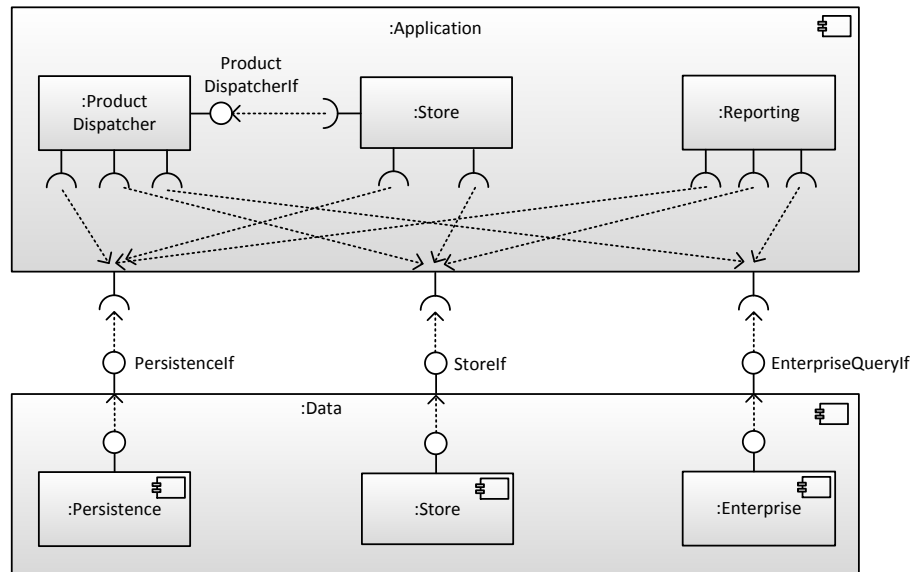


Figure 10.18.: A detailed view of the components `Inventory::Application` and `Inventory::Data`

not supposed to communicate with each other. The component `Application` also contains three sub components: `ProductDispatcher`, `Store`, and `Reporting`. `ProductDispatcher` and `Reporting` require all three interfaces from the `Data` component. `Store` requires only the two interfaces: `Persistenceelf` and `Storeelf`. In addition, `Store` can communicate with `ProductDispatcher` via the interface `ProductDispatcherIf`.

Table 10.8.1 shows the properties of the conceptual architecture of CoCoME and the two implementations. The conceptual architecture consists of eight composite components and 19 primitive components.

The reference implementation of CoCoME was created manually in plain Java. It consists of 127 classes and 21 interfaces and comprises more than 5000 lines of code. The reference implementation is freely available for download on the CoCoME website [CoC12].

The SOFA implementation of CoCoME was realised with the SOFA component framework [BHP06]. It contains more than double the amount of lines of code compared to the reference implementation. The number of classes and number of interfaces are also slightly higher.

10.8.2. Reference Implementation Validation Results

Initial Architecture Reconstruction The initial reconstruction of the CoCoME system from the source code of the reference implementation took 13 seconds. Eight primitive components and six composite components were reconstructed. A visualisation of the recovered architecture is shown in Figure 10.19. In ad-

Conceptual Architecture	
# Composite Components	8
# Primitive Components	19
Reference implementation	
# LOC	5116
# Classes	127
# Interfaces	21
SOFA implementation	
LOC	10502
# Classes	153
# Interfaces	28

Table 10.8.: Properties of the reference implementation of the CoCoME system

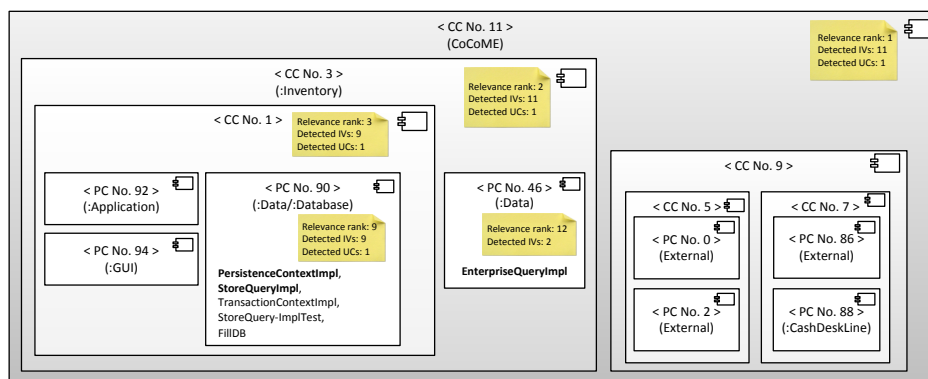


Figure 10.19.: Initially reconstructed architecture for the reference implementation of CoCoME

dition to the component names from the clustering, the names of the corresponding components from the conceptual architecture were mapped manually to the recovered components by comparing the contained classes. The names are displayed in parentheses. For example, the two primitive components PC No. 90 and PC No. 46 can be mapped to the `Inventory::Data` component from the conceptual architecture. For some components, which become relevant later in the discussion of the results, their relevance ranking and the number of detected deficiencies is shown in the yellow notes.

Compared to the conceptual architecture, the basic structure of the system is recognisable: Two main components, CC No. 3 and CC No. 9, were reconstructed representing the components `Inventory` and `CashDeskLine`. CC No. 9 also contains some components labelled `External` which contain classes that represent the bank. They are part of the reference implementation so that the interaction of bank and trading system can be simulated. They are, however, not part of the conceptual architecture. Component CC No. 3 contains the sub components that are also present in the conceptual architecture. They are arranged a little differently, though. The sub components `GUI` and `Application` were reconstructed as documented. The component `Data` is split into two components, (PC No. 46 and PC No. 90), which belong to different composite components (CC No. 3 and CC No. 5). PC No. 90 also contains the classes that would normally belong to the `Database` component. The primitive components PC No. 90, PC No. 92, and PC No. 94 are also encapsulated by an additional composite component, CC No. 1, which is not part of the conceptual architecture.

Component	Relevance	Component (cont.)	Relevance (cont.)
CC No. 11	0.7398	PC No. 94	0.1551
CC No. 3	0.4375	PC No. 90	0.0832
CC No. 1	0.4246	CC No. 5	0.0254
CC No. 9	0.3025	PC No. 88	0.0200
CC No. 7	0.2772	PC No. 46	0.0129
PC No. 86	0.2574	PC No. 0	0.0059
PC No. 92	0.1897	PC No. 2	0.0030
Duration	13s		

Table 10.9.: Component relevance analysis results for the reference implementation of CoCoME

Component Relevance Analysis In the next step, the component relevance analysis was performed. Table 10.9 lists the detected components and their relevance values. The first column contains the names assigned to the com-

Design deficiency	Detected occurrences	
	CC No. 3	Whole System
Transfer Object Ignorance	0	0
Interface Violation	11	11
Unauthorised Call	1	1
Inheritance Between Components	0	0
Duration	2m 26s	6m 25s

Table 10.10.: Detected design deficiencies in the reference implementation of CoCoME

ponents during the clustering. The second column shows the relevance values for the components. The component relevance analysis took 13 seconds. The three composite components that received the highest relevance ratings are CC No. 11, CC No. 3 and CC No. 1.

Deficiency Detection and Ranking In the next step, we applied the design deficiency detection. According to the results of the component relevance analysis, CC No. 11 is the most relevant component. However, as CC No. 11 encompasses the whole system, we chose the second most relevant component, CC No. 3, to be searched for design deficiencies. Additionally, we executed a design deficiency detection on the whole system and compared the results as well as the runtime.

Table 10.10 shows the detection results. The analysis of CC No. 3 took two minutes and 26 seconds and the analysis of the whole system took six minutes and 25 seconds. We detected eleven *Interface Violations* and one *Unauthorised Call* which were all identified as true positives by manual inspection.

Figure 10.19 also shows the number of deficiencies that were detected per component. As depicted there, all deficiencies were detected in the composite component CC No. 3. They are distributed among the primitive components PC No. 90 and PC No. 46. In PC No. 90, nine *Interface Violations* and the *Unauthorised Call* were detected. In component PC No. 46, two *Interface Violation* occurrences were detected.

The eleven occurrences of the design deficiency *Interface Violation* are documented in detail in Table 10.11. In the second column, the table lists the roles from the deficiency. For every detected deficiency occurrence, the elements which play the roles in that occurrence are shown. Column 3 shows the deficiency ranking values (Pareto optimal values are annotated with “(opt.)”). Column 4 lists the components which contain the participating classes.

All eleven occurrences concern the interface *PersistenceContext* and the method *getEntityManager* (located in the subclass *PersistenceContextImpl*). Two occurrences (#1 and #2) are located in the class *EnterpriseQueryImpl*. For the other occurrences, the accessing class is *StoreQueryImpl*. In the design deficiency ranking, occurrences #1 and #2 received a higher ranking (0.5562) than the other

		Roles		Ranking		Corresponding Components
interface	accessedMethod	accessingClass	accessingMethod			
#1	PersistenceContext	getEntityManager	EnterpriseQueryImpl	getQueryEnterpriseById	0.5562 (opt.)	PC 46 & PC 90
#2	PersistenceContext	getEntityManager	EnterpriseQueryImpl	getMeanTimeToDelivery	0.5562 (opt.)	PC 46 & PC 90
#3	PersistenceContext	getEntityManager	StoreQueryImpl	queryAllStockItems	0.4047	PC 90
#4	PersistenceContext	getEntityManager	StoreQueryImpl	queryLowStockItems	0.4047	PC 90
#5	PersistenceContext	getEntityManager	StoreQueryImpl	queryOrderById	0.4047	PC 90
#6	PersistenceContext	getEntityManager	StoreQueryImpl	queryProductById	0.4047	PC 90
#7	PersistenceContext	getEntityManager	StoreQueryImpl	getStockItems	0.4047	PC 90
#8	PersistenceContext	getEntityManager	StoreQueryImpl	queryStockItemById	0.4047	PC 90
#9	PersistenceContext	getEntityManager	StoreQueryImpl	queryStoreById	0.4047	PC 90
#10	PersistenceContext	getEntityManager	StoreQueryImpl	queryProducts	0.4047	PC 90
#11	PersistenceContext	getEntityManager	StoreQueryImpl	queryStockItem	0.4047	PC 90

Table 10.11.: Detected *Interface Violation* occurrences in the reference implementation of CoCoME and their ranking

ones (0.4047). Their relevance values are Pareto optimal. As depicted in the last column of the table, these occurrences involve two components (PC No. 46 and PC No. 90) while the other occurrences are located entirely within the component PC No. 90.

The detected *Unauthorised Call* occurrence concerns the same classes and is also located in the method `getEntityManager`.

Deficiency Removal and Subsequent Architecture Reconstruction The *Interface Violation* occurrence #1, being one of the two most relevant *Interface Violation* occurrences, was selected to be removed automatically by extending the interface `PersistenceContext`. The corresponding removal strategy is documented in Appendix B.1.4.

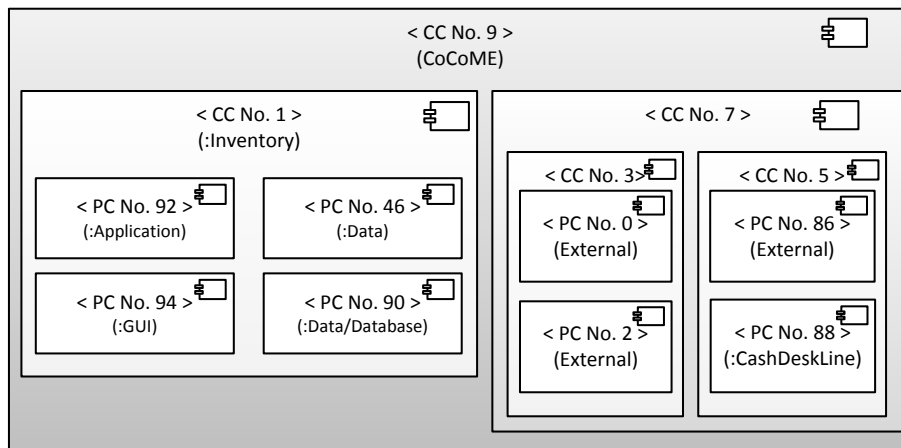


Figure 10.20.: Architecture preview for removal of *Interface Violation* occurrence #1

Figure 10.20 depicts the previewed architecture for the selected deficiency removal step. In comparison to the originally recovered architecture, the composite component, encapsulating the primitive components `:Application`, `:GUI`, and `:Data/:Database`, is no longer reconstructed. Thereby, the two primitive components `:Data/:Database` and `:Data` belong to the same composite component `(:Inventory)` in the previewed architecture. The previewed architecture is closer to the conceptual architecture (see Figure 10.17) than the originally reconstructed architecture (see Figure 10.19). However, the components PC No. 90 and PC No. 46 are still not reconstructed as designed.

Afterwards, the architecture preview for the removal of *Interface Violation* occurrence #2 with the same removal strategy was computed (without having removed *Interface Violation* #1, first). The result was the same architecture as in Figure 10.20.

Then, all eleven *Interface Violations* were removed successively. *Interface Violation* #1 was removed with the removal strategy as described above. As

all other occurrences are concerned with the same interface (`PersistenceContext`), they could be removed more easily afterwards as the interface had already been extended by the first removal. These ten remaining removals were performed manually.

The architecture that was reconstructed after all *Interface Violation* occurrences had been removed was identical to the one presented in Figure 10.20. Hence, the architecture did not change again after the first deficiency occurrence had been removed.

In order to validate the deficiency ranking, we changed the order of the deficiency removal. Starting again with the original, unmodified reference implementation, we first removed *Interface Violation* occurrence #3, one of the less relevant design deficiency occurrences. The removal was performed by applying the same automatic removal strategy as before. In this case, the previewed architecture was identical to the originally reconstructed one (see Figure 10.19). The removal of the less relevant deficiency occurrence did not influence the architecture reconstruction. The same applies for removing each of *Interface Violation* occurrences #4 to #11, first. The reconstructed architecture changes when occurrences #1 or #2 are removed but not for the other occurrences.

10.8.3. SOFA Implementation Validation Results

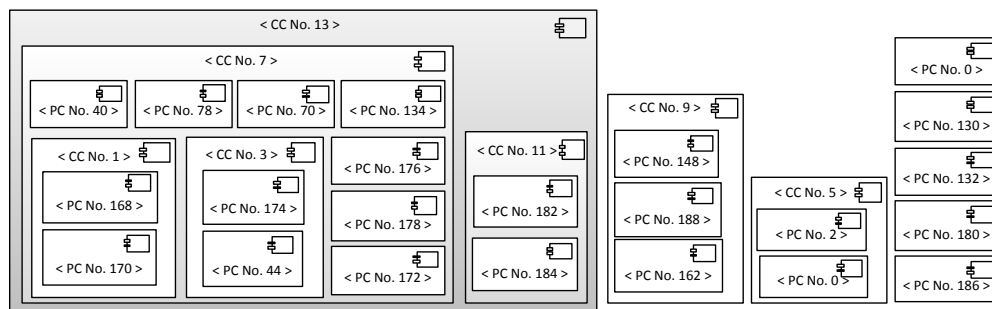


Figure 10.21.: Initially reconstructed architecture for the SOFA implementation of CoCoME

Initial Architecture Reconstruction The reconstructed architecture for the SOFA implementation of CoCoME is depicted in Figure 10.21. Seven composite components and 23 primitive components were recovered. The architecture reconstruction took 27 seconds.

Component Relevance Analysis Table 10.12 shows the results of the component relevance analysis. The duration of the relevance analysis was 13 seconds.

The component relevance analysis evaluated composite component CC No. 13 to be the most relevant component for the design deficiency detection. Several components received a relevance value below 0.01. A manual inspection showed

Component	Relevance	Component (cont.)	Relevance (cont.)
CC No. 13	0.1612	PC No. 176	0.0149
CC No. 7	0.1163	PC No. 166	0.0125
CC No. 9	0.0452	PC No. 174	0.0092
PC No. 178	0.0438	PC No. 172	0.0083
PC No. 0	0.0421	PC No. 132	0.0065
CC No. 11	0.0342	PC No. 146	0.0053
CC No. 3	0.0324	PC No. 42	0.0049
CC No. 5	0.0309	PC No. 76	0.0040
CC No. 1	0.0305	PC No. 158	0.0034
PC No. 170	0.0280	PC No. 130	0.0026
PC No. 184	0.0257	PC No. 38	0.0024
PC No. 168	0.0222	PC No. 68	0.0024
PC No. 164	0.0180	PC No. 128	0.0020
PC No. 182	0.0179	PC No. 180	0.0014
Duration	13s		

Table 10.12.: Component relevance analysis results for the SOFA implementation of CoCoME

that these are primitive components which only contain one class each. Therefore, they are assigned very low complexity values resulting in equally low total relevance values.

Deficiency Detection and Ranking Accordingly, I first executed a design deficiency detection on CC No. 13 and then, for comparison, on the whole system containing all components. Table 10.13 shows the results.

The analysis on the whole system took nearly an hour, while the analysis of CC No. 13 only took a few minutes. I detected two similar *Interface Violation* occurrences in the class `AbstractReportingServiceClient` in the component PC No. 174 (which is contained in CC No. 13). The manual inspection showed that the class `ReportProxy` that contains the accessed method is a data class, which makes these violations variants of the *Interface Violation* deficiency. There is also one occurrence of the *Unauthorised Call* design deficiency. The class `DataDownloadAction` calls one method of the class `DataExchangeClientObjectUpdater` in another component. This is not allowed since the first component does not require an interface provided by the latter component.

Design deficiency	Detected occurrences	
	CC No. 13	Whole System
Transfer Object Ignorance	0	0
Interface Violation	2	2
Unauthorised Call	1	1
Inheritance Between Components	1	1
Runtime	3m 27s	54m 33s

Table 10.13.: Detected design deficiencies in the SOFA implementation of CoCoME

Furthermore, I detected an occurrence of the design deficiency *Inheritance between Components*. This deficiency occurs between the classes `AbstractNeodatisDBObjectUpdater` and `DataExchangeClientObjectUpdater` with the latter being a sub class of the former. `DataExchangeClientObjectUpdater` is assigned to PC No. 38

Deficiency Removal and Subsequent Architecture Reconstruction I removed both *Interface Violation* occurrences by extending the violated interface. The architecture preview showed that the architecture reconstruction was not influenced by those deficiency occurrences. The reconstructed architecture was the same as before.

The occurrence of the *Unauthorised Call* deficiency was not removed in this case study. For a discussion on the removal of *Inheritance between Components* occurrences, see Appendix B.3.

10.8.4. Discussion

This section discusses the results of the case study in terms of the validation questions presented in Section 10.3.

Are the deficiency formalisations sufficiently precise to detect actual deficiency occurrences (instead of false positives)? (VQ1) Since a detailed design document is available for the CoCoME system, both implementations were expected to exhibit a good component-oriented design. I also expected to find deficiencies in the reference implementation as it was created manually by a group of students. In contrast, the SOFA implementation was created by researchers with the help of a component framework. Therefore, I did not expect it to contain any deficiencies. A large number of detection results in the SOFA implementation could have been indicative of imprecise formalisations. However, only a handful of deficiency occurrences were detected in the analysed systems (12 in the reference implementation of CoCoME, 4 in the SOFA implementation). Manual inspection ensured, that the detected occurrences are real

design problems and not false positives. Therefore, the validation results confirm my expectations and validation question VQ1 can be answered positively for this case study.

Do the defined design deficiencies occur in real-life systems, even if the systems were developed in a strictly component-based way? (VQ2) The reference implementation of CoCoME was created manually by a group of students. The design documentation is very detailed and clear. However, the validation results show that the data component was not implemented as specified. As the detected deficiency occurrences were all concerned with the same classes, it is possible that they all were introduced by the same developer. This developer might not have known the conceptual architecture which, in turn, might have led to the deficiencies.

The implementation of CoCoME with the SOFA framework also contains several design deficiencies. This is more surprising than for the reference implementation as the framework should enforce good component-oriented design and prevent the introduction of deficiencies.

It stands to reason that these problems will be even more significant in larger, more complex systems. Large business information systems can contain millions of lines of code. The validation results suggest that in spite of clear conceptual architectures and the use of frameworks, developers will still introduce deficiencies.

In summary, this case study shows that the design deficiencies presented in this thesis do occur in practice. This is even the case for implementations which were created with the support of a component framework. This allows to answer validation question VQ2 positively.

Is the calculated component relevance value a good indicator of components in which the detection of design deficiencies is worthwhile? (VQ3) In the CoCoME reference implementation, composite component CC No. 11 was identified as the most relevant component. However, as this component contains all other reconstructed components, it trivially also contains all deficiencies. Obviously, it is also the most complex of all components which partly explains its high relevance value. This suggests that components containing the complete system should be excluded from the relevance analysis in future.

CC No. 3 which received the second-highest relevance rank contained all deficiencies (eleven *Interface Violations* and one *Unauthorised Call*). These deficiencies were distributed among its contained primitive components PC No. 90 and PC No. 46. These primitive components only received very low deficiency ranks (#9 and #12). So while CC No. 3 supports the hypothesis that relevant components contain deficiencies, the results for the primitive components contradict the hypothesis.

For the SOFA implementation, the most relevant component, CC No. 13 was the one that contained all deficiencies. However, it also contained the majority of all reconstructed components and therefore, similar to the reference implementation, large parts of the system.

These results seem to indicate that the relevance analysis performs rather poorly and they warrant a discussion of the employed relevance metrics. The complexity is a well-proven heuristic for the identification of defect-prone classes and components [Rie96, BB01, Gla03]. However, this does neither guarantee that a complex component will contain deficiencies nor does it mean that simple components are always correct. The second relevance metric, closeness to threshold, does identify worthwhile components in the sense that changes in these components might affect the reconstructed architecture. Hence, this metric is not indicative of a component's probability to contain deficiencies.

These two observations explain the validation results: components with a high relevance value *are* a worthwhile input for the deficiency detection. This however does not necessarily mean that they will contain deficiencies. In the future, additional relevance metrics could be added to remedy this problem.

How does the limitation of scope by the relevance analysis improve the deficiency detection compared to pure pattern matching? (VQ4) For the reference implementation of CoCoME, the deficiency detection in the selected relevant component took approximately a third of the detection time in the complete system (two minutes and 26 seconds compared to six minutes and 25 seconds). This time reduction is even more significant in the SOFA implementation: The deficiency detection in the most relevant component took three minutes and 27 seconds. The detection in the whole system was accomplished in 54 minutes and 33 seconds. The scope limitation nearly led to a speed-up by a factor of 16, even though the selected component contained large parts of the complete system (cf. the discussion of VQ3). In part, this can probably be attributed to the exponential complexity of the graph matching algorithm which is used for the deficiency detection. On the other hand, factors like model loading and caching may also play a part, here. In the search of the selected relevant component, large parts of the underlying model can be ignored, possibly also contributing to the speed-up. To answer the question why the performance gain is so large, the deficiency detection would have to be profiled and analysed for both cases. This, however, is outside the scope of this thesis.

These results suggest that the deficiency detection can be made substantially more efficient by concentrating on a subset of components instead of searching the complete system. This effect will be even more important in larger systems in which the complete detection is not feasible due to time or memory constraints. So validation question VQ4 can be answered with “yes”.

Does the removal of the deficiencies that receive a high ranking value lead to architectural changes, and does the removal of deficiencies with a low ranking value leave the architecture unchanged, i.e. do the deficiency ranking heuristics work? (VQ5) The *Interface Violations* that were detected in the reference implementation received two different ranks. The removal of the two higher ranked occurrences did affect the architecture reconstruction. In contrast, removing one of the nine lower ranked occurrences did not lead to a change in the reconstructed architecture.

These results show that there are cases in which the removal of just one deficiency occurrence can influence the architecture reconstruction. It is however not clear which occurrences have an effect on the architecture reconstruction and which do not. The results for the reference implementation seem to indicate that higher ranked deficiency occurrences are more likely to have an influence than lower ranked occurrences. But this does not mean that the removal of highly ranked occurrences will always influence the architecture reconstruction. The removal of any of the deficiency occurrences in the SOFA implementation did not lead to a change in the architecture. Thus, the software architect has to determine if a deficiency removal changes the reconstructed architecture either by using the architecture preview or by manual removal and subsequent architecture reconstruction.

The case studies suggest, however, that if the removal of a deficiency occurrence will have an effect on the architecture reconstruction, it will be the removal of the highest ranked occurrence.

Is the recovered architecture after the removal of a relevant deficiency occurrence closer to the documented architecture? (VQ6) In the reference implementation, the reconstructed architecture differed from the conceptual architecture mainly regarding the Inventory component. After the removal of one of the highly ranked interface violations, however, the reconstructed architecture changed. While the classes belonging to the conceptual Data component were still distributed among different primitive components, the reconstructed architecture was definitely closer to the conceptual one.

As the removal of deficiency occurrences in the SOFA implementation did not influence the architecture, this validation question cannot be answered for the second part of this case study.

This means that Archimetrix correctly identified design deficiency occurrences whose removal lead to an architecture that is closer to the conceptual architecture than the originally reconstructed architecture. Design deficiency occurrences whose removal did not change the architecture received a lower ranking. These findings allow a positive answer to validation question VQ6.

10.9. Time and Effort

In summary, the execution of the different analysis steps, only took a few minutes for each system. The reported durations are just the analysis durations without the time needed for the loading of the analysed models, e.g. the GAST. Archimetrix uses the EMF framework which is not designed for model loading performance. Thus, approximately one to two hours have to be added for the loading of the models, depending on the size of the system.

For the architecture reconstruction step, the software architect has to determine a suitable clustering configuration. This is a non-trivial task which requires some experience. Our experiments with SoMoX showed that it is a reasonable heuristic to aim for the reconstruction of an architecture with ten to twenty components. Since an architecture reconstruction run only takes a

few seconds for each of the analysed systems and since there is an approved default configuration to start with, finding a suitable configuration can often be accomplished within about one hour.

Furthermore, the time for the documentation and formalisation of deficiencies and removal strategies has to be taken into account. As described in Section 4.4, this is a complex task, which can take several hours including the validation of the formalised deficiencies by performing detections on test and real systems. However, this is an effort that has only be done once. Once a deficiency has been formalised, it can be reused over and over again. Often, only small adaptations are necessary afterwards.

All in all, one or two days per system have to be allowed for the application of Archimetrix to a system of the approximate size of CoCoME. Given that the manual investigation of large systems is practically impossible, I conclude that Archimetrix can offer substantial support for the software architect in an acceptable time frame.

10.10. Level-II-Validation

In the context of this thesis, a level-I-validation of Archimetrix could be performed (i.e. we validated our conjecture about the influence of deficiency occurrences on the architecture reconstruction and about the feasibility of the Archimetrix process as formulated in the research questions of this thesis). The validation was carried out by two students and myself. All of us did participate in the development of Archimetrix and were therefore very familiar with the process and the prototypical implementation. The next logical step would be to perform a level-II-validation, i.e. a validation if the Archimetrix process can be reliably applied by trained professionals [BR08].

This section sketches a controlled experiment to evaluate the applicability of Archimetrix in practice. This experiment sketch is in line with the suggestions of Prechelt on the design of controlled experiments in software engineering [Pre01, p. 57ff]. Note that this section is not intended to provide a sound and complete prescription for a controlled experiment. It rather offers suggestions on the appropriate parameters for such an experiment.

Research question The main research question for a level-II-validation could look like this: Given

- a software system with a documented architecture
- which contains deficiency occurrences that influence the architecture reconstruction,

can a user apply the Archimetrix process and the research prototype in such a way, that he

- a) finds the deficiency occurrences,
- b) can remove the detected occurrences, either by applying an automatic removal strategy or by following a removal guide, and

- c) can reconstruct an architecture that is no longer influenced by deficiencies?

Participants Archimetrix is meant to be used by software architects. Therefore, the validation should be performed with participants that have an appropriate level of knowledge in this field, e.g. students with at least a bachelor's degree and a focus on software engineering. The participants have to be given a special training before the validation. First, the Archimetrix process has to be explained to them, i.e. the iterative cycle of architecture reconstruction, deficiency detection, and deficiency removal. Then, they have to be trained in the usage of the tool. Since, a catalogue with deficiency formalisations, removal strategies, and removal guide templates is provided to the participants, they do not need expertise in specifying these artefacts.

Experiment setup The participants are to be provided with the following resources:

- The source of the system that is to be analysed. The conceptual architecture of this system is known to the supervisors but it is not disclosed to the participants of the experiments. The system should contain a number of deficiency occurrences whose presence is also unknown to the participants.
- A catalogue with deficiency formalisations, removal strategies, and removal guide templates. The formalisations in this catalogue should match the deficiency occurrences in the system under analysis such that the participants are able to detect these deficiencies. The removal strategies and removal guide templates shall also be applicable to those deficiency occurrences.
- A computer with the research prototype of Archimetrix installed.

Tasks The participants should be given the task to apply the Archimetrix process to the provided software system. They should follow the process steps in the default order but are free to perform as many iterations of the process as they see fit. Their goal is to remove as many deficiencies as possible and reconstruct an architecture which is influenced as little as possible by deficiencies. The participants have to decide when they have accomplished this goal. The participants shall document which deficiency occurrences they detect during the process. They also shall document which deficiency occurrences they remove and how they do remove them.

Independent variables Several independent variables can be varied across different instances of this controlled experiment.

The first independent variable is the participant. It is automatically varied by having multiple people take part in the experiment.

Second, the system under analysis can be exchanged in different instances of the experiment. This allows to eliminate effects that occur due to some peculiar

property of a given system under analysis which might, for example, hinder the participant's understanding of the system.

Third, different deficiencies might have an effect on how well the Archimetric process can be applied. For example, participants might have difficulty in understanding certain deficiencies. Therefore, it might be beneficial to perform the experiment with different deficiency catalogues. This would, of course, necessitate an individual preparation of the system under analysis such that for every participant, it contains the deficiencies that are formalised in the given deficiency catalogue.

Dependent variables There is a number of dependent variables which can be measured in this validation.

First, the time that is needed by each individual participant to perform the given task should be measured. The measurement starts after the training, when all resources (as described above) are provided to the participants. The measurement ends when a participant states that he has finished the given task.

Second, the architecture that was eventually reconstructed by each participant has to be documented. The supervisors can then compare it to the known conceptual architecture.

Third, the number of detected deficiency occurrences as documented by the participant can be compared to the number of known deficiency occurrences in the system.

In addition, it should be measured if the participants think that Archimetric was helpful in accomplishing their task. The different parts of the prototype implementation should be graded, e.g. how helpful was the relevance analysis, how helpful was the deficiency detection, etc. This kind of feedback could be collected with questionnaires that the participants have to fill out after finishing the given task.

Internal and external validity In order to achieve a sufficient internal validity, the sketched experiment should be conducted with a large enough number of participants. This can smooth out the effects of different levels of previous knowledge, for example.

To achieve a high degree of external validity, the experiment would have to be conducted a number of times, varying the independent variables as described above. For example, this can mean repeating the experiment with different systems, with different deficiencies, and different participants.

10.11. Lessons Learned

This section summarises the results of the conducted cases studies and relates them to the validation questions from Section 10.3. It also discusses limitations that were discovered in the course of the validation.

VQ1: Are the deficiency formalisations sufficiently precise to detect actual deficiency occurrences (instead of false positives)? Especially, Case study 1

was useful in the validation of the deficiency formalisations. In the course of the validation, all of the formalised deficiencies were detected in at least one of case studies. Furthermore, the manual inspection of the detected occurrences showed that no false positives were reported, thereby allowing a positive answer to validation question VQ1.

It is of course possible that the current formalisation are *too* restrictive, i.e. that there is a number of false negatives in the systems of the case studies. These missed deficiency occurrences could be detected by introducing more additional elements into the formalisations (see Section 4.4). That way, the detected occurrences with a high structural accuracy would be very close to the formalisations. In contrast, detected occurrences with a lower structural accuracy could indicate variants of the formalised deficiency.

VQ2: Do the defined design deficiencies occur in real-life systems, even if the systems were developed in a strictly component-based way? Case studies 2 and 3 show that the example deficiencies used in this thesis do indeed occur in real-life software systems. Since the case studies were specifically selected to be representatives of typical business information systems, it stands to reason that similar deficiencies will occur in other systems as well.

However, even if Palladio Fileshare and CoCoME are intended to be realistic representatives of typical business information systems, they are still academic examples. Especially, they are much smaller than the systems used in practice. While the largest case study system, the SOFA implementation of CoCoME, contained more than 10,000 lines of code, realistic business information system can be more than 100 times larger. The occurrence of deficiencies in the case study systems, in the other hand, suggests that the observed effects will also be present in larger systems. If small teams in small systems introduce deficiencies in spite of a very clear design documentation, it stands to reason that this problem will be much more prevalent in large systems with more developers and maybe also worse documentation.

VQ3: Is the calculated component relevance value a good indicator of components in which the detection of design deficiencies is worthwhile? The results for VQ3 are mixed. In some cases, the component relevance analysis reliably identified components which contained deficiencies. In other cases, components which contained deficiencies received a low ranking value. Looking at the two relevance strategies, the reason for these observations can be identified. The Complexity strategy is based on the widely recognised fact that complex systems tend to contain more defects than simpler systems [Rie96, BB01, Gla03]. However, this does not guarantee the presence of deficiencies in complex components. It rather stresses the heuristic nature of the employed relevance strategy.

The Closeness to Threshold strategy, on the other hand, is not related to the likelihood of deficiencies in the analysed components. Instead it deals with the change-proneness of the reconstructed architecture.

In order to align the component relevance analysis more with the intuitive expectation that “relevant components should contain deficiencies”, the selected

heuristics may need to be adapted. The relevance analysis could also be extended with additional relevance strategies. The intentionally extensible design of the relevance analysis should facilitate future work in this area.

VQ4: How does the limitation of scope by the relevance analysis improve the deficiency detection compared to pure pattern matching? The performance gain from focusing the deficiency detection on previously reconstructed components was significant in most cases, most notably for the SOFA implementation of CoCoME for which the deficiency detection was sped up by a factor of 16. Due to the non-polynomial complexity of the underlying pattern detection algorithms, the difference will probably be even larger for systems with hundreds of thousands lines of code.

Focusing the detection on single components is not only useful for the detection of deficiencies. It could also be leveraged for the detection of other patterns, e.g. design patterns, in the future. The required architectural model could either come from an architecture reconstruction or a manually created architecture model can be used.

One thing to keep in mind, however, is that focusing the pattern detection on one component after the other might lead to overlooking patterns which exist *between* those components.

VQ5: Does the removal of the deficiencies that receive a high ranking value lead to architectural changes, and does the removal of deficiencies with a low ranking value leave the architecture unchanged, i.e. do the deficiency ranking heuristics work? For the Store Example and for the reference implementation of CoCoME, the removal of deficiency occurrences led to architectural changes. These changes could be observed for those deficiency occurrences that had received the highest ranking values. The removal of lower ranked deficiency occurrences left the architecture unchanged. This indicates that the deficiency ranking heuristics work.

However, it has to be stressed that the ranking metrics are for the most part deficiency-specific. If different deficiencies are formalised, new ranking metrics have to be devised that have to be evaluated separately.

VQ6: Is the recovered architecture after the removal of a relevant deficiency occurrence closer to the documented architecture? The Store Example and the reference implementation of CoCoME show that the reconstructed architecture may indeed change when deficiency occurrences are removed. In those cases, the newly reconstructed architecture was indeed closer to the documented architecture.

However, not every deficiency removal influenced the architecture reconstruction. Therefore, the answer to this validation question is based on a relatively small amount of data. More experiments on other systems should be conducted here in order to better understand the extent to which deficiency occurrences influence the reconstruction algorithm.

Further observations A problem with the removal of design deficiencies is that if one occurrence has been removed, others may be invalidated. For example, in CoCoME all detected *Interface Violation* occurrences are concerned with the interface `PersistenceContext`. As soon as one of the occurrences is removed by extending that interface, all other occurrences can be removed more easily (by using the new interface). The different deficiency occurrences are related to one another. Thus, it seems sensible to add support to remove such groups of related design deficiency occurrences altogether at the same time.

This observation could also be considered in the deficiency ranking. An additional ranking metric could measure if an interface is bypassed several times (maybe even in the same class). The removal of these deficiencies which are involved such a case may be more critical than the removal of an *Interface Violation* which is the only one to bypass a certain interface.

11. Conclusion

This chapter presents the conclusions that can be drawn from this thesis. It also points out future research challenges which can be derived from the results of this thesis.

11.1. Results and Conclusions

In this thesis, I addressed five research questions. Here, I answer them with respect to the results that were obtained in the validation of Archimatrix.

RQ1 Do design deficiencies that stem from the neglect of design principles influence architecture reconstruction techniques?

This thesis provided answers for this question in two ways. On the one hand, I used an analytic approach which documented the relationship between base metrics and the decision to reconstruct components in SoMoX. Then, I showed how the occurrence of deficiencies can influence the base metrics and therefore the architecture reconstruction as a whole.

On the other hand, these theoretical considerations were supported by the case studies carried out in the validation. As documented in Chapter 10, there were cases in which the removal of a deficiency occurrence lead to a change in the automatically reconstructed architecture. Therefore, it can be concluded that there are situations in which design deficiencies indeed influence the architecture reconstruction of component-based systems.

RQ2 Can the integration of pattern detection techniques into the architecture reconstruction process help in detecting such an influence?

Archimatrix integrates pattern detection with an architecture reconstruction approach in order to detect deficiency occurrences. The deficiency detection is performed on the result model of the architecture reconstruction. In the case studies, I was able to detect occurrences of all four example deficiencies that were formalised in this thesis.

The detected occurrences were ranked with the help of heuristics to determine which of them were the most critical with respect to their influence on the architecture. The experiments showed that those deficiency occurrences whose removal led to a changed architecture were reliably identified as critical by the deficiency ranking. Therefore, pattern detection techniques provide great support in detecting deficiency occurrences which influence the architecture reconstruction.

RQ3 How can relevant design deficiencies be discovered, documented, and formalised?

The example deficiencies that were presented in this thesis were derived from component-oriented design principles in literature [ACM01, SGM02, Fow02]. The creation of a comprehensive deficiency catalogue was not in the focus of this thesis. This would also be impractical since many deficiencies are dependent on the system under analysis. Therefore, I presented a dedicated process for the discovery, documentation, and formalisation of design deficiencies. It addresses both, general as well as project- and company-specific deficiencies.

RQ4 How can detected deficiencies be removed and can the influence of the removal on the architecture be predicted?

In general, there are several ways to remove a given deficiency. How a given occurrence can be removed has to be decided by the software architect on a case-by-case basis. However, Archimetrix supports this decision in different ways. On the one hand, it offers automatic removal strategies which can remove deficiency occurrences without manual intervention. In this case, Archimetrix can also predict how the removal will affect the reconstructed architecture and present this prediction to the architect.

In cases where an automatic removal is not possible, Archimetrix offers the generation of individual removal guides for every deficiency. In cases in which deficiency occurrences are removed manually, no automatic architecture preview is possible. Instead a new architecture reconstruction has to be performed for the reengineered system. The results can then be compared to the original architecture manually.

RQ5 Can architecture reconstruction techniques be helpful in mitigating the scalability issues of pattern detection techniques?

The case studies show that the performance gain by focusing the deficiency detection on certain components can be quite significant. In one case, the detection time was reduced by the factor 16. In addition to improving the performance of the pattern detection, the integration of pattern detection and architecture reconstruction also enables the software architect to focus his analysis on certain parts of the system. However, the architect also has to be aware of the drawbacks of this technique. Analysing one component after the other may lead to overlooking deficiencies which occur between those components. Additionally, the architecture reconstruction may yield component which contain all system elements (see Section 10.8.2). In these cases, there is obviously no performance gain. The software architect has to keep this in mind and should choose components of a lower complexity.

Overall, this thesis provided evidence that Archimetrix can support a software architect in reconstructing a high-quality software architecture by detecting and removing deficiencies. Thereby, it paves the way for further maintenance activities.

11.2. Future Research Challenges

The results of this thesis lead to several possibilities for future research. This section highlights a few of these research challenges. Minor possibilities for improvements of the different steps of the Archimetrix process are pointed out in the limitations sections of the previous chapters.

Consideration of architectural knowledge One of the basic assumptions of this thesis is that no architectural documentation of the software system under analysis is available (cf. Section 2.8). However, in many cases, the software architect may be in the possession of such knowledge. There may be informal documentation ("box-and-line diagrams") or pieces of information ("Classes A and B belong together.") from the original development, even though no formal models of the architecture exist. In other cases the architect may have an architecture model from a previous application of SoMoX or Archimetrix.

Furthermore, Archimetrix does only analyse architectural information which can be extracted from the source code. However, component frameworks often also use additional information, e.g. code annotations and deployment descriptors in Enterprise Java Beans [EJB12]. This metadata could also be taken into account when reconstructing the architecture.

At the moment, Archimetrix cannot consider such information in its analyses. At the moment of writing, the integration of architectural information, especially in the architecture reconstruction, is under investigation in a master's thesis [Str13].

Consideration of run-time information The analyses of run-time information in the pattern detection process is only briefly discussed in this thesis (cf. Section 7.3). However, run-time information can also be very useful in the detection of design deficiencies like Performance Antipatterns [SW03, CMRT10]. The investigation of this research challenge is part of the upcoming research project *CloudScale* [Clo12]. In this project, Archimetrix is to be reused and extended for the detection of such run-time-dependent patterns.

Consideration of design patterns At the moment, Archimetrix uses pattern detection techniques exclusively for the detection of design deficiencies. The same techniques could also be used for the detection of design patterns. (In fact, Reclipse has been originally developed for the detection of design patterns [vDMT10b, vDT10].) Some design patterns, e.g. the *Facade* pattern [GHJV95], implement architectural concepts and therefore contain relevant clues for the architecture reconstruction. Bauer and Trifu try to improve their architecture reconstruction approach this way [BT04a, BT04b] (see Section 2.4). It should be investigated if similar techniques can be integrated into Archimetrix.

Integration of architecture quality analyses At the moment, Archimetrix does not tell the software architect 'when to stop', i.e. when a sufficiently good

architecture quality level has been achieved. Instead the software architect has to decide this by himself (see Section 4.5). This situation could be improved by identifying sensible architecture quality metrics, e.g. the modularisation quality of the system [SKR08], and integrating them into the process. These metrics could be measured after every iteration of the Archimetrix process to indicate when a sufficient level of modularisation is achieved. It might also be worthwhile to integrate support for the software architecture analysis method (SAAM) [KBWA94] into the Archimetrix process.

Analysis of heterogeneous code bases Due to the use of the parser SISSy, Archimetrix can analyse systems that are written in Java, C++, or Delphi. However, the complete system under analysis has to be written in one of these languages. Typical business information systems are often implemented in different languages. In order to enable a comprehensive analysis of such heterogeneous code bases, a consistent GAST would have to be constructed from the different system parts. This extension would be especially useful if Archimetrix were to be applied in an industrial context.

A. Meta Models

This chapter shows and explains the three most important meta models used in this thesis. The source code of the systems that are analysed with Archimetric is parsed into instances of the GAST meta model by the parser SISSy (Section A.1). Instances of the Service Architecture Meta Model (SAMM) presented in Section A.2 are the result models of the architecture reconstruction with SoMoX. The Source Code Decorator model relates elements from GAST and SAMM to each other and is the starting point for the design deficiency detection. It is explained in Section A.3. The description of the classes is adapted from [Tra11].

A.1. Generalised Abstract Syntax Tree

The type graph used in the examples in this thesis describes the structure of an abstract syntax tree. It is based on the generalised abstract syntax tree (GAST) meta model developed in the QBech project [QBe06]. The GAST was developed to provide a unified syntax tree model for different programming languages like Java, C, and C++. Figure A.1 shows an excerpt of that meta model. Especially some specialised sub classes have been omitted for clarity reasons.

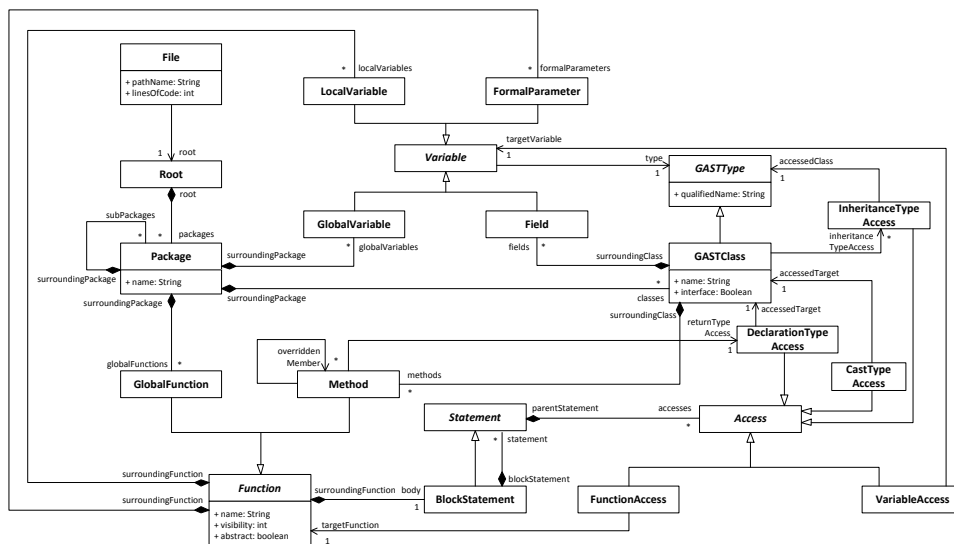


Figure A.1.: Meta model of the generalised abstract syntax tree (GAST, adapted from [Tra11])

Root The Root element is the central element of every GAST model. All other elements are reachable from the Root node via composition relations.

File Elements of the GAST, e.g. classes and packages, can be related to files in the file system. A File element holds references to those classes and packages and a String containing the path to the file.

Package Similar to packages in Java, the Package element provides name spaces and visibilities. A Package element can contain other packages, classes, global variables, and functions.

GASTType The GASTType element represents data types like primitive data types and classes. The attribute `qualifiedName` contains the unique, fully qualified name of the type. GASTType is abstract.

GASTClass Classes and Interfaces are represented by the element GASTClass in the GAST and are a sub type of the GASTType. Whether the GASTClass represents an actual concrete class or an interface is determined by the attribute `interface`. A GASTClass holds references to its methods, attributes, and inner classes. A GASTClass can be assigned to a Package.

Function Function is the abstract super type for all executable operations. In addition to attributes for its name, visibility and abstractness, a Function can have a number of local variables and formal parameters. The return type of a Function is determined by its `DeclarationTypeAccess`, a sub class of `Access`. A Function always contains a block statement which, in turn, can contain other statements.

GlobalFunction A GlobalFunction element represents a globally accessible operation, i.e. an operation that does not belong to a class. It can be assigned to a name space defined by a package. For example, C functions are represented by GlobalFunctions.

Method Functions that belong to a class are represented by Method elements, a sub type of Function. They can reference other methods which they override by means of the `overriddenMember` association.

Variable Variable is a super type for all kinds of variables. A Variable always has a name and a type. Variable is abstract.

LocalVariable LocalVariables are variables that are contained in a Function.

FormalParameter FormalParameters are variables that represent the parameters of a Function.

GlobalVariable GlobalVariables are variables that are globally accessible within a given scope. The scope is determined by the Package in which the GlobalVariable is contained.

Field The Field element represents class variables. Therefore, it is contained in a GASTClass.

Statement A Function consists of a number of Statements. There are multiple sub classes of Statement which represent the different kinds of statements. Most of them are omitted here. A Statement can contain a number of Accesses. Statement is abstract.

BlockStatement The BlockStatement is a special kind of Statement which can contain other Statements. It is the root element of all Statements contained within a Function.

Access An Access represents the use of a Variable or a Function. It always belongs to a certain Statement.

FunctionAccess A FunctionAccess represents the use of a Function in a Statement and therefore references the accessed Function element.

VariableAccess A VariableAccess represents the use of a Variable in a Statement and therefore references the accessed Variable element.

DeclarationTypeAccess The return type of a method can be specified by a DeclarationTypeAccess. It points to the GASTClass which specifies the return type.

CastTypeAccess A CastTypeAccess specifies the target type of a type cast. It can occur, for example, in a Statement.

InheritanceTypeAccess An InheritanceTypeAccess is used to reflect the inheritance relationships of a GASTClass. A class can have arbitrarily many InheritanceTypeAccesses but each of these accesses has only one target GAST-Type.

A.2. Service Architecture Meta Model

The Service Architecture Meta Model (SAMB) is a meta model for the specification of component-based systems. It was developed in the course of the Q-ImPRESS project [BBB⁺08] and is mainly based on the Palladio Component Model [RBB⁺11]. The architecture reconstruction with SoMoX creates an instance of the SAMB to represent the reconstructed system architecture.

As only the structural aspects of the architecture are relevant in this thesis, Figure A.2 only shows the elements that model a system's static structure. In general, the SAMB also contains elements for the representation of aspects like component behaviour, allocation, and resources.

Repository The Repository is the central element of the SAMB. It references all ComponentTypes, Interfaces, MessageTypes, and (data) Types of a SAMB instance. Therefore, the Repository entity represents a storage for first class entities that could be reused and composed into more complex architectures.

and a set of required ports. It also forms a common super class for `PrimitiveComponent` and `CompositeComponent`.

PrimitiveComponent This entity specifies a basic component on the lowest hierarchical level. A `PrimitiveComponent` has no subcomponents.

CompositeComponent The `CompositeComponent` entity is used to model hierarchical components, i.e. the components are allowed to be nested, thus forming a tree-like structure.

SubcomponentInstance This entity specifies a subcomponent of a `Composite Structure`. Note that this is not a run-time instance of a component, it is an instance from the architectural point of view, i.e. an instance of a `Component Type` in a `Composite Structure`.

EndPoint This abstract class represents a part of a `Connector` attachable to a `ComponentPort` to form a (communication) connection between components. It is subclassed by `ComponentEndpoint` and `SubcomponentEndpoint` to realise the assembly and delegation connection, respectively.

ComponentEndpoint The `ComponentEndpoint` entity represents the externally visible part of a `Connector` that is to be attached to a `ComponentPort` thus establishing a connection among components.

SubcomponentEndpoint This entity represents the externally visible part of a `Connector` that is to be attached to a port of a subcomponent thus establishing a communication link between the subcomponent and its super component.

Port This abstract superclass represents a general `Port` notion. In the SAMM, a `Port` can either be an `EventPort` or an `InterfacePort`. In this thesis, only the `InterfacePorts` are of interest.

InterfacePort An `InterfacePort` represents an instance of an `Interface` which allows the invocation of the functionality of a component via method calls.

Interface The `Interface` class represents the `Interface` concept of a SOA. It is the visible part of a component to which connector endpoints can be attached and which represents the access point to the component's functionality.

Operation This class represents an `Operation` signature, i.e. types of input and output messages and exceptions that can be thrown by an operation. It is similar to a WSDL `Operation`.

MessageType This entity represents the `MessageType` used for data communication. It is modelled according to the WSDL `Message` and it is of either the input (method parameters) or the output (return value) kind.

Parameter This class represents the data transmitted within messages, either as input or as output of the associated `Operation`.

Type This entity represents the data type for message `Parameters`.

A.3. Source Code Decorator

The source code decorator (SCD) serves as a bridge between the GAST and the SAM. It links the elements from both models, e.g. to show which classes are contained in a given component. Figure A.3 shows the relevant elements from the SCD meta model together with the elements from GAST and SAM connected by them.

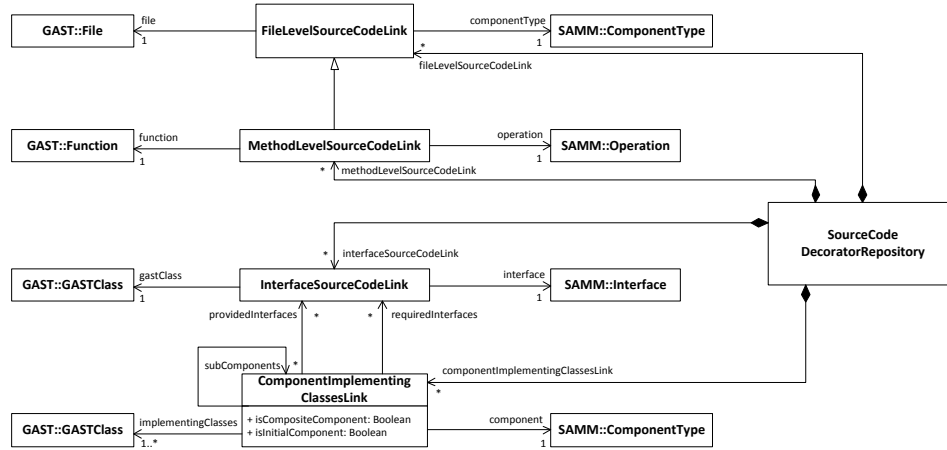


Figure A.3.: Meta model of the Source Code Decorator (SCD, adapted from [Tra11])

SourceCodeDecoratorRepository The SourceCodeDecoratorRepository is the central element of the SCD. It contains all the different links.

FileLevelSourceCodeLink The FileLevelSourceCodeLink connects Files from the GAST with ComponentTypes from the SAM.

MethodLevelSourceCodeLink A MethodLevelSourceCodeLink denotes which Functions from the GAST realise which Operations in the SAM.

InterfaceSourceCodeLink The InterfaceSourceCodeLink links component Interfaces from the SAM to the GASTClasses in which these interfaces are declared. (GASTClass is used to represent both, actual classes and interfaces, see Section A.1.)

ComponentImplementingClassesLink The ComponentImplementingClassesLink relates SAM Components to the GASTClasses contained therein. Each Component contains one to many GASTClasses. A Component can be an initial component or a composite component which is indicated by the respective boolean flags of the ComponentImplementingClassesLink. If a Component is a composite component, its subcomponents can be reached by the subComponents association. In addition, the Component's Interfaces can be referenced by navigating to the related InterfaceSourceCodeLinks via the associations providedInterfaces and requiredInterfaces.

B. Design Deficiencies

In this chapter, I present the various design deficiencies that I discovered in the course of this work. These are an addition to the *Transfer Object Ignorance* deficiency that serves as a running example in this thesis. All the deficiencies were also used in the validation of the approach described in Chapter 10.

For the presentation of the deficiencies, I adapted the Mini-AntiPattern Template proposed by Brown et al. [BMMM98]. For each deficiency, its name, the problem, and an example are presented. I also document one or more removal strategies, the deficiency's influence on the architecture reconstruction, and the deficiency formalisation.

B.1. Interface Violation

An *Interface Violation* represents a method call between components although the called method is not provided in an interface. This section briefly describes the problem with this deficiency and presents a small example. For this deficiency, I also present a more complex automated removal strategy than for the running example in Chapter 9. This removal strategy consists of three different Story Diagrams which call each other. It shows that the formalisation of complex removal strategies with Story Diagrams is feasible.

B.1.1. Design Deficiency Problem

Components in a good component-oriented architecture are supposed to communicate exclusively via their interfaces [SGM02, p. 30]. Thus, if two classes are part of different components, they can only invoke operations of each other which are provided by their counterpart's interfaces. This leads to a good decoupling of the components. Only classes which reside in the same component can directly access each other's public operations and attributes which, of course, results in a high coupling. If classes from different components invoke operations which are not part of their corresponding interface, this is called an interface violation.

An existing interface violation has a direct impact on the architecture reconstruction. Two classes which communicate with each other in a way that violates their interfaces are strongly coupled. Hence, they are probably assigned to the same component by a clustering-based reverse engineering approach.

B.1.2. Example

As an example, consider the example code in Figure B.1. `StoreQuery` implements the interface `IStoreQuery` and also contains a method `getInventory` which is *not* part

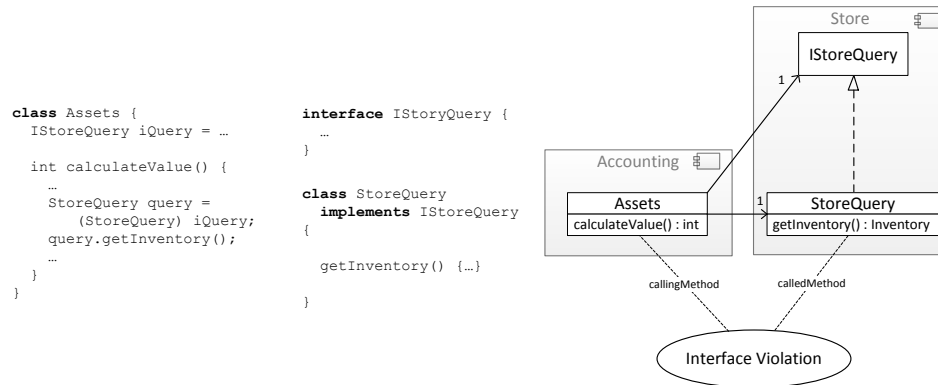


Figure B.1.: Example occurrence of an *Interface Violation* deficiency

of the interface. However, the class `Assets` needs the functionality of `getInventory` in its `calculateValue` method. Therefore, the programmer chose to downcast the reference `iQuery` to an object of the type `IStoreQuery` to the concrete type `StoreQuery`. After the downcast, the method `getInventory` is accessible because it is public.

This little 'trick' on part of the programmer leads to the situation depicted on the right of Figure B.1. `Assets` now has direct references to both, `IStoreQuery` and `StoreQuery`. Because the classes belong to different components, this constitutes an *Interface Violation*.

B.1.3. Influence on the Architecture Reconstruction

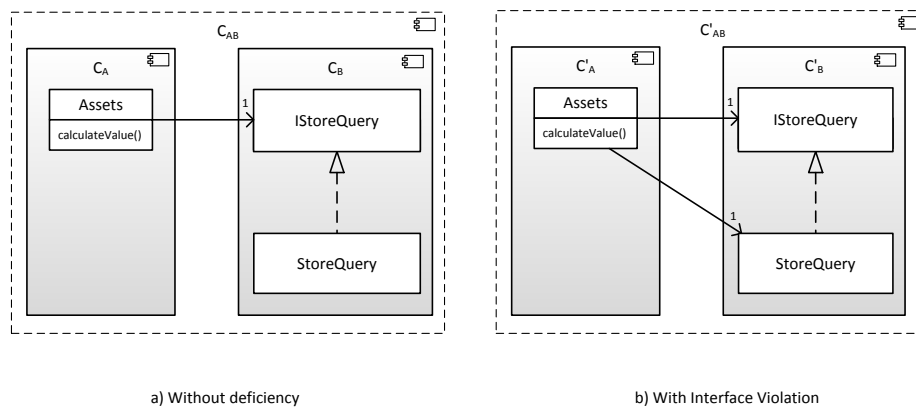


Figure B.2.: Example of the influence of an *Interface Violation* deficiency on the architecture recovery

Figure B.2 illustrates the influence of an occurrence of the *Interface Violation* deficiency on the architecture reconstruction. To the left of the figure, a situation without the deficiency is depicted while a situation with an *Interface Violation* is depicted on the right. The presence of an *Interface Violation* occurrence adds a direct reference between the classes *Assets* and *StoreQuery*. This is similar to the influence of a *Transfer Object Ignorance* occurrence which was analysed in Section 5.3. This means that the metric values for External Accesses Count(C_B), Internal Accesses Count (C_B, C_A), Afferent Coupling(C_A), Efferent Coupling(C_B), Coupling(C_B, C_A), and Adherence To Interface Communication(C_B, C_A) will be directly affected by this deficiency.

This leads to a similar conclusion: If *Assets* only communicated with *StoreQuery* via the *IStoreQuery* interface, *Assets* might be clustered into a different component than *StoreQuery*. However, if *Assets* accesses the method `getInventory()` of *StoreQuery* which is not provided by *IStoreQuery*, it bypasses the interface. This leads to a direct coupling between *Assets* and *StoreQuery*, which consequently may be grouped together into the same component, even if they do not belong together conceptually.

B.1.4. Removal Strategies

In order to remove an *Interface Violation* occurrence, the interface *IStoreQuery* could be extended and the method call in `calculateValue()` can be modified such that the method provided by the interface is called and the class *StoreQuery* is not accessed directly. As a consequence, all other classes that implement *IStoreQuery* have to be adapted, too.

A formalisation of this removal strategy is shown in the story diagram in Figure B.3. The story diagram and its two auxiliary diagrams are explained in detail in the following paragraphs. The syntax and semantics of story diagrams can be found in [vDHP⁺12].

The story diagram has four in-parameters: `call`, `interface`, `castStmt`, and `accessedMethodOwner`. `call` represents the interface violation, i.e. the statement that calls the method in the concrete class (the call of `m3` in `m1`). `interface` is the interface that will be extended (`IB` in the example). `castStmt` refers to the statement that down-casts the interface type to the concrete class type (i.e. the statement `B b = (B) ib;`). Finally, `accessedMethodOwner` is the class that contains the called method (`B` in the example). The story diagram has no out parameter.

The first story node (after the initial node) checks if a method with the same name as the `accessedMethod` already exists in the interface. This is accomplished by matching all methods of the interface in the set object `interfaceMethods`. Then the pattern constraint ensures that none of these methods has the same name as the parameter `method`. If this is not the case, i.e. if a method of the name in question already exists in the interface, the application of the story diagram fails. Otherwise, the control flow continues via the activity edge labelled with `[success]`.

The second story node creates a method declaration in the interface (`methodDecl`). This new method declaration is declared as `public` (attribute assignment `visibility := PUBLIC`) and `abstract` (attribute assignment `abstract := true`). The declara-

tion receives the same name as the formerly called method (attribute assignment `name := method.name`, `m3` in the example). The new method declaration is added to the methods of the interface by creating a method link between interface and `methodDecl`. It is also added to the previously matched set `interfaceMethods` by creating a corresponding inclusion link. The target accessed by the call is changed by deleting the link between `call` and `method` and recreating it between `call` and `methodDecl`. The return type of the method is set by creating a new object `typeAccessNew` of the type `DeclarationTypeAccess` and connecting it to `methodDecl`. It points to the same `GASTType` as the old declaration type access of the method.

The next node is an activity call node. It calls the story diagram `copyParameters` which is described in detail below. This story diagram is responsible for copying all the parameters of the formerly called method to the newly created declaration `methodDecl`.

The following story node contains only the two bound, mandatory object variables `castStmt` and `call`. Its responsibility is to try and match the link accesses between those object variables. If the link exists, the cast and the call are part of the same statement. In that case, the matching of the story node is successful and the control flow continues along the transition labelled with *[success]* to story node 4a. If the matching fails, i.e. the link does not exist and the cast and the call are therefore not part of the same statement, the story node is left via the *[failure]* transition. This distinction is necessary because the effort to remove the cast statement is much greater if the cast is not done in the same statement as the call (compare story nodes 4a and 4b).

If the cast is in the same statement as the call, story node 4a is executed: The `castStmt` and its access to `B` are deleted. If the cast is not in the same statement as the call that means that the cast is executed at some point before the call and the resulting down-cast object is stored in a temporary variable. In this case, this temporary variable can be deleted along with the accesses to it from the call and the cast statements. Instead, a new variable of the interface type (`IB` in the example) is created and then accessed by the call statement (4b). In both cases, activity node 5 is executed next.

Activity node 5 is responsible for adapting all other classes that implement the now changed interface. Thus, the node is a for-each activity node that binds a class which is connected to the interface in each iteration. For each of those bindings, the node that is reachable via the *[each time]* edge is executed. In this case, that is a story diagram call of the story diagram `generateMethodStub` which is explained in the following section. In contrast to the previous call to `copyParameters`, the called story diagram here can either succeed or fail. If the generation of the method stub fails, the application of the calling diagram is also aborted with a failure.

When no new classes implementing the interface can be found, i.e. method stubs have been generated for all implementing classes, the story diagram terminates at the success final node.

The story diagram `copyParameters` (see Figure B.4) copies all the parameters from a `sourceMethod` to a `targetMethod`. Both methods are provided as parameters. The diagram consists of two story nodes.

The first activity node is a for-each activity node. It successively binds all

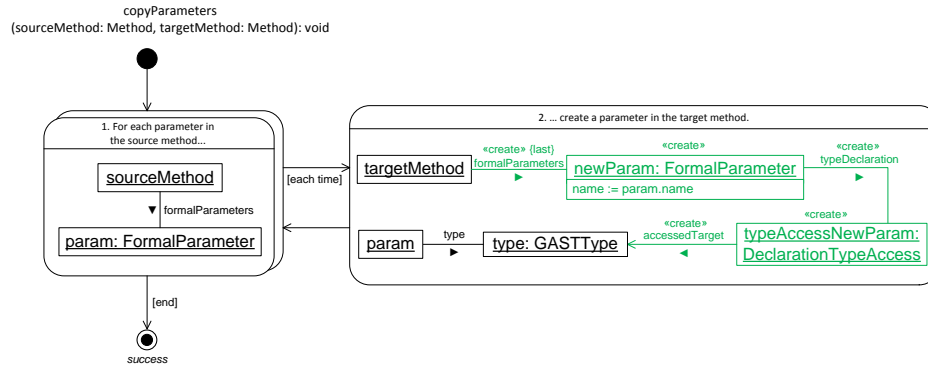


Figure B.4.: Auxiliary story diagram for the called by the removal strategy in Figure B.3

formal parameters of the given `sourceMethod` to the object variable `param`. Each time a new parameter is bound, the second node is executed. There, a new formal parameter `newParam` is created in the `targetMethod`. Its name is set to the same name as the original parameter's by the expression `'name := param.name'`. The type is also set accordingly by binding the type of `param`. Then, a new access to that type is created and connected to `newParam`. The `newParam` is inserted at the last position in the list of parameters as indicated by the link position constraint `{last}` at the new `formalParameters` link.

The story diagram `generateMethodStub` is shown in Figure B.5. It creates a method which implements a method `methodDecl` from an interface. This is accomplished by two story nodes and one story diagram call. The first node checks if the given class contains a method with the same name as the given declaration `methodDecl`. The check is performed by the attribute constraint `'name = methodDecl.name'`. Since the object variable `method` is negative (crossed-out), the matching of this story node is considered successful if *no* such method exists in the class. In that case the next story node is executed. If a method of the name in question already exists, the execution of the first story node fails and the story diagram terminates at the failure final node.

The second story node creates a new `methodStub` in the given class. The visibility of this method is set to public and its name is set to the name of the method declaration as signified by the expression `'name := methodDecl.name'`. The correct return type for the method is set by creating a `newTypeAccess` from the `methodStub` to the `returnType` that is also accessed by the `methodDecl`.

Finally, the story diagram `CopyParameters` is called in the story diagram call node. The `methodDecl` and the `methodStub` are passed as parameters. The called diagram then copies all parameters from the given method to the newly created `methodStub` as explained above.

Another possibility to remove an *Interface Violation* is a variant of the above removal strategy: Instead of extending the interface `IStoreQuery`, a completely new interface could be introduced and accessed instead of the class `StoreQuery`.

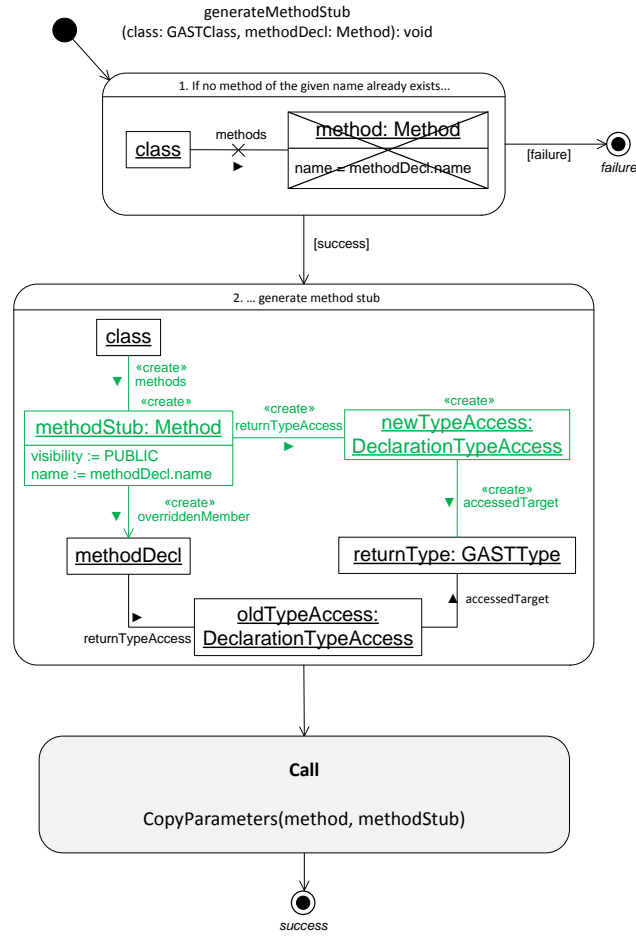
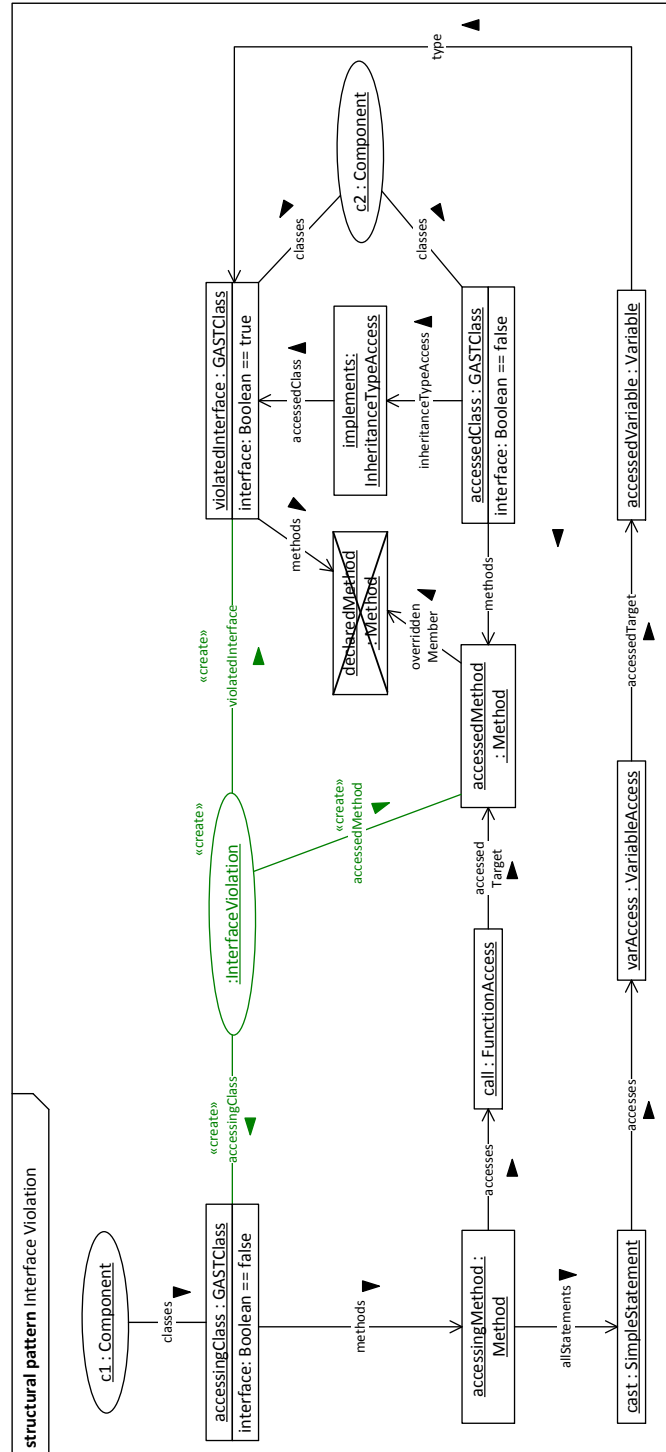


Figure B.5.: Auxiliary story diagram for the called by the removal strategy in Figure B.3

This way, other classes would not have to be adapted.

B.1.5. Formalisation

Figure B.6 shows a possible formalisation of the *Interface Violation* deficiency. The formalisation contains two **Component** sub patterns c1 and c2. c1 contains the **accessingClass** (Assets in the example) which may not be an interface. The **accessingClass** has an **accessingMethod** which contains two things: A call and a cast statement. The cast accesses a variable of the type of the **violatedInterface**. The call invokes the **accessedMethod** in the **accessedClass**. The **accessedClass** implements the **violatedInterface** and both are contained in component c2. In addition, the **accessedMethod** may not override a **declaredMethod** which is part of the **violatedInterface**.

Figure B.6.: Structural formalisation of the *Interface Violation* deficiency

B.2. Unauthorised Call

This section presents the *Unauthorised Call* deficiency which is a call of a method which is provided in the interface of the called component but which is not allowed because the calling component does not require the corresponding interface. The section briefly introduces the underlying problem, shows an example and discusses two removal strategies. I also describe the influence of this deficiency on the architecture reconstruction and explain a possible formalisation.

B.2.1. Design Deficiency Problem

In contrast to an *Interface Violation*, an *Unauthorised Call* is an invocation of an operation which is provided by an interface but which may not be called because the calling component is not connected to the called component. In programming languages which do not explicitly support the concept of components, e.g. plain Java, it is easy to introduce *Unauthorised Calls* accidentally because strict interface communication is not enforced by the language. This is especially true for components that are connected by some interface. Developers may only remember that those components are connected somehow and might then introduce the *Unauthorised Call*.

B.2.2. Example

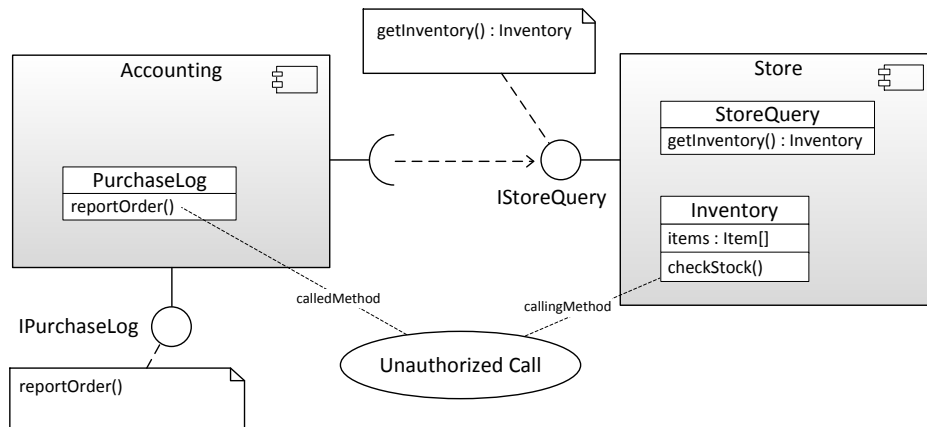


Figure B.7.: Example occurrence of an *Unauthorised Call* deficiency

In Figure B.7, the components **Accounting** and **Store** are connected via the interface **IStoreQuery**. Thus, the method `getInventory()` may be called by classes in the **Accounting** component. The **Accounting** component, on the other hand, provides the `reportOrder()` method on its interface **IPurchaseLog**. However, this

does not allow classes in the Store component, e.g. Inventory, to call `reportOrder()` because the Store component does not require that interface.

Developers may be tempted to call the method in `PurchaseLog` directly without using the appropriate interface `IPurchaseLog`, especially because the components are already connected. In the absence of component frameworks which prevent this, the deficiency may be introduced quickly.

B.2.3. Influence on the Architecture Reconstruction

Similar to an *Interface Violation*, an *Unauthorised Call* adds one direct reference between the two components that are involved in the deficiency. Therefore, adding this deficiency to a system will have a comparable influence on the metric values during the architecture reconstruction, i.e. the counting and coupling metrics will be directly affected. The influence on the naming-related metrics, like Name Resemblance and Package Mapping, depends on the concrete deficiency occurrence and cannot be analysed in the generic case.

B.2.4. Removal Strategies

Essentially, the *Unauthorised Call* deficiency is a special kind of *Interface Violation*. However, it is easier to remove. The called method is already provided by an interface. So, an obvious removal strategy is to make the use of that interface explicit instead of calling a method in the implementing class.

If the use of the interface is not desired, for some reason, the call has to be removed and some other way to provide the functionality has to be conceived of. For example, the called method could be moved to the calling component. However, this would necessitate changes to the corresponding interface which might lead to complicated refactorings.

B.2.5. Formalisation

Figure B.8 shows a formalisation of the *Unauthorised Call* deficiency. It specifies a situation in which a `callingClass` contains an `accessingMethod` which calls another method, the `accessedMethod`. The `accessedMethod` is located in the `calledClass` which implements an interface. This interface contains a `declaredMethod` which is implemented by the `accessedMethod`. The component `comp` which contains the `callingClass`, however, does not require the interface. This is expressed by the *negative fragment* in the upper right corner of the formalisation. A negative fragment encapsulates an object structure which may not be present in the host graph for the pattern to be detected. In this case, there may not be an `InterfaceSourceCodeLink` which points to the interface and which is marked as an interface that is required by `comp`.

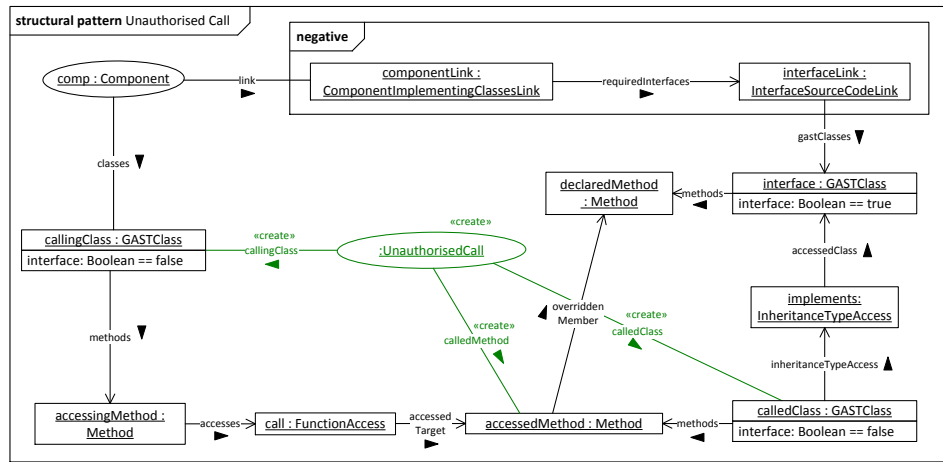


Figure B.8.: Structural formalisation of the *Unauthorised Call* deficiency

B.3. Inheritance between Components

The *Inheritance between Components* deficiency is slightly different from the other deficiencies presented in this thesis. In contrast to the other deficiencies, there is no real problem in the source code. Instead, the reconstructed architecture is not in line with the architectural guidelines by Szyperski [SGM02].

B.3.1. Design Deficiency Problem

Inheritance relations between classes result in dependencies in the implementation and thereby lead to a tight coupling between the super class and the sub class. If the super class changes, then the sub class has to be changed, too. According to the component definition by Szyperski, a “[...] component is a unit of independent deployment” [SGM02, p. 30]. Therefore, such a dependency must not exist between classes that are part of different components. If one class extends another class, they have to be assigned to the same component. Otherwise, the components cannot be reused independently.

B.3.2. Example

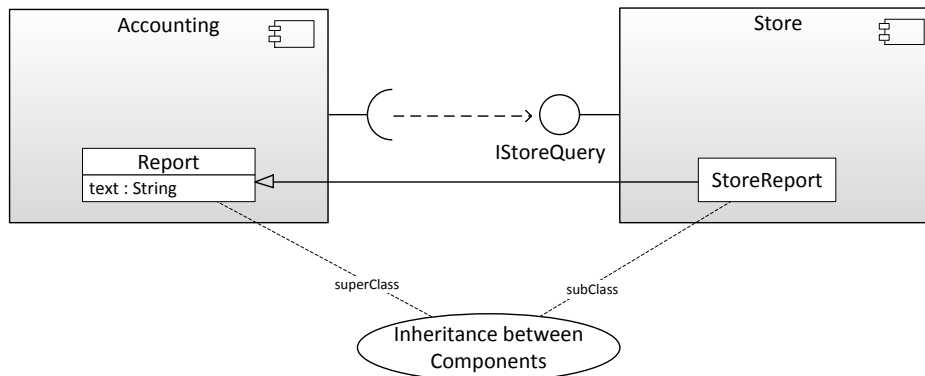


Figure B.9.: Example occurrence of an *Inheritance between Components* deficiency

In Figure B.9, the class `StoreReport` extends the class `Report`. However, `StoreReport` has been assigned to the `Store` component while `Report` is part of the `Accounting` component. Therefore, `Store` cannot be deployed independently from `Accounting` anymore.

B.3.3. Influence on the Architecture Reconstruction

The problem here is not that the original developer did make a mistake to introduce the deficiency. On the contrary, the architecture reconstruction heuristic

assigned the two classes to different components in spite of the inheritance relationship between them. This means that the inheritance relationship between the classes did not sufficiently influence the architecture reconstruction metrics to assign both classes to the same component.

B.3.4. Removal Strategies

In contrast to the other deficiencies presented in this thesis, there are no direct reengineering strategies for the *Inheritance between Components* deficiency. The placement of the classes in different components is a result of the automatic architecture reconstruction. Thus, something in the implementation of the system compelled the clustering algorithm to assign the classes to different components. However, it would be not be advisable to try and influence the architecture reconstruction indirectly, e.g. by adding additional references between the two classes.

Therefore, the *Inheritance between Components* deficiency is not so much a design deficiency of the system but it is a deficiency of the reconstructed architecture. Its occurrence can serve as a warning that the reconstruction may have made a mistake at that point and that the reconstructed components should be reviewed by the software architect.

In order to avoid this deficiency, a way to specify such architectural guidelines as an input for the architecture reconstruction would be needed. If such a mechanism was available, the software architect could specify that the architecture reconstruction may not assign two classes which are connected by an inheritance relation to two different components. Such an extension of SoMoX is currently under development in a master's thesis by Christian Stritzke [Str13].

B.3.5. Formalisation

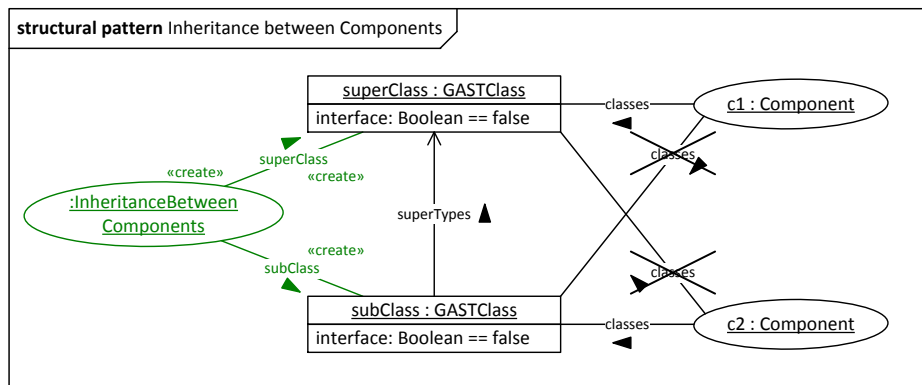


Figure B.10.: Structural formalisation of the *Inheritance Between Components* deficiency

Figure B.10 shows the formalisation of the *Inheritance between Components* deficiency which is rather straightforward. It describes two classes, `superClass` and `subClass`, where `superClass` is among the `superTypes` of `subClass`. `superClass` is contained in a component `c1` while `subClass` is part of a component `c2`. The mutual exclusion of a containment in both components implies that the two components are not composed in some way, e.g. `c1` is not contained within `c2` and vice versa.

C. Clustering Configurations

This section documents the configuration of SoMoX that were used in the validation (see Chapter 10). Each table lists the metric values and, where applicable, the configuration of the blacklist and additional filters. The clustering configuration for the Store Example is documented in Table C.1. Table C.2 shows the configuration used for the architecture reconstruction of Palladio Fileshare. Table C.3 contains the values for the reference implementation of CoCoME. Finally, the Table C.4 documents which weights and filters were used when analysing the SOFA implementaion of CoCoME.

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 Min Value	25
Clustering Composition Threshold Max Value	100
Clustering Composition Threshold Decrement	10
Merge: Interface Violation	10
Clustering Merge Threshold Min Value	45
Clustering Merge Threshold Max Value	100
Clustering Merge Threshold Increment	10
Blacklist	java.*

Table C.1.: Configuration used for the architecture reconstruction of the Store Example

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 Min Value	19
Clustering Composition Threshold Max Value	80
Clustering Composition Threshold Decrement	15
Merge: Interface Violation	10
Clustering Merge Threshold Min Value	41
Clustering Merge Threshold Max Value	100
Clustering Merge Threshold Increment	7
Blacklist	java.* de.uka.ipd.sdq.BySuite de.uka.ipd.sdq.✓ palladiofileshare.testdriver.*

Table C.2.: Configuration used for the architecture reconstruction of Palladio Fileshare

C. Clustering Configurations

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 Min Value	25
Clustering Composition Threshold Max Value	100
Clustering Composition Threshold Decrement	10
Merge: Interface Violation	10
Clustering Merge Threshold Min Value	45
Clustering Merge Threshold Max Value	100
Clustering Merge Threshold Increment	10
Blacklist	everything but org.cocome.*
Additional filter	.*TO .*Event

Table C.3.: Configuration used for the architecture reconstruction of the Co-CoME reference implementation

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 Min Value	25
Clustering Composition Threshold Max Value	100
Clustering Composition Threshold Decrement	10
Merge: Interface Violation	10
Clustering Merge Threshold Min Value	45
Clustering Merge Threshold Max Value	100
Clustering Merge Threshold Increment	10
Blacklist	java.*
	javax.*

Table C.4.: Configuration used for the architecture reconstruction of the SOFA implementation of CoCoME

D. List of Abbreviations

ADL	Architecture Description Language
AST	Abstract Syntax Tree
CBSE	Component-Based Software Engineering
CC	Composite Component
CoCoME	Common Component Modeling Example
COTS	Component Off-The-Shelf
DSL	Domain-Specific Language
DTO	Data Transfer Object
EMF	Eclipse Modeling Framework
GAST	Generalised Abstract Syntax Tree
PC	Primitive Component
POSA	Pattern-Oriented Software Architecture
SAM	Service Architecture Model
SAMM	Service Architecture Meta Model
SAR	Software Architecture Reconstruction
SCD	Source Code Decorator
SISSy	Structural Investigation of Software Systems
SoMoX	Software Model Extractor
TO	Transfer Object

References

- [ABJ⁺10] Arendt, Thorsten, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer: *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations*. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 121 – 135. Springer, 2010, ISBN 978-3-642-16145-2.
- [ACL05] Aversano, Lerina, Gerardo Canfora, and Andrea De Lucia: *Migrating Legacy Systems to the Web*. In Khosrow-Pour, Mehdi (editor): *Encyclopedia of Information Science and Technology (IV)*, pages 1949 – 1954. Idea Group, 2005, ISBN 978-1-59140-553-X.
- [ACM01] Alur, Deepak, John Crupi, and Dan Malks: *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001, ISBN 978-0-130-64884-1.
- [AFZ11] Arcelli Fontana, Francesca and Marco Zanoni: *A tool for design pattern detection and software architecture reconstruction*. *Information Sciences*, 181(7):1306 – 1324, 2011, ISSN 0020-0255.
- [AKM⁺11] Arendt, Thorsten, Sieglinde Kranz, Florian Mantz, Nikolaus Regnat, and Gabriele Taentzer: *Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support*. In *Proceedings of the Software Engineering 2011: Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 183 of *Lecture Notes in Informatics*, pages 63 – 74. Gesellschaft für Informatik, 2011.
- [All02] Allen, Eric: *Bug Patterns in Java*. Apress, 2002, ISBN 978-1-590-59061-4.
- [Arc12] *Archimetrix*, 2012. <http://www.fujaba.de/archimetrix>, visited on April 10th, 2012.
- [Art88] Arthur, Lowell J.: *Software Evolution - The Software Maintenance Challenge*. Wiley, 1988, ISBN 978-0-471-62871-9.
- [BB01] Boehm, Barry and Victor R. Basili: *Software Defect Reduction Top 10 List*. *IEEE Computer*, 34(1):135 – 137, 2001, ISSN 0018-9162.
- [BBB⁺08] Becker, Steffen, Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Lucia Kapová, Jan Kofroň, Heiko Koziol, Johan Kraft, Raffaella Mirandola, Johannes Stammel, Giordano Tamburrelli, and Mircea

- Trifu: *Service Architecture Meta-Model*. QImPrESS Project Deliverable D2.1, September 2008.
- [BBM96] Basili, Victor R., Lionel C. Briand, and Walcélio L. Melo: *A Validation of Object-Oriented Design Metrics as Quality Indicators*. IEEE Transactions on Software Engineering, 22(10):751 – 761, October 1996, ISSN 0098-5589.
- [BCL⁺06] Bruneton, Eric, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean Bernard Stefani: *The FRACTAL component model and its support in Java*. Software: Practice and Experience, 36(11/12):1257 – 1284, 2006, ISSN 1097-024X.
- [BCMV03] Bianchi, Alessandro, Danilo Caivano, Vittorio Marengo, and Giuseppe Visaggio: *Iterative Reengineering of Legacy Systems*. IEEE Transactions on Software Engineering, 29(3):225 – 241, March 2003.
- [Bec07] Beck, Kent: *Implementation Patterns*. Addison-Wesley, 2007, ISBN 978-0-321-41309-3.
- [BGH⁺08] Binkley, David, Nicolas Gold, Mark Harman, Zheng Li, Kiarash Mahdavi, and Joachim Wegener: *Dependence Anti Patterns*. In *Proceedings of the 4th International ERCIM Workshop on Software Evolution and Evolvability*, pages 25 – 34. IEEE, 2008, ISBN 978-1-4244-2776-5.
- [BHP06] Bureš, Tomáš, Petr Hnětynka, and František Plášil: *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*. In *Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications*, pages 40 – 48. IEEE, August 2006.
- [BHS07] Buschmann, Frank, Kevlin Henney, and Douglas C. Schmidt: *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Wiley, 2007, ISBN 978-0-470-05902-9.
- [BHT⁺10] Becker, Steffen, Michael Hauck, Mircea Trifu, Klaus Krogmann, and Jan Kofron: *Reverse Engineering Component Models for Quality Predictions*. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, CSMR, pages 199 – 202. IEEE, 2010.
- [BJ05] Basit, Hamid Abdul and Stan Jarzabek: *Detecting higher-level similarity patterns in programs*. SIGSOFT Software Engineering Notes, 30(5):156 – 165, September 2005, ISSN 0163-5948.
- [BK07] Bourqun, Fabrice and Rudolf K. Keller: *High-impact Refactoring Based on Architecture Violations*. In *Proceedings of the 11th Eu-*

-
- ropean Conference on Software Maintenance and Reengineering, CSMR, pages 149 – 158. IEEE, 2007, ISBN 978-0-7695-2802-3.
- [BKR09] Becker, Steffen, Heiko Koziol, and Ralf Reussner: *The Palladio component model for model-driven performance prediction*. Journal of Systems and Software, 82(1):3 – 22, 2009, ISSN 0164-1212.
- [BM06] Bayer, Joachim and Dirk Muthig: *A view-based approach for improving software documentation practices*. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 269 – 278. IEEE, March 2006.
- [BMMM98] Brown, William H., Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mombray: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998, ISBN 978-0-471-19713-0.
- [BMR⁺96] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996, ISBN 978-0-471-95869-7.
- [BMW93] Biggerstaff, Ted J., Bharat G. Mitbander, and Dallas Webster: *The concept assignment problem in program understanding*. In *Proceedings of the 15th International Conference on Software Engineering, ICSE*, pages 482 – 498. IEEE, 1993, ISBN 978-0-89791-588-7.
- [BPD12] Beck, Fabian, Alexander Pavel, and Stephan Diehl: *Interaktive Extraktion von Software-Komponenten*. Softwaretechnik-Trends, 32(2):47 – 48, May 2012.
- [BR08] Böhme, Rainer and Ralf Reussner: *Validation of Predictions with Measurements*. In *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science*, pages 14 – 18. Springer, 2008, ISBN 978-3-540-68946-1.
- [BT04a] Bauer, Markus and Mircea Trifu: *Architecture-Aware Adaptive Clustering of OO Systems*. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering, CSMR*, pages 3 – 14. IEEE, March 2004.
- [BT04b] Bauer, Markus and Mircea Trifu: *Combining Clustering with Pattern Matching for Architecture Recovery of OO Systems*. Softwaretechnik-Trends, 24(2), May 2004.
- [CC90] Chikofsky, Elliot J. and James H. Cross II: *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, 7(1):13 – 17, January 1990.

- [CCDJ10] Coello Coello, Carlos, Clarisse Dhaenens, and Laetitia Jourdan: *Multi-Objective Combinatorial Optimization: Problematic and Context*. In *Advances in Multi-Objective Nature Inspired Computing*, volume 272 of *Studies in Computational Intelligence*, pages 1 – 21. Springer, 2010, ISBN 978-3-642-11217-1.
- [Che11] Checkstyle, 2011. <http://eclipse-cs.sourceforge.net/>, visited on August 8th, 2012.
- [CK94] Chidamber, Shyam R. and Chris F. Kemerer: *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, 20(6):476 – 493, June 1994, ISSN 0098-5589.
- [CKK01] Cho, Eun Sook, Min Sun Kim, and Soo Dong Kim: *Component metrics to measure component quality*. In *8th Asia-Pacific Software Engineering Conference*, APSEC, pages 419 – 426. IEEE, December 2001.
- [CKK08] Chouambe, Landry, Benjamin Klatt, and Klaus Krogmann: *Reverse Engineering Software-Models of Component-Based Systems*. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, CSMR, pages 93 – 102, Athens, Greece, April 2008. IEEE, ISBN 978-1-4244-2157-2.
- [CKS07] Christl, Andreas, Rainer Koschke, and Margaret Anne Storey: *Automated Clustering to Support the Reflexion Method*. Information and Software Technology, 49(3):255 – 274, 2007, ISSN 0950-5849.
- [Clo12] CloudScale, 2012. <http://www.cloudscale-project.eu/>, visited on November 16th, 2012.
- [CMRT10] Cortellessa, Vittorio, Anne Martens, Ralf Reussner, and Catia Trubiani: *A Process to Effectively Identify Guilty Performance Antipatterns*. In *Proceedings of the Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 368 – 382. Springer, 2010.
- [CoC12] Common Component Modeling Example Website, 2012. <http://cocome.org/>, visited on September 24th, 2012.
- [DDL99] Demeyer, Serge, Stéphane Ducasse, and Michele Lanza: *A hybrid reverse engineering approach combining metrics and program visualisation*. In *Proceedings of the 6th Working Conference on Reverse Engineering*, WCRE, pages 175 – 186. IEEE, October 1999.
- [DDN03] Demeyer, Serge, Stéphane Ducasse, and Oscar Nierstrasz: *Object-Oriented Reengineering Patterns*. Morgan Kaufman Publishers, 2003, ISBN 978-1-55860-639-4.
- [DKG08] Denier, Simon, Foutse Khomh, and Yann Gaël Guéhéneuc: *Reverse-Engineering the Literature on Design Patterns and*

-
- Reverse-Engineering*. Technical Report EPM-RT-2008-09, DGIGL, École Polytechnique Montréal, October 2008.
- [dLDGR10] Lucia, Andrea de, Vincenzo Deufemia, Carmine Gravino, and Michele Risi: *Improving Behavioral Design Pattern Detection through Model Checking*. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, CSMR, pages 179 – 188. IEEE, 2010.
- [DMTS10] Dietrich, Jens, Catherine McCartin, Ewan Tempero, and Syed M. Ali Shah: *Barriers to Modularity - An Empirical Study to Assess the Potential for Modularisation of Java Programs*. In Heine-man, George T., Jan Kofron, and Frantisek Plasil (editors): *Research into Practice ? Reality and Gaps*, volume 6093 of *Lecture Notes in Computer Science*, pages 135 – 150. Springer, 2010, ISBN 978-3-642-13820-1.
- [DP09] Ducasse, Stéphane and Damien Pollet: *Software Architecture Reconstruction: A Process-Oriented Taxonomy*. IEEE Transactions on Software Engineering, 35(4):573 – 591, July/August 2009.
- [DYM⁺08] Dietrich, Jens, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow: *Cluster analysis of Java dependency graphs*. In *Proceedings of the 4th ACM symposium on Software Visualization*, SoftVis, pages 91 – 94. ACM, 2008, ISBN 978-1-60558-112-5.
- [DZP09] Dong, Jing, Yajing Zhao, and Tu Peng: *A Review of Design Pattern Mining Techniques*. International Journal of Software Engineering and Knowledge Engineering, 19(6):823 – 855, September 2009.
- [Ecl12] *Eclipse*, 2012. <http://eclipse.org/>, visited on August 7th, 2012.
- [EFH⁺11a] Erdmenger, Uwe, Andreas Fuhr, Axel Herget, Tassilo Horn, Uwe Kaiser, Volker Riediger, Werner Teppe, Marianne Theurer, Denis Uhlig, Andreas Winter, Christian Zillmann, and Yvonne Zimmermann: *SOAMIG Project: Model-Driven Software Migration towards Service-Oriented Architectures*. In *Joint Proceedings of the First International Workshop on Model-Driven Software Migration and Fifth International Workshop on Software Quality and Maintainability*, pages 15 – 16. CEUR-WS.org, March 2011.
- [EFH⁺11b] Erdmenger, Uwe, Andreas Fuhr, Axel Herget, Tassilo Horn, Uwe Kaiser, Volker Riediger, Werner Teppe, Marianne Theurer, Denis Uhlig, Andreas Winter, Christian Zillmann, and Yvonne Zimmermann: *The SOAMIG Process Model in Industrial Applications*. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, CSMR, pages 339 – 342. IEEE, 2011.

- [EJB12] *Java Enterprise Edition*, 2012. <http://www.oracle.com/technetwork/java/javaee/overview/index.html>, visited on October 10th, 2012.
- [Epp95] Eppstein, David: *Subgraph Isomorphism in Planar Graphs and Related Problems*. In *Proceedings of the 6th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 632 – 640. Society for Industrial and Applied Mathematics, 1995, ISBN 978-0-89871-349-8.
- [FH10] Frey, Sören and Wilhelm Hasselbring: *Model-Based Migration of Legacy Software Systems to Scalable and Resource-Efficient Cloud-Based Applications: The CloudMIG Approach*. In *Proceedings of the First International Conference on Cloud Computing, GRIDS and Virtualization*, pages 155 – 158. Xpert Publishing Services, November 2010, ISBN 978-1-61208-106-9.
- [FHR11a] Fuhr, Andreas, Tassilo Horn, and Volker Riediger: *An Integrated Tool Suite for Model-Driven Software Migration towards Service-Oriented Architectures*. *Softwaretechnik-Trends*, 31(2):8 – 9, May 2011.
- [FHR11b] Fuhr, Andreas, Tassilo Horn, and Volker Riediger: *Using Dynamic Analysis and Clustering for Implementing Services by Reusing Legacy Code*. In *Proceedings of the 18th Working Conference on Reverse Engineering, WCRE*, pages 275 – 279. IEEE, October 2011.
- [Fin12] *FindBugs*, 2012. <http://findbugs.sourceforge.net/>, visited on August 8th, 2012.
- [Fow99] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999, ISBN 978-0-201-48567-2.
- [Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002, ISBN 978-0-321-12742-6.
- [GD12] Göde, Nils and Florian Deissenboeck: *Delta Analysis*. *Softwaretechnik-Trends*, 32(2):69 – 70, May 2012.
- [GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995, ISBN 978-0-201-63361-2.
- [Gla03] Glass, Robert L.: *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003, ISBN 978-0-321-11742-7.
- [GM05] Gil, Joseph and Itay Maman: *Micro patterns in Java code*. In *Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA*, pages 97 – 116. ACM, 2005.

-
- [HHHL03] Heuzeroth, Dirk, Thomas Holl, Gustav Högström, and Welf Löwe: *Automatic Design Pattern Detection*. In *Proceedings of the 11th International Workshop on Program Comprehension, IWPC*, pages 94 – 103. IEEE, 2003.
- [HKW⁺08] Herold, Sebastian, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolk, Raffaella Mirandola, Benjamin Hummel, Michael Meisinger, and Christian 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, 2008.
- [HWY⁺09] Han, Zhixiong, Linzhang Wang, Liqian Yu, Xin Chen, Jianhua Zhao, and Xuandong Li: *Design pattern directed clustering for understanding open source code*. In *Proceedings of the 17th International Conference on Program Comprehension, ICPC*, pages 295 – 296. IEEE, May 2009.
- [IEE00] IEEE-SA Standards Board: *Recommended Practice for Architectural Description of Software-Intensive Systems*. Technical Report IEEE Std 1471-2000, IEEE, 2000.
- [Int11] *IntelliJ IDEA*, 2011. <http://www.jetbrains.com/idea/>, visited on May 4th, 2012.
- [Jav99] *Code Conventions for the Java Programming Language*, 1999. <http://www.oracle.com/technetwork/java/codeconv-138413.html>, visited on May 25th, 2012.
- [Jav12] *Java Platform, Enterprise Edition, Technical Documentation*, 2012. <http://docs.oracle.com/javaee/>, visited on April 18th, 2012.
- [KBC05] Kalnins, Audris, Janis Barzdins, and Edgars Celms: *Model Transformation Language MOLA*. In *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 62 – 76. Springer, 2005, ISBN 978-3-540-28240-2.
- [KBWA94] Kazman, Rick, Len Bass, Mike Webb, and Gregory Abowd: *SAAM: A Method for Analyzing the Properties of Software Architectures*. In *Proceedings of the 16th International Conference on Software Engineering, ICSE*, pages 81 – 90. IEEE, 1994, ISBN 978-0-8186-5855-X.
- [KDGA12] Khomh, Foutse, Massimiliano Di Penta, Yann Gaël Guéhéneuc, and Giuliano Antoniol: *An exploratory study of the impact of antipatterns on class change- and fault-proneness*. *Empirical Software Engineering*, 17(3):243 – 275, 2012.

- [Ker04] Kerievsky, Joshua: *Refactoring to Patterns*. Addison-Wesley, 2004, ISBN 978-0-321-21335-8.
- [Kin76] King, James C.: *Symbolic execution and program testing*. Communications of the ACM, 19(7):385 – 394, 1976, ISSN 0001-0782.
- [KJ04] Kircher, Michael and Prashant Jain: *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley, 2004, ISBN 978-0-470-84525-7.
- [KK11] Klatt, Benjamin and Klaus Krogmann: *Towards Tool-Support for Evolutionary Software Product Line Development*. Softwaretechnik-Trends, 31(2):38 – 39, May 2011.
- [KKR10] Krogmann, Klaus, Michael Kuperberg, and Ralf Reussner: *Using genetic search for reverse engineering of parametric behavior models for performance prediction*. IEEE Transactions on Software Engineering, 36(6):865 – 877, 2010.
- [KLMN06] Knodel, Jens, Mikael Lindvall, Dirk Muthig, and Matthias Naab: *Static evaluation of software architectures*. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, CSMR, pages 285 – 294. IEEE, March 2006.
- [Koe95] Koenig, Andrew: *Patterns and antipatterns*. Journal of Object Oriented Programming, 8(1):46 – 48, March 1995.
- [Kos00] Koschke, Rainer: *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, University of Stuttgart, 2000.
- [KOV03] Kazman, Rick, Liam O’Brien, and Chris Verhoef: *Architecture Reconstruction Guidelines*. Technical Report CMU/SEI-2002-TR-034, Carnegie Mellon Software Engineering Institute, November 2003.
- [KPS⁺99] Krikhaar, René, André Postma, Alex Sellink, Marc Stroucken, and Chris Verhoef: *A two-phase process for software architecture improvement*. In *Proceedings of the 15th International Conference on Software Maintenance*, ICSM, pages 371 – 380. IEEE, 1999.
- [KR87] Kafura, Dennis and Geereddy R. Reddy: *The Use of Software Complexity Metrics in Software Maintenance*. IEEE Transactions on Software Engineering, 13(3):335 – 343, March 1987, ISSN 0098-5589.
- [KRG⁺11] Kpodjedo, Segla, Filippo Ricca, Philippe Galinier, Yann Gaël Guéhéneuc, and Giuliano Antoniol: *Design evolution metrics for defect prediction in object oriented systems*. Empirical Software Engineering, 16(1):141 – 175, 2011.

-
- [Kri97] Krikhaar, René: *Reverse Architecting Approach for Complex Systems*. In *Proceedings of the 13th International Conference on Software Maintenance*, ICSM, pages 4 – 11. IEEE, 1997, ISBN 978-0-8186-8013-X.
- [Kro10] Krogmann, Klaus: *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*. PhD thesis, Karlsruhe Institute of Technology (KIT), 2010.
- [KSB⁺11] Koziolok, Heiko, Bastian Schlich, Carlos Bilich, Roland Weiss, Steffen Becker, Klaus Krogmann, Mircea Trifu, Raffaella Mirandola, and Anne Koziolok: *An Industrial Case Study on Quality Impact Prediction for Evolving Service-Oriented Software*. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE, pages 776 – 785. ACM, 2011, ISBN 978-1-4503-0445-0.
- [KSRP99] Keller, Rudolf K., Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé: *Pattern-Based Reverse-Engineering of Design Components*. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE, pages 226 – 235. IEEE, May 1999.
- [Lak97] Lakhotia, Arun: *A unified framework for expressing software subsystem classification techniques*. *Journal of Systems and Software*, 36(3):211 – 231, 1997, ISSN 0164-1212.
- [Leh80] Lehman, Meir M.: *On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle*. *Journal of Systems and Software*, 1(3):213 – 221, 1980.
- [Leh96] Lehman, Meir M.: *Laws of Software Evolution Revisited*. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT, pages 108 – 124. Springer, 1996, ISBN 978-3-540-61771-X.
- [LMV05] Lindstrom, Gary, Peter C. Mehltz, and Willem Visser: *Model Checking Real Time Java Using Java PathFinder*. In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 444 – 456. Springer, 2005, ISBN 978-3-540-29209-8.
- [LS80] Lientz, Bennett P. and E. Burton Swanson: *Software Maintenance Management*. Addison-Wesley, 1980, ISBN 978-0-20104-205-3.
- [LTC02] Lindvall, Mikael, Roseanne Tesoriero, and Patricia Costa: *Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture*. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS, pages 77 – 86. IEEE, 2002, ISBN 978-0-7695-1339-5.

- [LW07] Lau, Kung Kiu and Zheng Wang: *Software component models*. IEEE Transactions on Software Engineering, 33(10):709 – 724, October 2007, ISSN 0098-5589.
- [LXZS06] Lung, Chung Horng, Xia Xu, Marzia Zaman, and Anand Srinivasan: *Program restructuring using clustering techniques*. Journal of Systems and Software, 79(9):1261 – 1279, September 2006, ISSN 0164-1212.
- [LYN⁺09] Liu, Hui, Limei Yang, Zhendong Niu, Zhyi Ma, and Weizhong Shao: *Facilitating software refactoring with appropriate resolution order of bad smells*. In Vliet, Hans van 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, pages 265 – 268. ACM, 2009, ISBN 978-1-60558-001-2.
- [Mar94] Martin, Robert C.: *OO Design Quality Metrics - An Analysis of Dependencies*. In *Proceedings of the Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*. ACM, October 1994.
- [Mar04] Marinescu, Radu: *Detection strategies: Metrics-based rules for detecting design flaws*. In *Proceedings of the 20th International Conference on Software Maintenance*, ICSE, pages 350 – 359. IEEE, September 2004.
- [Mar09] Martin, Robert C.: *Clean Code - A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009, ISBN 978-0-13-235088-4.
- [Mas12] *Massey Architecture Explorer*, 2012. <http://xplrarc.massey.ac.nz/>, visited on June 12th, 2012.
- [McC76] McCabe, Thomas J.: *A Complexity Measure*. IEEE Transactions on Software Engineering, SE-2(4):308 – 320, December 1976, ISSN 0098-5589.
- [MD08] Mens, Tom and Serge Demeyer: *Software Evolution*. Springer, 2008, ISBN 978-3-540-76439-7.
- [MEG03] Medvidovic, Nenad, Alexander Egyed, and Paul Grünbacher: *Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery*. In *Proceedings of the 2nd International Workshop from Software Requirements to Architectures*. IEEE, May 2003.
- [Mey09] Meyer, Matthias: *Musterbasiertes Re-Engineering von Softwaresystemen*. PhD thesis, University of Paderborn, December 2009. In German.

-
- [Mic12] *Microsoft COM: Component Object Model Technologies*, 2012. <http://www.microsoft.com/com/default.aspx>, visited on June 11th, 2012.
- [MK01] Mendonça, Nabor C. and Jeff Kramer: *An Approach for Recovering Distributed System Architectures*. *Automated Software Engineering*, 8:311 – 354, 2001, ISSN 0928-8910.
- [MNS01] Murphy, Gail C., David Notkin, and Kevin J. Sullivan: *Software Reflexion Models: Bridging the Gap between Design and Implementation*. *IEEE Transactions on Software Engineering*, 27(4):364 – 380, April 2001, ISSN 0098-5589.
- [MR09] Marino, Jim and Michael Rowley: *Understanding SCA (Service Component Architecture)*. Addison-Wesley, 2009, ISBN 978-0-321-51508-7.
- [MSC⁺01] Mancoridis, Spiros, Timothy S. Souder, Yih Farn Chen, Emden R. Gansner, and Jeffrey L. Korn: *REportal: A Web-based Portal Site for Reverse Engineering*. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 221 – 230. IEEE, 2001.
- [MW05] Meyer, Matthias and Lothar Wendehals: *Selective Tracing for Dynamic Analyses*. In *Proceedings of the 1st Workshop on Program Comprehension through Dynamic Analysis, PCODA*, pages 33 – 37. University of Antwerp, 2005.
- [Nie04] Niere, Jörg: *Inkrementelle Entwurfsmustererkennung*. PhD thesis, University of Paderborn, 2004. In German.
- [NSW⁺02] Niere, Jörg, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh: *Towards Pattern-Based Design Recovery*. In *Proceedings of the 24th International Conference on Software Engineering, ICSE*, pages 338 – 348. ACM, May 2002.
- [NWW03] Niere, Jörg, Jörg P. Wadsack, and Lothar Wendehals: *Handling Large Search Space in Pattern-Based Reverse Engineering*. In *Proceedings of the 11th International Workshop on Program Comprehension, IWPC*, pages 274 – 279. IEEE, 2003.
- [Obj06] Object Management Group: *CORBA Component Model, Version 4.0*, 2006. <http://www.omg.org/spec/CCM/4.0/>, Document formal/06-04-01.
- [Obj10] Object Management Group: *Unified Modeling Language (UML) 2.3 Superstructure Specification*, May 2010. <http://www.omg.org/spec/UML/2.3/>, Document formal/2010-05-05.
- [Obj11] Object Management Group: *Meta Object Facility (MOF) Query/View/Transformation Specification 1.1*, 2011. <http://www.omg.org/spec/QVT/1.1/>, Document formal/2011-01-01.

- [Obj12] Object Management Group: *Object Constraint Language (OCL), Version 2.3.1*, 2012. <http://www.omg.org/spec/OCL/2.3.1>, Document formal/2012-01-01.
- [OSL05] O’Brien, Liam, Dennis Smith, and Grace Lewis: *Supporting Migration to Services using Software Architecture Reconstruction*. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 81 – 91. IEEE, 2005.
- [Pac03] Pacione, Michael J.: *A Review and Evaluation of Dynamic Visualisation Tools*. Technical Report EFoCS-50-2003, University of Strathclyde, Scotland, 2003.
- [Par94] Parnas, David Lorge: *Software Aging*. In *Proceedings of the 16th International Conference on Software Engineering, ICSE*, pages 279 – 287. IEEE, 1994, ISBN 978-0-8186-5855-X.
- [Par11] Parnas, David Lorge: *The Risks of Stopping Too Soon*. Communications of the ACM, 54(6):31 – 33, June 2011.
- [PR10] Păsăreanu, Corina S. and Neha Rungta: *Symbolic PathFinder: symbolic execution of Java bytecode*. In *Proceedings of the International Conference on Automated Software Engineering, ASE*, pages 179 – 180. ACM, 2010, ISBN 978-1-4503-0116-9.
- [Pre01] Prechelt, Lutz: *Kontrollierte Experimente in der Softwaretechnik: Potenzial und Methodik*. Springer, 2001, ISBN 978-3-540-41257-3.
- [PRW03] Pacione, Michael J., Marc Roper, and Murray Ian Wood: *A Comparative Evaluation of Dynamic Visualisation Tools*. In *Proceedings of the 10th Working Conference on Reverse Engineering, WCRE*, pages 80 – 89. IEEE, 2003.
- [PTV⁺10] Passos, Leonardo, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor C. Mendonça: *Static Architecture-Conformance Checking: An Illustrative Overview*. IEEE Software, 27(5):82 – 89, September/October 2010, ISSN 0740-7459.
- [QBe06] *QBench*, 2006. <http://www.qbench.de>, visited on February 13th, 2012.
- [RBB⁺11] Reussner, Ralf, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolk, Heiko Koziolk, Klaus Krogmann, and Michael Kuperberg: *The Palladio Component Model*. Technical Report 2011,14, Fakultät für Informatik, Institut für Programmstrukturen und Datenorganisation (IPD), University of Karlsruhe, Germany, 2011.
- [RH06] Reussner, Ralf and Wilhelm Hasselbring (editors): *Handbuch der Software-Architektur*. dpunkt.verlag, 1st edition, 2006, ISBN 978-3-89864-372-7. In German.

-
- [Rie96] Riel, Arthur J.: *Object-oriented Design Heuristics*. Addison-Wesley, 1996, ISBN 978-0-201-63385-X.
- [RL04] Roock, Stefan and Martin Lippert: *Refactorings in großen Softwareprojekten – Komplexe Restrukturierungen erfolgreich durchführen*. dpunkt.verlag, 2004, ISBN 978-3-89864-207-0. In German.
- [RLGB⁺11] Rosik, Jacek, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar, and Dave Connolly: *Assessing architectural drift in commercial software development: a case study*. *Software: Practice and Experience*, 41(1):63 – 86, 2011, ISSN 1097-024X.
- [RRMP08] Rausch, Andreas, Ralf Reussner, Raffaella Mirandola, and Frantisek Plasil: *The Common Component Modeling Example - Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008, ISBN 978-3-540-85288-9.
- [Sar03] Sartipi, Kamran: *Software Architecture Recovery based on Pattern Matching*. In *Proceedings of the 19th International Conference on Software Maintenance, ICSM*, pages 293 – 296. IEEE, September 2003, ISBN 978-0-7695-1905-9.
- [SDM12] Shah, Syed M. Ali, Jens Dietrich, and Catherine McCartin: *Making Smart Moves to Untangle Programs*. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering, CSMR*, pages 359 – 364. IEEE, March 2012.
- [SG96] Shaw, Mary and David Garlan: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996, ISBN 978-0-131-82957-2.
- [SGM02] Szyperski, C., D. Gruntz, and S. Murer: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002, ISBN 978-0-201-74572-0.
- [SH08] Sartipi, Kamran and Lei Hu: *Behavior-Driven Design Pattern Recovery*. In *Proceedings of the IASTED International Conference on Software Engineering and Applications*, pages 179 – 185. IASTED, 2008.
- [Sis11] *SISSy - Structural Investigation in Software Systems*, 2011. <http://www.sqools.org/sissy/>, visited on February 20th, 2012.
- [SKR08] Sarkar, Santonu, 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, September/October 2008.

- [SKSS07] Stürmer, Ingo, Ingo Kreuz, Wilhelm Schäfer, and Andy Schürr: *The MATE Approach: Enhanced Simulink/Stateflow Model Transformation*. In *Proceedings of MathWorks Automotive Conference*. MathWorks, June 2007.
- [Som10] Sommerville, Ian: *Software Engineering*. Pearson, 9th edition, 2010, ISBN 978-0-137-05346-9.
- [Son12] *Sonargraph-Architect*, 2012. <http://www.hello2morrow.com/products/sonargraph/architect>, visited on June 12th, 2012.
- [SPL03] Seacord, Robert C., Daniel Plakosh, and Grace A. Lewis: *Modernizing Legacy Systems*. Addison-Wesley, 2003, ISBN 978-0-321-11884-4.
- [SRK⁺09] Sarkar, Santonu, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam: *Modularization of a Large-Scale Business Application: A Case Study*. IEEE Software, 26(2):28 – 35, March/April 2009.
- [SSL01] Simon, Frank, Frank Steinbrückner, and Claus Lewerentz: *Metrics Based Refactoring*. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, CSMR, pages 30 – 39. IEEE, 2001, ISBN 978-0-7695-1028-0.
- [SSRB00] Schmidt, Douglas C., Michael Stal, Hans Rohnert, and Frank Buschmann: *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000, ISBN 978-0-471-60695-6.
- [Str12] *Restructure101, Structure101, and Structure101build*, 2012. <http://http://www.headwaysoftware.com/products/>, visited on June 12th, 2012.
- [SW03] Smith, Connie U. and Lloyd G. Williams: *More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot*. In *Proceedings of the Computer Measurement Group Conference*. Performance Engineering Services and Software Engineering Research, 2003.
- [Tai07] Taibi, Toufik (editor): *Design Patterns Formalization Techniques*. IGI Publishing, March 2007, ISBN 978-1-599-04219-0.
- [TGLH00] Tran, John B., Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt: *Architectural Repair of Open Source Software*. In *Proceedings of the 8th International Workshop on Program Comprehension*, IWPC, pages 48 – 59. IEEE, 2000, ISBN 978-0-7695-0656-9.
- [TH00] Tzerpos, Vassilios and Richard C. Holt: *ACDC: An Algorithm for Comprehension-Driven Clustering*. In *Proceedings of the 7th Working Conference on Reverse Engineering*, WCRE, pages 258 – 267. IEEE, 2000, ISBN 0-7695-0881-2.

-
- [TK03] Tahvildari, Ladan and Kostas Kontogiannis: *A Metric-Based Approach to Enhance Design Quality Through Meta-Pattern Transformations*. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, CSMR, pages 183 – 192. IEEE, March 2003.
- [TM03] Tourwé, Tom and Tom Mens: *Identifying refactoring opportunities using logic meta programming*. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, CSMR, pages 91 – 100. IEEE, March 2003.
- [TMD09] Taylor, Richard N., Nenad Medvidovic, and Eric Dashofy: *Software Architecture: Foundations, Theory, and Practice*. Wiley, February 2009, ISBN 978-0-470-16774-8.
- [Tra07] Travkin, Dietrich: *Bewertung automatisch erkannter Ausprägungen von Software-Mustern*. In *Proceedings of the Software Engineering 2007 - Beiträge zu den Workshops*, volume 106 of *Lecture Notes in Informatics*, pages 369 – 372. Gesellschaft für Informatik, March 2007. In German.
- [TSG04] Trifu, Adrian, Olaf Seng, and Thomas Genssler: *Automated Design Flaw Correction in Object-Oriented Systems*. In *Proceedings of the 8th Conference on Software Maintenance and Reengineering*, CSMR, pages 174 – 183. IEEE, 2004, ISBN 978-0-7695-2107-X.
- [UZ09] Umar, Amjad and Adalberto Zordan: *Reengineering for service oriented architectures: A strategic decision model for integration versus migration*. *Journal of Systems and Software*, 82:448 – 462, March 2009, ISSN 0164-1212.
- [VB07] Varró, Dániel and András Balogh: *The Model Transformation Language of the VIATRA2 Framework*. *Science of Computer Programming*, 68(3):214 – 234, October 2007.
- [vDHK⁺04] Deursen, Arie van, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva: *Symphony: View-Driven Software Architecture Reconstruction*. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture*, WICSA, pages 122 – 136. IEEE, 2004, ISBN 978-0-7695-2172-X.
- [vGB02] Gorp, Jilles van and Jan Bosch: *Design Erosion: Problems and Causes*. *Journal of Systems and Software*, 61(2):105 – 119, January 2002, ISSN 0164-1212.
- [vHFG⁺11] Hoorn, André van, Sören Frey, Wolfgang Goerigk, Wilhelm Haselbring, Holger Knoche, Sönke Köster, Harald Krause, Marcus Porembski, Thomas Stahl, Marcus Steinkamp, and Norman Wittmüss: *DynaMod project: Dynamic analysis for model-driven*

- software modernization. In *Joint Proceedings of the 1st International Workshop on Model-Driven Software Migration and the 5th International Workshop on Software Quality and Maintainability*, volume 708 of *CEUR Workshop Proceedings*, pages 12 – 13, March 2011.
- [Wen04] Wendehals, Lothar: *Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams*. In *Proceedings of the 6th Workshop Software Reengineering*, volume 24 of *Softwaretechnik-Trends*, pages 63 – 64. Gesellschaft für Informatik, 2004.
- [Wen07] Wendehals, Lothar: *Struktur- und verhaltensbasierte Entwurfsmustererkennung*. PhD thesis, University of Paderborn, September 2007. In German.
- [Zdu05] Zdun, Uwe: *Applying Patterns for Reengineering to the Web*. In Khan, Khaled M. and Yan Zhang (editors): *Managing Corporate Information Systems Evolution and Maintenance*, pages 167 – 196. Idea Group Publishing, 2005, ISBN 978-1-591-40366-1.
- [ZHB11] Zhang, Min, 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] Zhao, Jianjun, Hongji Yang, Liming Xiang, and Baowen Xu: *Change impact analysis to support architectural evolution*. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5):317 – 333, 2002, ISSN 1532-0618.

Own Publications

- [PvDB12] Marie Christin Platenius, Markus von Detten, and Steffen Becker. Archimetrix: Improved Software Architecture Recovery in the Presence of Design Deficiencies. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, CSMR, pages 255 – 264. IEEE, March 2012.
- [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, 2011.
- [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*. Gesellschaft für Informatik, February 2011.
- [vD11] Markus von Detten. Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection through Symbolic Execution. In *Proceedings of 10th ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE, pages 17 – 20. ACM, June 2011.
- [vD12] Markus von Detten. Archimetrix: A Tool for Deficiency-Aware Software Architecture Reconstruction. In *Proceedings of the 19th Working Conference on Reverse Engineering*, WCRE, pages 503 – 504. IEEE, 2012.
- [vDB11] Markus von Detten and Steffen Becker. Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems. In *Proceedings of the 7th International Conference on the Quality of Software Architectures*, QoSA, pages 23 – 32. ACM, June 2011.
- [vDHP⁺12] Markus von Detten, Christian Heinzemann, Marie Christin Platenius, Jan Rieke, Dietrich Travkin, and Stephan Hildebrandt. Story Diagrams – Syntax and Semantics. Technical Report tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2012. Ver. 0.2.
- [vDMT10a] Markus von Detten, Matthias Meyer, and Dietrich Travkin. Reclipse - A Reverse Engineering Tool Suite. Technical Report tr-

- ri-10-312, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, March 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*, pages 299 – 300. ACM, May 2010.
- [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.
- [vDPB13] Markus von Detten, Marie Christin Platenius, and Steffen Becker. Reengineering Component-Based Software Systems with Archimetrix. *Journal on Software and Systems Modeling*, 2013. Theme Issue on Models for Quality of Software Architecture, to appear.
- [vDT10] Markus von Detten and Dietrich Travkin. An Evaluation of the Reclipse Tool Suite based on the Static Analysis of JHotDraw. Technical Report tr-ri-10-322, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, October 2010.

Supervised Theses

- [Foc10] Markus Fockel. Interpretation von Graphtransformationsregeln zur statischen Erkennung von Software-Mustern. Master's thesis, University of Paderborn, October 2010. In German.
- [Pla09] Marie Christin Platenius. Berücksichtigung von Objektmengen bei der dynamischen Entwurfsmustererkennung. Bachelor's thesis, University of Paderborn, October 2009. In German.
- [Pla11] Marie Christin Platenius. Reengineering of Design Deficiencies in Component-Based Software Architectures. Master's thesis, University of Paderborn, October 2011.
- [Str13] Christian Stritzke. Considering Architectural Knowledge in the Reverse Engineering Process of Component-Based Software Systems. Master's thesis, University of Paderborn, 2013. to appear.
- [Tra11] Oleg Travkin. Kombination von Clustering- und musterbasierten Reverse-Engineering-Verfahren. Master's thesis, University of Paderborn, June 2011. In German.
- [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.

List of Figures

1.1	Overview of the reengineering process with Archimetrix	6
1.2	An example business information system	9
2.1	Overview of the research areas related to this thesis	11
2.2	Reengineering terminology (Figure adapted from [CC90])	20
3.1	General form of the <i>Transfer Object Ignorance</i> deficiency	37
3.2	Variation of the <i>Transfer Object Ignorance</i> deficiency: A class is exposed by returning it on a call.	39
3.3	Running example: <i>Transfer Object Ignorance</i>	40
3.4	Reengineered version of the running example from Figure 3.3	41
4.1	The reengineering process with Archimetrix	47
4.2	Process for the documentation and formalisation of design deficiencies	51
4.3	Structural formalisation of the <i>Transfer Object Ignorance</i> deficiency	53
5.1	Architecture reconstruction process in SoMoX (from [Kro10])	58
5.2	Merge, composition, or discarding of a component candidate from the running example	59
5.3	Example of component candidates during the clustering	60
5.4	Dependencies between the metrics and strategies	65
5.5	Example of the influence of a <i>Transfer Object Ignorance</i> deficiency on the architecture recovery	66
5.6	Relationship between Source Code Decorator, GAST, and Service Architecture Meta Model (Figure adapted from [Tra11])	72
5.7	Illustration of the relationship between exemplary result model instances (Figure adapted from [Tra11])	73
6.1	Relevance analysis result calculation	81
7.1	Process for the pattern detection with Reclipse	86
7.2	Candidate for an occurrence of the <i>Transfer Object Ignorance</i> deficiency	88
7.3	Behavioural formalisation of the <i>Transfer Object Ignorance</i> deficiency	89
7.4	Expected behaviour for the candidate from Figure 7.2	90
7.5	Structural formalisation of the abstract <i>Component Composition</i> pattern	93
7.6	Structural formalisation of the <i>Direct Composition</i> pattern	93
7.7	Structural formalisation of the <i>Indirect Composition</i> pattern	95

7.8	Structural formalisation of the abstract <i>Component</i> pattern . . .	95
7.9	Structural formalisation of the <i>Direct Component Classes</i> pattern	96
7.10	Structural formalisation of the <i>Indirect Component Classes</i> pattern	96
9.1	Process for the removal of deficiency occurrences	113
9.2	Removal strategy: <i>Mark exposed class as transfer object</i>	117
9.3	Removal strategy: <i>Move called method</i>	118
9.4	Architecture preview for the removal of <i>Transfer Object Ignorance</i> occurrence no. 1	121
9.5	Architecture preview for the removal of <i>Transfer Object Ignorance</i> occurrence no. 2	121
10.1	Illustrative overview of the software architecture of Archimetric .	128
10.2	Configuration of the metric weights for the architecture reconstruction	129
10.3	Reconstructed components	130
10.4	Reconstructed architecture	131
10.5	Result of the component relevance analysis	132
10.6	Example deficiency specification: <i>Interface Violation</i>	133
10.7	Dialogue for the selection of components for the deficiency detection	133
10.8	List view of the detected deficiency occurrences	134
10.9	Host graph view of a detected <i>Interface Violation</i> occurrence . .	134
10.10	Result of the design deficiency ranking	135
10.11	Selection of an automated removal strategy	135
10.12	Architecture preview	136
10.13	Initially reconstructed architecture of the Store Example	141
10.14	Reconstructed architecture after the removal of the most relevant <i>Interface Violation</i> occurrence	143
10.15	Conceptual architecture of Palladio Fileshare (adapted from [KKR10])	146
10.16	Reconstructed architecture for Palladio Fileshare	147
10.17	An overview of the conceptual architecture of CoCoME (adapted from [HKW ⁺ 08])	151
10.18	A detailed view of the components <code>Inventory::Application</code> and <code>Inventory::Data</code>	152
10.19	Initially reconstructed architecture for the reference implementation of CoCoME	153
10.20	Architecture preview for removal of <i>Interface Violation</i> occurrence #1	157
10.21	Initially reconstructed architecture for the SOFA implementation of CoCoME	158
A.1	Meta model of the generalised abstract syntax tree (GAST, adapted from [Tra11])	175
A.2	Excerpt of the Service Architecture Meta Model (SAMB, adapted from [Tra11])	178
A.3	Meta model of the Source Code Decorator (SCD, adapted from [Tra11])	180

B.1	Example occurrence of an <i>Interface Violation</i> deficiency	182
B.2	Example of the influence of an <i>Interface Violation</i> deficiency on the architecture recovery	182
B.3	Removal strategy for the <i>Interface Violation</i> deficiency	184
B.4	Auxiliary story diagram for the called by the removal strategy in Figure B.3	186
B.5	Auxiliary story diagram for the called by the removal strategy in Figure B.3	187
B.6	Structural formalisation of the <i>Interface Violation</i> deficiency . .	189
B.7	Example occurrence of an <i>Unauthorised Call</i> deficiency	190
B.8	Structural formalisation of the <i>Unauthorised Call</i> deficiency . . .	192
B.9	Example occurrence of an <i>Inheritance between Components</i> de- ficiency	193
B.10	Structural formalisation of the <i>Inheritance Between Components</i> deficiency	194

List of Tables

3.1	Examples of positive and negative patterns at different levels of generality	32
5.1	Influence of a <i>Transfer Object Ignorance</i> deficiency on the metric values	68
10.1	Properties of the Store Example system	141
10.2	Component relevance analysis results for the Store Example . . .	142
10.3	Detected design deficiencies in the Store Example	143
10.4	Detected <i>Interface Violation</i> occurrences in the Store Example .	144
10.5	Properties of the Palladio Fileshare system	147
10.6	Component relevance analysis results for Palladio Fileshare . . .	148
10.7	Detected design deficiencies in Palladio Fileshare	148
10.8	Properties of the reference implementation of the CoCoME system	153
10.9	Component relevance analysis results for the reference implementation of CoCoME	154
10.10	Detected design deficiencies in the reference implementation of CoCoME	155
10.11	Detected <i>Interface Violation</i> occurrences in the reference implementation of CoCoME and their ranking	156
10.12	Component relevance analysis results for the SOFA implementation of CoCoME	159
10.13	Detected design deficiencies in the SOFA implementation of CoCoME	160
C.1	Configuration used for the architecture reconstruction of the Store Example	198
C.2	Configuration used for the architecture reconstruction of Palladio Fileshare	199
C.3	Configuration used for the architecture reconstruction of the CoCoME reference implementation	200
C.4	Configuration used for the architecture reconstruction of the SOFA implementation of CoCoME	201

Index

- Anti Pattern, **16**, **31**
- Archimatrix Process, **46**
- Architect, *see* Software Architect
- Architectural Drift, **2**, **14**
- Architecture, *see* Software Architecture
- Architecture Preview, **119**
- Architecture Reconstruction, **3**, **12**, **46**
 - Metric Dependencies, **64**
 - Metrics, **14**, **61**
 - Reconstruction Process, **57**
 - Strategies, **63**
- Architecture Recovery, **3**
- Auxiliary Component Pattern, **91**
- Bad Smell, **16**, **31**
- Behavioural Analysis, **19**, **89**
- Business Information System, **1**
- Class Diagram View, **87**
- Closeness to Threshold Metric, **79**
- Clustering-Based Reverse Engineering, **4**, **14**
- Complexity Metric, **78**
- Component
 - Composite, **13**
 - Definition, **12**
 - Primitive, **13**
- Component Composition Pattern, **92**
- Component Pattern, **94**
- Component Relevance Analysis, **46**, **75**
- Component-Based Software Engineering, **1**
- Composite Component, **13**
- Conceptual Architecture, **13**
- Concrete Architecture, **13**
- Deficiency, *see* Design Deficiency
- Deficiency Detection, *see* Design Deficiency Detection
- Deficiency Expert, **46**
- Deficiency Ranking, *see* Design Deficiency Ranking
- Deficiency Removal
 - Architecture Preview, **119**
 - Automated, **116**
 - Manual, **114**, **122**
 - Removal Strategy, **21**, **117**
- Design Deficiency, **5**, **16**, **31**, **179**
 - Description, *see* Pattern Description
 - Formalisation, *see* Pattern Formalisation
 - Occurrence, *see* Pattern Occurrence
- Design Deficiency Detection, **48**, **85**
- Design Deficiency Ranking, **48**, **101**
- Design Erosion, **2**
- Direct Component Classes Pattern, **94**
- Direct Composition Pattern, **92**
- Dynamic Analysis, **19**, **89**
- False Positive, **17**
- Forward Engineering, **20**
- GAST, *see* Generalised Abstract Syntax Tree
- Generalised Abstract Syntax Tree, **71**, **173**
- Guide Template, **112**, **115**
- Host Graph View, **87**
- Indirect Component Classes Pattern, **94**

- Indirect Composition Pattern, 92
- Inheritance between Components, 191
- Interface Violation, 179
- Knowledge-Based Reverse Engineering, 4
- Negative Fragment, 189
- Pattern, 15
- Pattern Detection, 15
- Pattern View, 87
- Pattern-Based Reverse Engineering, 4, **15**
- Patterns
 - Abstract Pattern, 92
 - Architectural Pattern, 16
 - Design Pattern, 16
 - False Positive, 17
 - Implementation Pattern, 16
 - Pattern Candidate, 17
 - Pattern Description, 17
 - Pattern Formalisation, 17
 - Pattern Occurrence, 18
 - Pattern Role, 18
 - Specification, *see* Pattern Formalisation
 - True Positive, 17
- Primitive Component, 13
- Reengineering, 19
- Refactoring, 21
- Relevance Analysis, *see* Component Relevance Analysis
- Removal Guide, 112, **115**
- Removal Strategy, 21, **112**, 117
- Restructuring, 21
- Reverse Engineering, 20
 - Architecture Reconstruction, **12**, 46
 - Clustering-Based, 4, **14**
 - Knowledge-Based, 4
 - Metrics, 14
 - Pattern Detection, 15
 - Pattern-Based, 4, **15**
- Role Mapping, 112
- SAM, *see* Service Architecture Model
- SAMM, *see* Service Architecture Meta Model
- SCD, *see* Source Code Decorator
- Service Architecture Meta Model, 71, **175**
- Service Architecture Model, 71, **175**
- Software Aging, 2
- Software Architect, 46
- Software Architecture
 - Conceptual Architecture, 13
 - Concrete Architecture, 13
 - Definition, 12
 - Original Architecture, 120
 - Previewed Architecture, 120
- Software Architecture Reconstruction,
see Architecture Reconstruction
- Software Evolution, 1, 19
- Source Code Decorator, 71, **178**
- Static Analysis, 18, **87**
- Structural Analysis, 18, **87**
- Trace Collection, 88
- Transfer Object Ignorance, 35
- True Positive, 17
- Unauthorised Call, 188