universität
wien

D I S S E R T A T I O N

Titel der Dissertation

Supporting Software Architecture Documentation and Evolution

Semi-automated Architectural Component Model Abstraction, Pattern Identifaction, and Consistency Managment During Software Evolution.

verfasst von

Dipl.-Ing. Thomas Haitzer

angestrebter akademischer Grad

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2016

Studienkennzahl lt. Studienblatt:	A 786 880
Dissertationsgebiet lt. Studienblatt:	Informatik
Betreuer:	Univ.-Prof. Dr. Uwe Zdun

Declaration of Authorship

I, Thomas Haitzer, declare that this thesis with the title, "Supporting Software Architecture Documentation and Evolution" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

Every software has an architecture. A number of approaches propose to document this architecture from different perspectives using different views. A commonly used view is the architectural component view, which shows a system's overall structure and abstracts away the low-level details. A central problem of all architectural views, including component views, is that the architecture and implementation of a software system can drift apart as software systems evolve, often leading to architectural knowledge evaporation. A related problem is the erosion of architectural knowledge over time. Both problems can lead to inconsistent, outdated, or completely lost architectural knowledge.

A number of approaches have been proposed to automatically reconstruct architectural views from the source code, in some cases even including the detection of higher-level abstractions such as design patterns. However, the precision and recall of such approaches is still limited, and they are not designed to be continuously used in the software development process, but rather to reconstruct a view at a certain point in time. That is, the reconstructed architecture is likely to suffer from similar architectural knowledge evaporation problems in the future.

In this thesis, we provide evidence that component diagrams are beneficial to architecture understanding. Based on these findings, we then introduce a semi-automatic approach for creating architectural component views based on architecture abstraction specifications that allows automatic consistency checking during a system's evolution and thus reduces the risk of architectural drift and erosion. Our approach also supports the identification and documentation of architectural pattern instances in those views. In contrast to the aforementioned approaches, our approach explicitly considers the evolution of the documented system and actively supports this evolution by providing navigable documentation and consistency checking between the involved artifacts. It does not only focus on a limited set of abstractions, like certain design patterns, but can also support the identification and documentation of higher-level architectural patterns.

The evolution of software is a complex process that consists of multiple tasks that have many interdependencies. In large systems it usually involves multiple development teams that might be located in different time zones or countries to work on the same system together. In such a situation managing the order of these tasks or evolution steps and their distribution on the different teams becomes challenging. Our approach also takes the architectural decision making during evolution into account and aids the complex task of actually evolving a software system by letting the architects define multiple, interdependent implementation tasks and automatically generate plans for performing the implementation tasks that satisfy all dependencies and constraints.

Zusammenfassung

Jedes Programm hat eine Architektur. Einige Ansätze verwenden zur Dokumentation dieser Softwarearchitektur verschiedene Perspektiven, die in mehreren Sichten festgehalten werden. Eine oft verwendete Sicht sind Diagramme, die die Komponenten der Architektur und deren Verbindungen zeigen. Ein wichtiges Problem aller dieser Sichten, die Architekturkomponenten zeigen, ist, dass sich die dokumentierte Architektur und die Implementierung während der Evolution des Systems auseinander bewegen können. Dies führt oft zum Verlust von Architekturwissen. Ein verwandtes Problem ist die Erosion von Architekturwissen im Lauf der Zeit. Beide Probleme können zu inkonsistentem, obsoletem oder vollständig verlorenem Architekturwissen führen.

Einige Ansätze versuchen, Architektursichten automatisch aus dem Quellcode wiederherzustellen. Manche dieser Ansätze versuchen auch, Artefakte wie Entwurfsmuster, die auf einem höheren Abstraktionsniveau liegen, wiederherzustellen. Allerdings sind diese Ansätze limitiert im Bezug auf Präzision und Wiedererkennung von Entwurfsmustern und sind meist nicht dazu gedacht, den Software Prozess fortlaufend zu unterstützen, sondern zu einem Zeitpunkt eine Architektursicht wiederherzustellen. Sie verhindern nicht, dass eine wiederhergestellte Architektur und der Quellcode in der weiteren Entwicklung wieder auseinander driften.

In dieser Arbeit zeigen wir, dass Komponentendiagramme zum besseren Verständnis von Softwarearchitektur beitragen. Basierend auf diesen Erkenntnissen stellen wir einen semi-automatischen Ansatz zur Erstellung von Komponentensichten vor, der die Gefahr für den Verlust von Architekturwissen reduziert, in dem er während der Evolution eines Systems die Artefakte automatisch auf Konsistenz prüft. Unser Ansatz unterstützt auch die Identifikation und Dokumentation von Architekturmustern in den Komponentensichten. Im Gegensatz zu den erwähnten Ansätzen berücksichtigt unser Ansatz die Software Evolution und unterstützt den Softwarearchitekten während der Evolution, in dem er navigierbare Dokumentation zur Verfügung stellt und die Konsistenz der involvierten Artefakte prüft. Des Weiteren beschränkt sich unser Ansatz nicht auf fixe Abstraktionen wie einige, bestimmte Entwurfsmuster, sondern unterstützt die Identifikation und Dokumentation von Architekturmustern, die sich auf einem höheren Abstraktionsniveau befinden.

Die Evolution von Software ist ein komplexer Prozess, der aus vielen einzelnen Aufgaben besteht, die voneinander abhängig sein können. Bei großen Systemen müssen für die Evolution oft mehrere Entwicklerteams zusammenarbeiten, die eventuell über mehrere Länder und Zeitzonen verstreut sind. In solchen Szenarien ist die Koordination der einzelnen Aufgaben und der Reihenfolge ihrer Bearbeitung eine große Herausforderung. Unser Ansatz berücksichtigt auch die dokumentierten Entwurfsentscheidungen und hilft bei der komplexen Aufgabe, die die Weiterentwicklung (Evolution) eines Systems darstellt, indem er es erlaubt, die einzelnen Aufgaben und ihre Abhängigkeiten zu definieren und dann automatisch Pläne für die Abarbeitung der Aufgaben generiert, die die Abhängigkeiten und Einschränkungen unter den einzelnen Aufgaben berücksichtigen.

Acknowledgements

First of all, I would like to thank my advisor Prof. Uwe Zdun for his valuable input and guidance throughout the dissertation. Without his patience and advice this thesis would not have been possible. To my dear Sabrina I would like to say thank you for your listening ear and moral support when I needed it. Without you, this thesis would have never been completed. Elena Navarro and Srdjan Stevanetic, my co-authors, it was an honor working with you. The results of our cooperation play a substantial role in this dissertation. I want to thank all my co-workers for the fruitful discussions, advice, and moral support, especially Gerhard, Patrick and Simon, it was a pleasure sharing our cosy office. A big thank you goes to my family for supporting me during my studies in any way they could.

List of Publications

This doctoral dissertation is mainly based on research work which has been either already published in scientific workshops, conferences, and journals or submitted to a research venue (currently under review). In particular, content of the following publications has been used in this thesis:

- Thomas Haitzer and Uwe Zdun. "DSL-based Support for Semi-automated Architectural Component Model Abstraction Throughout the Software Lifecycle." In: *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '12. Bertinoro, Italy: ACM, 2012, pp. 61–70. ISBN: 978-1-4503-1346-9. DOI: 10.1145/2304696.2304709

- Thomas Haitzer and Uwe Zdun. "Controlled Experiment on the Supportive Effect of Architectural Component Diagrams for Design Understanding of Novice Architects." English. In: *Software Architecture*. Ed. by Khalil Drira. Vol. 7957. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 54–71. ISBN: 978-3-642-39030-2. DOI: 10.1007/978-3-642-39031-9_6

- Thomas Haitzer and Uwe Zdun. "Semi-automated architectural abstraction specifications for supporting software evolution." In: *Sci. Comput. Program.* 90 [Sept. 2014], pp. 135–160. ISSN: 01676423. DOI: 10.1016/j.scico.2013.10.004

- Srdjan Stevanetic et al. "Supporting Software Evolution by Integrating DSL-based Architectural Abstraction and Understandability Related Metrics." In: *Proceedings of the 2014 European Conference on Software Architecture Workshops*. ECSAW '14. Vienna, Austria: ACM, 2014, 19:1–19:8. ISBN: 978-1-4503-2778-7. DOI: 10.1145/2642803.2642822

- Thomas Haitzer and Uwe Zdun. "Semi-automatic Architectural Pattern Identification and Documentation Using Architectural Primitives." In: *J. Syst. Softw.* 102.C [Apr. 2015], pp. 35–57. ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.12.042

- Thomas Haitzer et al. "Reconciling software architecture and source code in support of software evolution." submitted. Oct. 2015

- Thomas Haitzer et al. "Architecting for Decision Making About Code Evolution." In: *Proceedings of the 2015 European Conference on Software Architecture Workshops*. ECSAW '15. Dubrovnik, Cavtat, Croatia: ACM, 2015, 52:1–52:7. ISBN: 978-1-4503-3393-1. DOI: 10.1145/2797433.2797487. URL: http://doi.acm.org/10.1145/2797433.2797487

Vita

Thomas Haitzer is a PhD student at the Research Group Software Architecture (SWA), Faculty of Computer Science, University of Vienna since 2010. Before that, he completed his Master in Software Engineering and Internet Computing at the Vienna University of Technology in 2010. His research interests include software architectures, architecture evolution, architecture documentation, software patterns, and model-driven development.

Contents

List of Figures

List of Tables

Abbreviations

AADSL	**A**rchitecture **A**bstraction **S**pecification **L**anguage
ADD	**A**rchitectural **D**esign **D**ecision
ADL	**A**rchitecture **D**escription **L**anguage
ADvISE	**A**rchitectural **D**esign Dec**I**sion **S**upport Fram**E**work
AK	**A**rchitectural **K**nowledge
SA	**S**oftware **A**rchitecture
CoCoADvISE	**C**onstrainable **C**ollaborative **A**rchitectural **D**esign Dec**I**sion **S**upport Fram**E**work
DSL	**D**omain-**s**pecific **L**anguage
MDD	**M**odel-**D**riven **D**evelopment
NCOM	**N**umber of **COM**ponents
NCONN	**N**umber of **CONN**ectors
NELEM	**N**umber of **ELEM**ents
NC	**N**umber of **C**lasses
NID	**N**umber of **I**ncoming **D**ependencies
NOD	**N**umber of **O**utgoing **D**ependencies
NIntD	**N**umber of **Int**ernal **D**ependencies
QT	**Q**uestion **T**ype
Qx	**Q**uestion x
RQx	**R**esearch **Q**uestion x
IDE	**I**ntegrated **D**evelopment **E**nvironment
GUI	**G**raphical **U**ser **I**nterface
sLoC	**L**ines **o**f **S**ource **C**ode
MVC	**M**odel **V**iew **C**ontroller
CSS	**C**ascading **S**tyle **S**heets
UML	**U**nified **M**odeling **L**anguage
EMF	**E**clipse **M**odeling **F**ramework
PSG	**P**rocess **S**tructure **G**raph
RM	**R**eflexion **M**odels
CSP	**C**onstraint **S**atisfaction **P**roblem

SOA	**S**ervice-**o**riented **A**rchitecture
SOAP	**S**imple-**O**bject **A**ccess **P**rotocol
GoF	**G**ang-**o**f **F**our
QVT	**Q**uery/**V**iew/**T**ransformation

Part I

Foundations and Research Overview

1 Introduction

Bass et al. [Bas+03] define software architecture in the following way: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." Based on this definition, it has been stated that every software has an architecture [Tay+10]. With growing complexity of a system's architecture, the need to document this architecture grows. A number of different approaches have been proposed for documenting software architectures [Kru95; Cle96]. Today a software architecture description is usually comprised of multiple views [ISO11; Cle+02; Hof+00; Kru95; Zac87; ISO11]. The component and connector view (or component view for short) of an architecture is a view that is often considered to contain the most significant architectural information [Cle+02]. This view deals with the components, which are units of runtime computation or data-storage, and the connectors which are the interaction mechanisms between components [PW92a; Cle+02]. An architectural component view is a high-level abstraction of the entities in the source code of the software system, as the software architecture concerns only the major design decisions about a software system, and abstracts from irrelevant details [JB05].

In practice, architecture component views are often documented in the form of drawn component diagrams that cannot easily be related to the source code. During the evolution of a system, however, this often leads to the problem that source code changes are not reflected in the architecture documentation, commonly referred to as architectural drift and architectural erosion. Taylor et al. [Tay+10] define architectural drift as the "introduction of principal design decisions into a system's descriptive architecture that (a) are not included in, encompassed by, or implied by the prescriptive architecture, but which (b) do not violate any of the prescriptive architecture's design decisions" and architectural erosion as "the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture" [Tay+10].

In small projects, architectural drift and erosion can be avoided, as it might be possible to understand and maintain well written source code without additional architectural documentation. For many larger systems, this is not an option, and additional architectural documentation is required to aid the understanding of the system and especially to comprehend the "big picture" by providing architectural knowledge about a system's design [Bro13]. Otherwise important knowledge like e.g.

knowledge about architectural components and implemented architectural patterns can get lost during a system's evolution.

In this thesis, we provide evidence that component diagrams are beneficial to architecture understanding (in Chapter 4). Based on these findings, in Chapter 5, we then introduce an approach for creating architectural component views based on architecture abstraction specifications that allows automatic consistency checking during a systems evolution and thus reduces the risk of architectural drift and erosion.

Another essential part of today's architectural knowledge is information about the patterns used in a system's architecture. Patterns codify reusable design expertise that provides time-proven solutions to commonly occurring software problems that arise in particular contexts and domains [SB03].

A considerable number of approaches support software pattern identification [BJ94; BP00; Shu+96]. Most of these approaches (see e.g. [Heu+03; KP96; BP00; Phi+03]) focus on automatically detecting design patterns in source code. Such pattern identification approaches are often restricted to design patterns that were identified by Gamma et al. [Gam+95] (GoF patterns). Architectural patterns, in contrast, convey broader information about a system's architecture as they usually are described at a larger scale than GoF patterns. The precision and recall of automatic pattern identification approaches in general and thus also the automatic identification of architectural patterns often requires a significant amount of manual corrections, as false positives need to be discarded and instances that were not found need to be added manually. However, in order to manually correct these mistakes, the architect needs to understand the system first. Similar to architectural component diagrams, architectural patterns are often documented in purely textual form that, at best, references specific elements from the source code. In Chapter 6 we propose a semi-automatic approach for identifying and documenting architectural patterns based on architectural component views that are annotated with pattern primitives [ZA05] and thus works on a higher level of abstraction rather then directly in the source code. In addition, we support the architect during the evolution of a software system through automatic consistency checks of the documented artifacts.

Software evolution itself is a challenging topic that has attracted a lot of attention during the last decades, since Bersoff et al. [Ber+80] and Lehman [Leh80] presented their seminal articles in the area. Even in the eighties, the need for managing the software evolution was already detected and highlighted as one of the most complex aspects of the software lifecycle. Basically, from the first steps of a software project onwards, the need of change starts to arise because new market needs constantly impose new requirements, supporting technology is updated, decisions about the software system change, and so on. In this context, the use of Software Architecture (SA) has been highlighted as an important asset because, SA can be used as an artifact for the evolution to guide the planning and restructuring of the software [Cue+13; Hol02], but it is also an artifact of the evolution, because it must be evolved itself [Bar+08].

In Chapter 8 we propose an iterative approach for evolving architectural knowledge and source code in a consistent manner. Our approach is based on two existing approaches by Garlan et al. [Gar+09] and Cuesta et al. [Cue+13] that proposed to use architectural knowledge as an evolution driver. We integrate these approaches with our approach for documenting architectural component views in order to provide a software evolution approach that aids the consistent evolution of the documented architecture and the source code.

When evolving a software system, not only the challenge of identifying the "right" changes to a system arises, but also the challenge of planning and performing the necessary tasks to actually evolve a system once the changes to a system's architecture are decided. When evolving a complex system, this often requires multiple teams to perform different inter-dependent tasks (evolution steps) in a coordinated fashion in order to avoid inconsistencies. In this thesis we propose an approach for planning architecture evolution that uses the Alloy model analyzer [Jac11] to automatically compute plans for the evolution based on tasks that integrates with our approach for documenting architectural component views.

When changes are made to a software system without careful consideration, often important qualities of the system like performance, stability, understandability, and many more, can start to degrade and, when left unchecked, can bring a software system to the point where it is no longer usable or feasible to maintain. In recent time, a number of approaches that measure a software systems properties with respect to different qualities like e.g. understandability have been proposed [Ste+14b; Ste+15]. In Chapter 7 we propose an extension to our approach for documenting architectural component views from Chapter 5, which integrates the automatic calculation of understandability metrics in order to reduce the risk that architectural changes impact the understandability of a documented architectural component view in a negative way.

1.1 Key Concepts and Terminology

In this section we introduce the most important concepts and terminology that is used in this thesis.

1.1.1 Software Architecture Documentation

We use the term Software architecture documentation in two ways. First, we use it to describe the process of creating an architectural documentation of a software system, and second, we use it to describe the documents, wikis, diagrams, documented ADDs, that together describe a software systems architecture [Cle+02]. Clements et al. state that documenting the architecture of a software system is as essential as the architecture itself, as architectural knowledge that is not

communicated is already lost. For small programs the source code might suffice as documentation but for bigger, more complicated systems this is not sufficient. Different approaches for architecture documentation have been proposed that capture architecture from different point of views in order to not only capture a systems structure but also other aspects like e.g. its expected behavior and functionality [Cle+02; Kru95].

1.1.2 Software Architecture Recovery

Software architecture recovery or software archaeology is the process of understanding and documenting the architecture of an existing software system from its source code and or other available sources [Tay+10]. This is necessary if the architectural knowledge has been lost. For example, if no documentation exists, or only outdated, inconsistent documentation exists. Manual software architecture recovery is a cumbersome process that requires a significant amount of time and thus is an expansive process. That is why a number of automatic approaches have been proposed which automatically identify implemented patterns or automatically group a system into architectural components. However most of these automatic approaches often require a significant number of manual corrections [Cor+10], which limits their applicability.

1.1.3 Design Pattern and Architectural Pattern

As already mentioned above, Schmidt and Buschmann [SB03] define patterns as "Patterns codify reusable design expertise that provides time-proven solutions to commonly occurring software problems that arise in particular contexts and domains." One of the best known sets of design patterns was described by Gamma et al. [Gam+95]. These patterns usually target problems on a detailed level of abstraction. For example the observer pattern [Gam+95] which describes a mechanism for notifying interested objects about the state changes of an observed object. Architectoral patterns like the ones described by Buschmann et al. [Bus+96] describe problems and solutions on a higher level of abstraction than design patterns. While there is not always a clear distinction between design patterns and architectural patterns, a design pattern often only affects a small part of a software system, while an architectural pattern usually affects the system as a whole. A more detailed description of architectural patterns is given in Section 6.2 in Chapter 6.

1.1.4 Domain Specific Language

Fowler [Fow10] defines a Domain Specific Language as: "a computer programming language of limited expressiveness focused on a particular domain." He also gives a wide range of examples on DSLs like regular expressions [Fri06] - which are a family of languages for finding patterns in text, CSS [Mey06] - which is a language for manipulating the layout and style of a web-page,

make [Mec04] - a tool handling software builds in Linux and Unix, ant [Hol05] - a language for describing builds in xml, graphviz [Ell+03] - which is a language for visualizing graphs, SQL - the standard query language for relational databases, and many more. Domain specific languages are usually not Turing-complete as they are tailored towards a specific purpose. One of the benefits of DSLs is, that since DSLs lack the complexity of general purpose languages, domain experts are able to learn these languages more easily and can edit or enhance the code written in a DSL. In this thesis we propose the usage of multiple DSLs to define architecture abstraction specifications (in Chapter 5) and to describe pattern templates and pattern instances (in Chapter 6).

1.1.5 Architectural Design Decision (ADD)

The ISO 42010 standard specifies that an architectural decision affects one or more architectural elements and pertains to one or more concerns [ISO11]. An ADD might also raise new concerns. In recent years, ADDs have become widely regarded as important architectural knowledge and a significant amount of research has been done in this area [Lyt+13b; Lyt+13a; JB05; Zim+07; Har+07]. ADDs are often documented using templates and capture the rationale behind an architecture relevant decision, why a solution has been chosen, what the alternatives would have been, and why the decision was necessary. This complements other approaches to architecture documentation like the 4+1 views by Kruchten [Kru95] as it captures another important aspect of a system's architecture. While a documentation created using the 4+1 views approach might document, that a specific pattern has been used, a documented ADD also tells that this decision might have been based on a specific requirement, which is long obsolete.

Thesis Structure

This PhD thesis is structured as follows:

In Part I, we introduce the research problems and questions this thesis addresses (see Chapter 2). In Chapter 3 we discuss the state of the art and compare the approach proposed in this thesis to existing work. We conclude Part I by describing the research methods we used in the creation of this thesis.

Part II first discusses our controlled experiment on the supportive effect of component diagrams, which motivates our choice to use architectural component views in the other approaches discussed in this part. These approaches focus on creating architecture documentation and keeping architecture documentation and source code consistent throughout the evolution by defining architectural abstraction specifications that relate the architecture documentation (architectural component views and architectural patterns) to source code.

In Part III we introduce our approaches that address research questions focusing on consistency management throughout the software evolution by integrating the approaches proposed in Part II with existing approaches, as well as proposing an approach for planning the software evolution itself.

Finally, in Part IV, we conclude by and giving an overview of the main contributions of the thesis, discussing limitations of the proposed approach, and open challenges that remain in the different research areas.

2 Problem Analysis and Research Approach

In this chapter, we motivate our work by discussing existing research problems that we identified in the research areas introduced in Chapter 1. We then formulate the research questions this thesis addresses based on the identified problems. We conclude this chapter by discussing the research methods that have been used in the creation of the approaches presented in this thesis.

2.1 Problem Statement

This dissertation contributes to different problems in the area of software architecture documentation and software architecture evolution.

Components or components and connectors have received a lot of focus in recent years, especially in the context of component and connectors as part of different view-models that were proposed for software architecture documentation, like the ones by Kruchten [Kru95]. While a number of studies have been conducted that focus on the usage of architecture documentation, like architectural component views in practice, only a limited number of empirical studies about architectural component views have been conducted to study their positive or negative effects while understanding a software system. As a foundation for our further work, we studied the effects of component diagrams on the understanding of a software system's source code by (novice) software architects in Chapter 4.

Research Question 1

Can architectural component views (in the form of UML component diagrams) have a positive effect during the understanding of a software system?

Studies researching the usage of software architecture documentation approaches in practice [Ros+13] often find that software architecture documentation "is outdated and not updated after changes to requirements or source code". Rost et al. also find that architecture documentation does often not provide the right information for the stakeholders, often is inconsistent, and that architecture documentation often does not provide sufficient navigation support. This fits with other observations

from Jansen et al. [Jan+07] who observe the problem that design and implementation of software systems often drift apart during implementation and system evolution. A number of approaches propose to resolve inconsistency between architecture documentation and source code by automatically creating/generating documentation from the source code [Abr+00; Die+08; DB11]. For some of these approaches the size of resulting diagrams (e.g. class diagrams) is a problem, while the approaches that automatically create abstractions often require manual corrections [Cor+10].

Based on these problems we formulate the next research question:

Research Question 2

Can we support the software architect in the creation and maintenance of architectural component views in order to mitigate the risk for architecture drift and erosion of the architecture and the source code during system evolution?

We study this research question in Chapter 5.

Like architectural component views, architecture patterns are one of the most important means to document and communicate a system's structure. If an architect or developer is familiar with an architectural pattern, she can draw conclusions about a system if she knows, that the system follows e.g. an N-tier architecture or uses a Message Bus [HW03]. While an architectural component view only describes a systems structure, architectural patterns can also include information about a system's behavior. However, documenting a system's architectural patterns is still a cumbersome task. Architectural patterns are often documented in the form of free-text documents with their quality highly depending on the creator of the document. This form of documentation usually has very limited ties to the actual source code and traceability links are purely text-based. Similar to architectural component views, the manual documentation of architectural pattern instances is tedious and prone to the same problems of becoming obsolete and inconsistent or being lost. The identification of architecture patterns during software architecture recovery is especially time consuming, as this requires the recovering architect to get a big picture understanding of the system. Due to this, a number of approaches have been proposed that support the automatic identification of patterns. A significant number of these pattern identification approaches focus on design patterns [Heu+03; KP96; BP00; Phi+03; Kac+06b] or only support a subset of specific patterns. In addition, automatic approaches often have problems with respect to false positives, which leads to additional manual effort when using such approaches. Zdun and Avgeriou [ZA05] proposed to use pattern primitives as basis for the documentation of architectural patterns and show the use of the primitives in the form of UML diagrams [Obj10]. While this approach proposes a model-driven approach to architectural pattern documentation, it does not consider traceability links and is a purely manual approach. We investigate the following research question:

Research Question 3

Can we support the software architect in the identification and documentation of architectural patterns during implementation and throughout the evolution of a software system?

We discuss this research question in Chapter 6.

Keeping architecture documentation and source code consistent during evolution is an important aspect of being able to communicate a system's architecture [Cle+02]. While existing architecture documentation helps people that have to evolve a system by providing insight into the architecture of the system under evolution, the plain existence of architecture documentation is not sufficient, as the quality of this documentation might be low, especially with respect to understandability. Just consider the extreme cases: A component diagram depicting only a client, a server and a connector between those two components. This diagram provides very little information that is not contained somewhere else or would be obvious from the source code, where client and server code are probably separated into different packages or projects. The same is true for any automatically generated class diagram for a system with more than a hundred classes, which is significantly below the number of classes in the prototypes for this thesis. This diagram, if it shows all classes and their relations, has at least a hundred boxes and many more lines between these boxes. However the research in our group shows, that such complex diagrams are not beneficial for the understanding of a software system [SZ14]. One possible approach to measure and quantify the quality of a software architecture documentation are metrics, more specifically understandability metrics like the ones discussed by Stevanetic and Zdun [SZ14]. This leads us to our next research question, which we tackle in Chapter 7:

Research Question 4

In how far is it possible to integrate and automatically calculate understandability metrics during the creation and evolution of architectural component views and support the architect in ensuring the quality/understandability of the documented architecture during evolution?

With respect to software archictecture documentation and evolution a huge number of approaches have been published [Kno+06; Pas+10; San+05; Mur+95a], some of those focus on documenting and evolving the architecture [Cue+13; Bar+12; Bar+13; FM06a] while some focus on reconstructing the software architecture from source code [Lun+06; RD99; Hol98; Men+02]. Some works establish traceability links between software architecture and source code [Tek+07; WP10; KZ10b]. Also, as already mentioned before, a lot of approaches automatically detect design patterns in source code [Heu+03; KP96; BP00; Phi+03; Kac+06b] or create automatic clusterings of source code [Abr+00; Cor+10; Det+10; MM01]. However, to the best of our knowledge, there are only very few approaches that specifically target the evolution of software architecture and source

code in a consistent manner and that treat the architecture and the code as first class assets. This brings us to the next research question:

Research Question 5

How to reconcile the different points of view that software architects and developers have when the software is being evolved and how to enforce the integration of software architecture and source code?

We address this research question in Chapter 8.

Software architecture evolution is a complex topic that has received a lot of attention over the years, but still remains a complex task. This is especially true for large software projects. There often are multiple development teams that have to coordinate in order to evolve different parts of a software system in parallel. For example there could be one team responsible for the server of an application and multiple development teams for the different clients that exist (Web, Android, iOS). In such cases, tasks like modifying the server's interface require a lot of communication effort. This is even worse, if the teams are located in different countries and or time zones. So there is a clear need to manage properly who is in charge of each requested change and how and when it will be carried out. While there exist a number of approaches for planning activities that have inter-dependencies in other areas that might be applicable in this situation, to the best of our knowledge, no approach has been applied to the field of software architecture evolution and integrated with the software architecture description itself.

The last research question we study in this thesis (Chapter 9) is:

Research Question 6

Is it possible to aid the architect in the description of the different, necessary evolution steps and their dependencies, and automatically provide plans for the specific evolution steps during the evolution of a software system?

2.2 Research Methods

The research in this thesis is based on Design Science research [Hev+04]. In Design Science research, first a research question is posed, and then the develop/evaluate cycle is continuously repeated until a satisfactory solution for the research question has been obtained. In the course of this research, the research question can be altered or refined. In the first iterations, usually simplifying assumptions are made, which are stepwise removed during later iterations. In the remainder of this section, we briefly introduce and discuss different methods that were used throughout this thesis.

2.2.1 Design Science Research

According to March and Smith [MS95] Design Science research aims to "produce and apply knowledge of tasks or situations in order to create effective artifacts", in contrast to explanation research which focuses on gaining theoretical knowledge [VK07].

According to Vaishnavi and Kuechler [VK07] Design Science research produces 5 different outputs: constructs, models, methods, instantiations. While constructs are the conceptual vocabulary of a domain, models are a set of prepositions expressing relationships among constructs. Methods are a set of steps to perform a task, while instantiations are operationalized constructs, models, and methods.

Design Science research is comprised of the following steps [VK07]:

- **Awareness of Problem**: this might result from different sources like e.g. earlier research efforts or other disciplines and it results in a proposal.
- **Suggestion**: is "a creative step where new functionality is envisioned based on a novel configuration of either existing or new and existing elements" [VK07]. This step results in a tentative design.
- **Development**: Here, the tentative design is further developed and implemented.
- **Evaluation**: The constructed artifact is evaluated according to criteria that are implicitly (or sometimes explicitly) mentioned in the proposal, e.g. with respect to performance.
- **Conclusion**: In this step, the evaluation results are judged to be sufficient or insufficient. In this phase, the results are also consolidated and written up.

The last step, **conclusion**, might create additional iterations in the research loop. In the research of this thesis, we included an optional fifth step, dependent on the results of the conclusion step, the communication of the written up results through articles in scientific journals or at conferences and workshops.

2.2.2 Case Study

Wohlin et al. [Woh+12] define a case study in software engineering as *"an empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified."* The aim of performing case studies is to understand how and why some phenomena occur within a specific time space,

as well as the mechanisms by which various cause-effect relationships are established [Eas+08; Woh+03]. Two types of case studies exist: *exploratory* case studies that are used to derive new hypotheses and build theories from the investigation of some phenomena, and *confirmatory* case studies that test existing theories [Eas+08]. According to Wohlin et al. case studies are suitable for industrial evaluation of software engineering methods and tools as they can avoid scale-up problems [Woh+03]. We evaluated the approaches and tools that we developed in this thesis using case studies based on existing real-life open-source systems. During the execution of the case studies we investigated e.g. the feasibility of our approaches, the usability of our prototypes or collected data to evaluate the performance of the introduced tools.

2.2.3 Controlled Experiment

Wohlin et al. [Woh+12] define a controlled experiment as an investigation of a testable hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables. The main difference between a controlled experiment and a case study is that an experiment samples over the variables that are being manipulated (controlled study), while a case study samples from the variables representing the typical situation (observational study).

The independent variables or factors represent the treatments in the experiment. In our case, we had one independent variable (the participants experiment group). Dependent variables are measured to investigate whether they are affected by the independent variables [Woh+12].

We have performed a controlled experiment to study the effects of component diagrams on the understanding of a software architecture, which we present in detail in Chapter 4. For this experiment we followed the guidelines by Kitchenham et al. [Kit+02] and Wohlin et al. [Woh+12] and manipulated one independent variable (the group), while we observed one dependent variable (the quality of the answers of the participants) and observed a number of other independent variables like prior experience to prevent effects from these other independent variables.

3 State of the Art

In this chapter we give an overview of the current state of the art with respect to the different research topcis that are relevant for this thesis, discuss the related work, and compare the discussed approaches to our proposals.

3.1 Approaches Focusing on Software Architecture Reconstruction

The approaches discussed in this section are related to the field of software architecture reconstruction. Ducasse and Pollet [DP09] presented a survey on the state-of-the-art in the field of software architecture reconstruction. They analyzed and categorized the existing approaches with respect to their goals, inputs, process, techniques, and outputs.

We have split the discussed approaches in this section into different groups: In Subsection 3.1.1 we present a number of selected articles that apply different approaches making use of automatic clustering. Subsection 3.1.2 discusses articles that propose different kinds of model-based approaches that create abstractions or views from source code. Finally, Subsection 3.1.3 presents selected approaches that are either hybrid approaches or other approaches that do not fit into one of the other sections but are nonetheless relevant for our work.

3.1.1 Software Architecture Reconstruction Approaches Based on Automatic Clustering

Abreu et al. introduce a reengineering approach using cluster analysis [Abr+00]. This approach uses six different affinity schemes and seven clustering methods to produce a series of clustering proposals to verify which one produces the best results. However, the clustering leads to solutions similar to those proposed by human experts only if the average number of classes per module is not too high.

Another approach for recovering architecture information is introduced by von Detten and Becker [DB11]. The authors combine clustering and (anti-)pattern information to extract components from existing source code.

Qingshan et al. [Qin+05] describe an architecture recovery approach that focuses on extracting the Process Structure Graph (PSG) of a system. While this approach works fully automatic, it does not allow to create views on different levels of abstraction but is limited to the PSG.

Corazza et al. [Cor+10] introduce a clustering approach that uses lexical information. It uses a probabilistic model and the Expectation Maximization algorithm to weigh this information and customizes the K-Medoids algorithm in order to group classes. In their case study they compare their approach with other automatic clustering approaches previously compared by Bittencourt and Guerrero [BG09]. In the case study they state that the authoritativeness values are close to 0.5 in 5 of 7 cases. This means that in five cases, it is necessary to execute move or join operations for about half the entities.

Maletic and Marcus [MM01] used an automatic clustering approach that utilized latent semantic indexing for the data-retrieval and a minimal spanning tree for partitioning the data. This approach shares the same problem with aforementioned clustering approaches: The results it produces need to be manually corrected.

Dietrich et al. [Die+08] describe an approach for analyzing Java dependency graphs with clustering. However this approach still needs the configuration of the separation level (the number of iterations of removing the edges with the maximum betweenness level) and does not produce stable results when code changes occur.

De Lucia et al. [Luc+07] integrate a latent semantic indexing approach [Dee+90] into a software artifact management system in order to recover traceability links. However they also state that one of the limitations in using information retrieval techniques is that in order to find all traceability links, it is necessary to manually discard a big amount of false positives.

All approaches discussed so far, deal with automatic recovery of design knowledge. More clustering approaches and clustering measures are reviewed and compared by Maqbool and Babri [MB07]. They define a number of groups of clustering algorithms and compare the performance of the different groups for different open source software projects. While Maqbool and Babri conclude which approach works best for each of the applications, they do not draw any conclusions regarding the overall effort necessary to correct the automatic clustering. A lot of clustering approaches assume that no architectural knowledge about a system exists and approaches using heuristic approaches based on inter-class relationships like the one by [Die+08] potentially have the problem that their clusterings are not stable with respect to changes in the source code as small changes in the source code might result in significantly different clusterings. This is especially true for heuristic algorithms that contain a random element. In contrast to all these approaches, the approach we

introduce in Chapter 5 is semi-automatic, enables the checking of design constraints during the abstraction process, and provides traceability between source code and models.

3.1.2 Model-based Approaches for Creating Architecture Abstractions and Views

Various approaches have been proposed for creating abstractions or views from source code. Scaniello et al. [Sca+10] propose an approach for semi-automatically detecting layers in software systems based on the algorithm introduced by Kleinberg [Kle99] while Sartipi models the process of recovering design patterns [Sar03] as a graph pattern matching problem between an entity relationship graph and an architecture pattern graph. Others use graph based approaches for keeping models synchronized [IK04]. A different data source use Brosig et al. [Bro+09], who describe how they extract a Palladio component model from Enterprise Java Beans and the runtime control flow.

Egyed [Egy04] describes an approach for model abstraction by using existing traceability information and abstraction rules. However, the author identified 120 abstraction rules for the example of UML class models, which need to be extended with a probability value because the rules may not always be valid.

Another important approach for mapping source code models to high-level models is introduced by Murphy et al. [Mur+95b]. They use software reflexion models which they compute from a mapping between source model and high-level model. However, it requires a substantial amount of effort, since it requires to define both, the high-level model and the mapping. In contrast to the approach proposed in this thesis their approach also does not support traceability links nor is it explicitly geared towards producing stable architecture abstraction during evolution. In addition, we extend our approach for documenting architectural component views to support the semi-automatic identification and documentation of architectural patterns.

An interesting software architecture recovery approach is introduced by Hassan and Holt [HH04]. It uses modification records from source code versioning systems. The authors also discuss how to use this information with software reflexion models to assist in the understanding of a system's architecture. While the approach provides additional information to the developers, its basic workings do not differ from the already discussed approach from Murphy et al. [Mur+95b].

Mens et al. [Men+02] propose intentional source code views that allow grouping of source code by concerns. These views are defined in a logic programming language. Their approach provides generic source views on a low abstraction level.

An approach that integrates source code with information found in problem reports and changelogs to do architecture analysis is presented by Pinzger et al. [Pin+05]. The analysis produces specific directed attributed graphs which then are integrated into a FAMIX [Tic+00] model and compute

architectural views using binary relational algebra based on an approach introduced by Holt et al. [Hol98]. These views then show intended and unintended couplings between architecture elements.

Other approaches [RD99; RR02] use manually written logic facts in Prolog to analyze object oriented applications based on static and dynamic information of a system in their approaches for reverse engineering. Like most of the architecture reconstruction approaches mentioned in this section, these approaches do not take the evolution of a software artifact into account, while the approach in this thesis specifically targets the evolution and tries to prevent architectural knowledge evaporation. Our approach also allows to identify and document architectural patterns and supports the architect by automatically creating traceability links between the different artifacts.

3.1.3 Hybrid and Other Approaches

Another approach that uses the package structure of a software system as a starting point is presented by Lungu et al. [Lun+06]. They propose a visual architecture recovery approach that introduces package patterns which they automatically detect in the package structure based on heuristics. This results in the same drawbacks that all automatic approaches share and that we discussed at the end of Section 3.1.1.

Passos et al. [Pas+10] give a illustrative overview on static architecture-conformance checking. They compare three approaches: The Lattix Dependency Manager (LDM) [San+05], which is based on Dependency-Structure Matrices, .QL [Moo+08], which is a source code query language (SCQL) [UM12; Haj+06], and the reflexion models (RM) introduced by Murphy et al. [Mur+95b]. As Passos et al. summarize, all of these approaches have drawbacks. While the LDM tool has very limited capabilities of expressing constraints, .QL has only a low abstraction level, and RMs have only limited support for architecture reasoning and discovery.

An approach that is similar to our approach for creating architectural component views, which we present in Chapter 5, is used by Feilkas et al. [Fei+09] to perform an industrial case study on the loss of architectural knowledge during system evolution. In order to measure the loss of architectural knowledge, they use an approach based on machine readable component descriptions and policies in XML that are created manually. However their approach offers only limited ways to describe mappings between components and source code as their mappings are solely based on regular-expressions that map package-names to components, while we provide more means for creating architecture abstraction specifications in Chapter 5 and then, in the remainder of this thesis use these architecture abstraction specifications as a starting point for semi-automatically identifying and documenting architectural patterns (see Chapter 6) as well as the definition of implementation tasks during software evolution in Chapter 9.

3.2 Identification and Documentation of Patterns

In this section we discuss the current state of the art with respect to the models for describing patterns, pattern identification, as well as pattern documentation. In Table 3.1 we give an overview of the related work discussed in this section and also provide a short comparison of this related work. Most of the related works focus on automatic design pattern identification while only a limited number focuses on finding architectural elements. As already discussed in Section 6.1 automatic approaches are limited by a high number of false positives while existing semiautomatic approaches focus either on a specific pattern [Sca+10] or on design patterns only [Guo+99]. In contrast, our approach focuses on architecture documentation and evolution of architectural patterns. It provides support for pattern variants and does not have the drawback of finding many false positives as it is semiautomatic and requires the software architect to annotate the architecture model with architectural primitive information. In Subsection 3.2.1 we discuss approaches with a focus on architectural patterns, while Subsection 3.2.2 discusses approaches focusing on design patterns. Our approach can broadly be categorized as a software architecture documentation approach.

TABLE 3.1: Comparison of related approaches

Approach	Pattern types	Method	Pattern variants support	Automation	Focus
Medividovic et al. [Med+03]	Architectural styles	Model comparison	None	Manual	Stemming architecture erosion
Yan et al. [Yan+04]	Architectural styles	State-machine	Design	Automatic	Stemming architecture erosion
Scaniello et al. [Sca+10]	Layers pattern	Link analysis	Implementation	Semiautomatic	Pattern identification
Sartipi [Sar03]	Architectural patterns	Graph transformation and AQL queries	A* heuristic matching	Automatic	Architecture reconstruction
Harris et al. [Har+95]	Style-library	Source code queries	Implementation	Automatic	Pattern identification
Lungu et al. [Lun+06]	Packaging patterns	Custom script	None	Semiautomatic	Architecture reconstruction
Paakki et al. [Paa+00]	Architectural and Design patterns	CSP	None	Automatic	Quality Assessment
Di Penta et al. [DP+07]	Architectural patterns	mu-calculus	Design and implementation	Automatic	Detecting SOA patterns
Wuyts [Wuy98]	Design patterns	Declarative reasoning	None	Manual	Finding structural relationships
Krämer et al. [KP96]	Design patterns	Prolog queries	None	Automatic	Architecture reconstruction

Table 3.1 – continued from previous page

Approach	Pattern types	Method	Pattern variants support	Automation	Focus
Tonella and Antoniol [TA01]	Design patterns	Concept analysis	None	Automatic	Pattern identification
Arévalo et al. [Are+04]	Collaboration patterns	Concept analysis	None	Automatic	Detection of collaboration patterns
Guéhéneuc et al. [GJ01]	Design patterns	Explanation-based CSP	Implementation	Automatic	Pattern identification
Alnusair et al. [Aln+13]	Design patterns	Semantic web technologies and first order logic	Design	Automatic	Pattern identification
Lucia et al. [DL+10]	Design patterns	Model checking	None	Automatic	Pattern identification
Kaczor et al. [Kac+06a]	Design patterns	Graph-based	None	Automatic	Pattern identification
von Detten [Det11]	Design patterns	Graph-based	None	Automatic	Pattern identification
Seeman and Gudenberg [SG98]	Design patterns	Graph-based	None	Automatic	Pattern identification
Balanyi and Ferenc [BF03]	Design patterns	Graph-based and DPML	None	Automatic	Pattern identification
Wendehals et al. [Wen03]	Design patterns	Graph-rewrite and fuzzy logic	Design and implementation	Automatic	Pattern identification
Tsantalis et al. [Tsa+06]	Design patterns	Similarity scoring between graph vertices	Implementation	Automatic	Pattern identification
Shull et al. [Shu+96]	Design patterns	Guidelines for manual identification	None	Manual	Architecture reconstruction
Bergenti and Poggi [BP00]	Design patterns	Interactive design assistance	None	Automatic	Design improvement
Palma et al. [Pal+12]	Design patterns	Goal-question-metric	None	Semiautomatic	Pattern recommendations
Guo et al. [Guo+99]	Design patterns	Rigi standard format	None	Semiautomatic	Architecture reconstruction
Heuzeroth et al. [Heu+03]	Design patterns	Custom detection algorithm per pattern	None	Automatic	Pattern identification
Washizaki et al. [Was+09]	Design patterns	Comparing code before and after a pattern's introduction	None	Semiautomatic	Pattern identification
Pinzger and Gall [PG02]	Design patterns	String-pattern-matching	Implementation	Automatic	Architecture reconstruction

Table 3.1 – continued from previous page

Approach	Pattern types	Method	Pattern variants support	Automation	Focus
Rasool and Mäder [RM11]	Design patterns	SQL queries and RegEx	Implementation	Automatic	Pattern identification
Stencel and Wegrzynowicz [SW08]	Design patterns	First order logic translated to SQL queries	Design and Implementation	Automatic	Pattern identification
Keller et al. [Kel+99]	Design patterns	Model-based	Implementation	Automatic	Reverse engineering
Philippow et al. [Phi+03]	Design patterns	Minimal key structures	Implementation	Automatic	Pattern recovery
Our approach	Architectural patterns	DSL based	Design + implementation	Semiautomatic	Architecture evolution

3.2.1 Approaches Based on Architectural Patterns

A number of other approaches have been presented that focus on the recovery of architectural pattern information. Medividovic et al. [Med+03] combine architectural recovery and architectural identification to create discovered and recovered architectural models and leverage architectural styles to identify and reconcile mismatches between them.

DiscoTect [Yan+04], which is introduced by Yan et al. focuses on the behavoral information of a system, as it uses a state machine to automatically detect architectural styles in low level execution events during the runtime of a system.

Scaniello et al. [Sca+10] propose an approach for semi-automatically detecting layers in software systems based on the algorithm introduced by Kleinberg [Kle99]. The authors implemented a prototype and provide a case study for JHotDraw [1].

Harris et al. [Har+95] propose a style library and use a recognition engine to detect instances of these abstractions in the source code.

Paakki et al. [Paa+00] present an approach for the detection of architectural and design patterns. It treats pattern detection as a constraint satisfaction problem (CSP) and uses the AC-3 algorithm [Mac77] to find pattern candidates and then use software metrics to assess the quality of the systems architecture. The approach is implemented in Java but uses a Prolog variant for the representation of architectures and patterns.

[1] http://www.jhotdraw.org/

The approach presented by Di Penta et al. [DP+07] automatically analyzes SOAP messages in execution traces to detect architectural patterns in a SOA system. It is based on model checking, verifying patterns on a model of the system, where patterns are described as mu-calculus logic formulae.

In this subsection we presented a number of approaches focusing on recovering or detecting architectural patterns with a majority of approaches focusing on the automatic detection of patterns. As discussed in detail in Chapter 6, Section 6.1 automatic approaches are limited by a high number of false positives while existing semiautomatic approaches focus either on a specific pattern [Sca+10] or are limited to specific languages [Lun+06]. In contrast, the approach we propose in Chapter 6 focuses on architecture documentation and evolution of architectural patterns. It provides support for pattern variants and does not have the drawback of finding many false positives as it is semiautomatic and requires the software architect to annotate the architecture model with architectural primitive information.

3.2.2 Approaches Based on Design Patterns

In this subsection we describe selected approaches for the identification of design patterns ranging from manual identification techniques to automatic detection approaches.

In the first part we discuss approaches that utilize formal methods like concept analysis [TA99; Are+04] or constraint satisfaction problems [GJ01] to tackle design pattern identification. The second part contains approaches that represent patterns as graphs and or treat the problem of pattern identification as a graph matching problem [Tsa+06]. In the last part we discuss approaches that use various other techniques for the identification of patterns like String pattern matching [PG02] or SQL queries [RM11].

3.2.2.1 Approaches Based on Logic Oriented Programming / Formal Methods

In this section we present a number of approaches that use logic oriented programming or formal methods to identify design patterns in source code. Wuyts [Wuy98] uses declarative reasoning to find structural relationships in Smalltalk programs. He created a declarative framework for describing the structure of an object oriented system that he then uses to describe design patterns. However his approach is not directly focused on reconstructing patterns.

Another approach is presented by Krämer et al. [KP96] that uses Prolog queries on C++ header files to find a number of structural design patterns. However the precision of their approach is only about 40 percent.

Tonella and Antoniol [TA01] introduce an approach for object oriented pattern interference. It utilizes concept analysis to detect design patterns in C++ source code. The authors present three case studies that showcase the approach.

Arévalo et al. [Are+04] also use a formal approach based on concept analysis to detect collaboration patterns between software artifacts. Their approach is language independent and analyzes a system's structure and improves the pattern detection algorithm introduced by Tonella and Antoniol [TA01]. In comparison to the approach we discuss in Chapter 6, their approach works on a lower abstraction level and focuses on class relations, while our approach is on the level of architectural components and focuses on architectural patterns.

Guéhéneuc et al [GJ01] propose a explanation-based constraint programming approach for identifying and correcting micro-architectures that are similar to design patterns. They use a library of constraints to search for design patterns. Their approach provides the benefit of being able to explain why pattern candidates where rejected. Based on these explanations their approach lets the user decide to relax specific constraints if desired and create new solutions.

Alnusair et al. [Aln+13] propose an approach that uses semantic web technologies for the detection of design patterns in source code. They formalize the structure and behavior of patterns using first order predicate logic.

Lucia et al. [DL+10] present an approach based on linear temporal logic that analyzes pattern instances' behavior statically and dynamically. First a set of pattern candidates is computed based on structural information. For the pattern candidates the model checking tool SPIN is used to check if they fulfill the behavior specified in the pattern description using sequence diagrams.

All the approaches in this section utilize formal methods for the identification or the description of design patterns. With the exception of one [Wuy98], all these approaches are automatic and thus have a potentially slow run-time behavior and possibly yield a high number of false positives. In addition, the discussed approaches work on the abstraction level of source code while the approach in this thesis, is semiautomatic and searches for architectural patterns on the level of architectural components, which, compared to the source code level, reduces the search space.

3.2.2.2 Graph-based Approaches

This section presents different approaches for identifying design patterns in source code that represent pattern descriptions or source code relations as graphs or use graph matching algorithms.

An approach that uses operations on finite sets of bit-vectors to detect design patterns is introduced by Kaczor et al. [Kac+06a]. They utilizes the parallelism of bit-wise operations in a bit-vector algorithm that is able to detect exact as well as approximate instances of a pattern. The program

and the patterns are represented as digraphs from which a string representation is computed. These string representations are used as input for the bit-vector algorithm. They present 3 case studies where they search for the Composite and AbstractFactory patterns in three different existing applications.

von Detten [Det11] proposes to use symbolic execution in order to improve the detection of behavioral design patterns which are specified on the basis of UML sequence diagrams. The author integrates his approach into the Reclipse tool.

An automatic approach for pattern-based design recovery in Java was introduced by Seemann and von Gudenberg [SG98]. Their approach is based on graphs and the authors showcase their approach for the detection of the design patterns Composite, Bridge, and Strategy.

Balanyi and Ferenc [BF03] introduce a design pattern description language DPML that is based on XML and allows the description of a patterns structure and behavior. They then use the C++ reverse engineering framework Columbus to create an abstract semantic graph of the software system and compare this graph to the pattern descriptions.

Wendehals et al. [Wen03] present an approach that they use to identify design patterns in Java source code. It uses graph rewrite rules and fuzzy logic to allow for design- and implementation variants of design patterns and utilizes dynamic analysis to improve the recognition of design patterns. The approach confirms the candidates that are found by static analysis using dynamic analysis. The confirmed pattern candidates are then presented to the user.

Tsantalis et al. [Tsa+06] propose a pattern detection approach based on the similarity scoring between graph vertices. Due to the underlying algorithm it supports the recognition of patterns that deviate from the defined structure. However in the presented version of their implementation, pattern descriptions are hard-coded within the tool.

All of the graph-based approaches that we presented in this section work automatically and focus on the identification of design patterns in source code. The approach we describe in Chapter 6 is semi-automatic and focuses on the documentation of architectural patterns as architectural knowledge based on architectural components. It specifically supports consistency checking of documented patterns in order to keep architectural documentation and source code consistent throughout the evolution of a software system.

3.2.2.3 Miscellaneous Design Pattern Identification Approaches

While a considerable number of approaches are based on formal methods or graphs, various approaches have been proposed that use different techniques for the identification of design patterns. Some of these automatic design pattern identification approaches use existing existing query tools

and languages like Dali [Guo+99] or regular-expressions and SQL [RM11; SW08]. While some of the design pattern identification approaches are based on pattern catalogs or similar [Det11; BF03; GJ01], other approaches only support a sub-set of existing design patterns [Phi+03; Kel+99]. Philippow et al. [Phi+03] give an overview over existing automatic design pattern recovery approaches and introduce an approach based on minimal key structures that uses a number of positive and negative search criteria including the possibility of uncertain elements. They implemented the search algorithms for the GoF patterns [Gam+95] Composite, Singleton, and Interpreter. The approach provides support for variability but uses fixed algorithms for the detection of the design patterns. Like already mentioned before all fully automatic approaches require the manual correction of false positives and cannot guarantee a hundred percent detection rate, which both leads to additional manual effort in the architecture recovery process. However the most important difference between all these approaches that focus on identifying design patterns and our approach (Chapter 6) is, that we specifically target the identification and documentation of architectural patterns and thus work on the abstraction level of architectural components while design pattern identification approaches work on the level of source code or source models.

Different approaches try to tackle the problems usually encountered with automatic approaches for the identification of design patterns and try to reduce the number of false positives by taking not only structural but also behavioral information into account, like Heuzeroth et al. [Heu+03], who detect design patterns in legacy code and classify their found pattern candidates based on the evidence they find during static and dynamic analyses. In another direction go Bergenti and Poggi [BP00], who present an interactive design assistant that automatically finds a subset of the GoF patterns [Gam+95] in UML diagrams and produces critiques for the found patterns thus providing suggestions for design improvements. Another recommendation system for design patterns is proposed by Palma et al. [Pal+12]. They implement a design pattern recommender based on a goal-question-metric that focuses on supporting the engineers during the implementation / adaptation of a system. Based on the answers a user provides to their questions, the system suggests the pattern with the highest weight. However in order to recompute the weights and re-evaluate the recommended patterns, a complete re-run of the tools is necessary.

Other approaches aim at aiding the user during the manual identification of design patterns. For example, Shull et al. [Shu+96] proposed an inductive method based on a set of procedures and guidelines to aid the engineers who have no knowledge in the architecture reconstruction process. Washizaki et al. [Was+09] combine automatic searching for design patterns in a system's structure and behavior with additional user input, who needs to specify if the system matches the conditions and smells of a pattern manually.

In contrast to the aforementioned approaches, Pinzger and Gall [PG02] also consider patterns on a higher abstraction level. They use an interactive and iterative architecture recovery approach that is built upon low level patterns that are automatically detected via string-pattern-matching

and creates pattern views. The pattern views are then used to abstract higher level patterns which enable the description of a system's architecture. In contrast to this approach where design patterns are automatically detected in the source code via string-pattern-matching, the approach we present in Chapter 6 allows to semi-automatically define abstractions, and then provides the architect with a number of possible existing patterns. Furthermore we specifically take evolution into account and provide the user with automatic consistency checking and traceability links between documented architectural pattern instances, architectural component views, and the source code during the future evolution of a system.

3.3 Software Architecture Evolution

Since the latest definition of the Lehman's Laws of Software Evolution [Leh96], many empirical studies on their validation have been published. All these studies, as Herraiz et al. summarize [Her+13], validate the first law of evolution, that is, software "must be continually adapted, or else it becomes progressively less satisfactory in use.´´ Mainly, we can consider that systems must evolve in order to continue being useful. This confirms the need of techniques, methods, tools, etc. guiding the different stakeholders in the software evolution process. There are different approaches to follow that facilitate such software evolution. However, several authors, such as [Bar+08; Bar+12; Bre+12; Hol02; NT10], claim that architecting is the most appropriate point of view to address this issue as it enables stakeholders to work at a higher abstraction level.

There is a great deal of attention on architecting for software evolvability, especially since 2008. Several authors [Bre+12; Jam+13] have pointed out that this can be due to the fact that more and more systems turn into legacy systems, so that both practitioners and researchers have realized the importance of software architecture evolution. This attention on software architecture evolution is also shown by the number of literature reviews and mapping studies that have been published during the last five years. Some of them [Bre+12; Jam+13] have analyzed the available architecting practices and have classified them mainly into two different groups: (i) those practices that help stakeholders to design [Tar07] and assess [Cle+01] the evolvability of the software architecture when this is being specified; (ii) those practices (see e.g. [Bar+12; Le +08]) applied to evolve the software architecture. Our approach focuses on the latter category, that is, on providing architects and developers with assistance when both software architecture and code must be evolved.

3.3.1 Techniques for Evolving Architectures

When analyzing the available techniques for evolving architectures, the above-mentioned literature reviews identify a wide spectrum of approaches. There have been approaches, such as [Hun+08; Cor+02], that have analyzed the exploitation of transformation techniques for evolving legacy

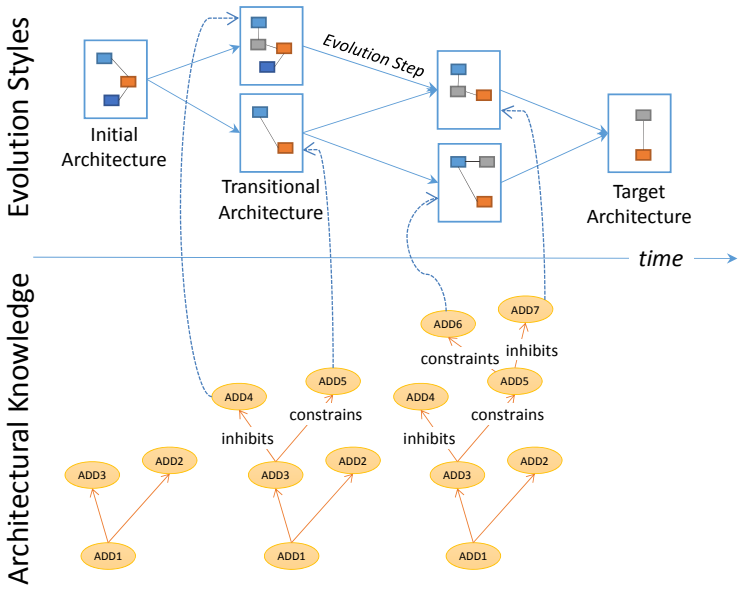

FIGURE 3.1: Evolution Styles and AK: guiding the evolution

systems. Specifically, they transform the architecture, previously abstracted from the code, to apply all the necessary changes and then transform it to code again. Therefore, all the architectural evolution is carried out just in a single step, without keeping the knowledge behind each evolution decision, analyzing the different alternatives during the evolution, etc. This kind of transformations from architectural specifications to code, is not always applicable, especially when the evolution to be carried out entails the development of new code.

A different approach was followed by Tamzalit and others [Le +08; NT10; Tam+06]. They have used the concept of architectural styles as the way to constrain the changes that can be applied to a system being evolved. With this aim, they define these architectural styles as sets of evolution patterns, which are specified as generic transformations. Despite the importance of AK, highlighted by the authors [Le +08; NT10], during the evolution process, no support or assistance is provided for its use. A similar proposal is that presented by Barnes et al. [Bar+12; Bar+13]. They defined Evolution Styles to support software architects while evolving a software system from an initial architecture to a target architecture. As shown on top of Figure 3.1, while applying the approach, several evolution paths can be defined as sequences of evolution steps. As a result of every evolution step a transitional architecture is obtained, that is, a new snapshot of the software architecture. Every one of these evolution steps is specified by means of an evolution operator that determines how the architecture must be modified.

One of the most interesting aspects of this approach is that it provides two different kinds of analysis: path evaluation functions and evolution path constraints. In order to carry out the former analysis, every evolution path is annotated with a list of properties, such as duration or cost, so that the software architects can determine, for instance, the economical or temporal feasibility of an evolution path. On the other hand, and thanks to the automatic support for the

analysis of evolution path constraints [Bar+13], the software architects can determine whether the identified evolution paths satisfy the rules of the evolution style. For instance, they could check that a component is not removed until a wrapper has been developed.

As can be observed, the use of evolution styles helps software architects to plan the evolution of the software architecture that should be subsequently implemented by the developers. Moreover, they can also evolve a system in a way that does not violate any of the rules described in the style. However, so far the relation to code that implements the architecture has been neglected and in particular that the code might not be compliant with the planned software architecture.

Garlan et al. [GS09] propose a tool called Ævol, that supports the definition and planning of architecture evolution based on Evolution styles. It allows the specification of evolution paths and the comparison of alternative evolution paths regarding correctness conditions and cost-benefit. While their approach has a similar focus, it requires the architect to specify all possible evolution paths, whereas the approach we present in Chapter 9 only requires the definition of the necessary evolution tasks and their dependencies, being generated automatically all possible decisions for executing the tasks.

McVeigh et al. [McV+11] proposed Evolve, a model driven tool that captures incremental change in the definition of software architecture. It implements Backbone, an architectural description language with a graphical (UML2) and a textual representation. Based on the architecture documentation, backbone directly constructs initial implementations and extensions to these implementations. While their approach supports the evolution of a system's architecture, the approach we propose in Chapter 9 focuses on managing the complexity inherent to evolution and provides valid plans for executing the tasks that arise during the evolution of a system.

An approach called ADVERT is proposed by Konersmann et al. [Kon+13]. Their approach provides support for evolution on an architectural level by maintaining tracelinks between requirements, design decisions, and architectural elements and also including software architecture information into the source code. However, their approach does not consider planning the evolution and thus has a focus different from our approach. In addition, their approach was only partially implemented, being other elements assumed to work and only described for EJBs. However, the approach we suggest in Chapter 9 is programming language independent.

Grunske [Gru05] proposes an approach for architectural refactoring based on a hypergraph-based data structure that allows the formalization of refactorings as hypergraph transformation rules that can be applied automatically. However, unlike the approach we present in Chapter 9, this approach does not support evolution in general but is limited to behavior preserving architectural refactoring.

3.3.2 Managing Architectural Knowledge

Regarding the evolution of the SA, a second important aspect to be considered is how it became specified in its latest form. It is important to know not only which or how many components and connectors it has, or how they are related among themselves, but also it is important to know what decisions have been made and why, that is, the rationale [PW92b] behind the architecture specification. During the last decade, a great deal of attention has been paid to the field of Architectural Knowledge (AK), since Bosch [Bos04] recalled the attention to this important aspect. Unfortunately, too frequently, this AK vaporizes as architects fail to describe it in a proper way. As Harrison et al. state [Har+07], this vaporization can have critical consequences, the expensive system evolution being one of them.

Several empirical studies support this argument. For instance, Bratthall et al. [Bra+00] carried out a survey with 17 subjects from both industry and academia and concluded that most of the interviewed architects considered that by using AK they could shorten the time required to perform change tasks. The interviewed subjects also concluded that the quality of the results, when they had to predict changes on unknown real-time systems, was better using AK. Ozkaya et al. [Ozk+10a] have also concluded, during their interview study, that the difficulties during both the initial phases and the evolution of systems are not only due to the unavailability of AK, but also depend on its ineffective use. This is especially noteworthy because, despite this unavailability and ineffective use of AK, the interviewed architects have also remarked that they perceive AK to be essential when evolution hits. Feilkas et al. [Fei+09] have carried out a case study on three industrial information systems which had the importance of AK as one of its research questions. The authors have detected that one of the problems in these projects was that developers were not aware of the intended architecture because the AK was not properly described.

On the other hand, other empirical studies [Ahm+14; Din+14; Li+13; Tan+06; Tof+14] have focused their attention on analyzing available AK practices. As one of their main results all of them emphasize its usefulness as a valuable artifact for the software architecture evolution. For instance, some works [Boe+09; Pah+09] have exploited an ontology-based approach to describe AK that can be used for impact analysis by means of if-then scenarios. Another approach called Architecture Rationale and Element Linkage (AREL) [Tan+07] is basically a language used to describe causal relationships between Architectural Design Decisions (ADDs) and architectural elements. Using these descriptions, the approach exploits Bayesian Belief Networks to carry out three different probability-based reasoning methods that enable architects to estimate the impact of a change before it is carried out. Other interesting work is a family of languages called ACL [Tib+10] that enable software architects to relate architectural choices to decisions that are made throughout the development process. One of the main strengths of the approach is that these decisions are defined as constraints, so that the architectural model can be checked after its evolution to determine its conformance. Another approach [Nav+13] proposes the use of anti-patterns to detect and record

conflicting ADDs while the architecture is being evolved. However, as can be noticed, none of these approaches really considers the evolution of the architecture, but just the use of AK for analysis purposes.

Recently Cuesta et al. [Cue+13] have presented a proposal called AK-driven evolution styles (AKdES), which focuses on the bottom part of Figure 3.1. Basically, the authors claim that every evolution step is carried out because an evolution condition has been triggered. Then an evolution decision, that must be stored as part of the AK as an ADD, has to be made as reaction to the evolution. This evolution decision leads to an evolution step that must be carried out. It is worth noting that all these ADDs as well as the relationships among them must be recorded in order to have all the history of the evolution of the system properly described, so that the AK can be used to avoid the previously sketched problems in the evolution process. As a result, ADDs are used to convey the ideas behind each evolution step to both software architects and developers. However, the already stated misalignment between code and SA can also happen, despite of considering such AK as part of the evolution process, because misunderstandings, as well as architectural erosion [Ter+13] or architectural drift [Ros+11], can, accidentally or not, occur. The approach presented in Chapter 8 aims to address this open issue.

3.3.3 Approaches That Focus on Traceability and/or Change Impact Analysis

Feng and Maletic [FM06b] present an approach to analyze the impact of changing components at runtime based on slicing on component interaction traces. Their dynamic component composition model is based on a static model and a set of UML sequence diagrams. While their approach focuses on the analyis of the impact of dynamic changes during runtime, in Chapter 5 we primarily focus on providing architectural abstractions from source code to architectural component views during software evolution and thus focus on changes at compile time between different evolution steps of a software system.

Another approach that focuses on change impact analysis is presented by Zhao et al. [Zha+02]. They present an automated approach that uses architectural slicing and chopping to analyze formal architectural specifications based on WRIGHT [AG97]. While this approach uses an architectural specification as an input, in Chapter 5, we focus on semi-automatically providing this architecture abstractions.

Knodel et al. [Kno+06] give an overview on how and when static architecture evaluations can contribute to architecture development and show how architecture development is influenced by architecture evaluations in the area of software product lines. While the approach from Chapter 5 targets the maintenance and recovery of architecture information in general, the approach introduced by Knodel et al. [Kno+06] focuses on evaluating a system's architecture regarding software product lines.

A wide variety of work has been done in the field of traceability: i.e. traceability of concerns between architectural views [Tek+07], to linking design decisions to design models [KZ10a]. Winkler and Pilgrim [WP10] present a survey on the state of the art of traceability in requirements engineering and model-driven development who base their work amongst others on the works by Spanoudakis and Zisman [SZ05] who, based on extensive literature study, define eight different kinds of traceability links, discuss a number of approaches for the generation and maintenance of traceability links, and present a number of ways how traceability relations can be used in software development.

3.4 Understandability of Software Architecture Documentation

In this section we give an overview of the current state of art regarding understandability metrics as well as other approaches that utilize metrics in a similar way.

Many different software metrics for measuring the system's architecture, components as its constituting parts, and structures similar to architectural component views, such as other higher-level software structures (packages, graph-based structures) have been proposed. Metrics related to components and the corresponding architectures [Kan+08; Sha+09; Sar01] measure size, coupling, cohesion, and dependencies of individual components but also the complexity of the whole architecture when all the components and their interactions are taken into account. Different authors have proposed different package level metrics that measure their size, coupling, stability, and cohesion [Mar03; GC12]. Graph-based metrics measure different interactions between the nodes in the graph [Bha+12; Ma+06]. Some of the graph-based metrics have been shown to be useful in measuring large scale software systems in the sense that those systems share some properties that are common for complex networks across many fields of science [Ma+06]. All these mentioned metrics can be applied or can be more or less easily adapted to be applicable for architectural component views. However, none of the metrics is empirical evaluated regarding the understandability of architectural components or architectural component views so far. In the software architecture literature we find only very few studies that provide empirical evidence regarding the architectural understandability or the measurement of architectural understandability (see e.g. [GC09; Eli10]). The empirical studies in our group by Stevanetic et al. [Ste+14a] try to provide more evidence in that context. While the main focus of this thesis is the creation and maintenance of software architecture documentation during software evolution, the integration of understandability metrics with the architecture documentation approach we propose in this thesis in Chapter 7, can be seen as a starting point for future work on utilizing understandability metrics and additional studies on the effect of understandability metrics on the architecture and source code quality are necessary.

3.5 Empirical Studies Researching Software Architecture and Design Understanding

In this thesis, as already discussed in Section 2.2, we have performed a controlled experiment that lies the foundation for this thesis in Chapter 4, as it shows, that architectural component diagrams can be helpful in aiding the architecture understanding. Based on this knowledge we propose the use of architectural component views for documenting a system's architecture in Chapter 5 and build upon these architectural component views in Chapter 6 for the identification and documentation of architectural patterns. We keep the focus of this section on controlled experiments and case studies that provide contributions to software architecture and design understanding, while there exists a huge number of case studies in the literature that have been performed to showcase the applicability of specific approaches, that we do not discuss in this section.

The general notion of empirical studies in software architecture has been studied by Falessi et al. [Fal+10]. They conclude from their study that a greater synergy between empirical software engineering and software architecture would support the emergence of a body of knowledge consisting of more widely accepted and well-formed theories on software architecture and the empirical maturation of the software architecture area.

In Section 3.5.1 we discuss different studies that focus on architecture design. There also exist a number of studies that focus on different aspects of components ranging from the optimal component size to the reuse of components, which we discuss in Section 3.5.2. A number of works that present studies and approaches with the topic of design understanding in general are presented in Section 3.5.3. Section 3.5.4 discusses multiple empirical studies on different aspects of understandability of UML diagrams. Finally a number of approaches that study the helpfulness of traceability links between different artifacts are presented in Section 3.5.5.

3.5.1 Empirical Studies Related to Architecture Design

Only a few of the empirical studies in the area of software architecture are directly related to architecture design or design understanding. Boucke et al. [Bou+10] introduce an approach that explicitly supports compositions of models, together with relations among models in an architecture description language. In an empirical study they show that their approach reduces the number of manually specified elements and manual changes.

van Heesch et al. study in two surveys the reasoning process of architects, one with students [HA10] and one with professionals [HA11]. A related study performs a controlled experiment about the supportive effect of patterns in architecture decision recovery [Hee+12].

Many empirical studies in the field of software architecture study other aspects, like quality aspects or other views. For instance, a number of studies related to evaluating architectures have been conducted. Barbar et al. [Bab+07] performed an empirical study aiming at understanding the different factors involved in evaluating architectures in industry. The influence of software visualization on source code comprehensibility was studied by Umphress et al. [Ump+06] based on control structure diagrams and complexity profile graphs. Biffl. et al. [Bif+08] study the impact of experience and team size on the quality of scenarios for architecture evaluation. A number of empirical studies aim at better understanding the relation of architecture and requirements [Fer+10; Mil+10]. Various empirical studies relating architecture to certain qualities or metrics have been conducted. For example, Hansen et al. study the relation of product quality metrics and architecture metrics [Han+11].

Heijstek et al. [Hei+11] conducted a controlled experiment on whether visual or textual artifacts are more effective at conveying architectural knowledge with 47 students and software engineering professionals. In their experiment, they found no particular advantage of visual architectural documentation compared to textual documentations. While we, in the controlled experiment conducted in this thesis, used source code in the control group and source code aided by component views in the experiment group, they did not use the source code at all for their experiment.

3.5.2 Empirical Studies Focusing on Other Aspects of Components

Even though we found no rigorous empirical studies of the effects of architectural component views on understandability so far, aspects like reuse or fault density of components have been studied empirically before. Fenton and Ohlsson have studied the relations of fault density and component size in a large telecom system [FO00]. Mohagheghi et al. provide a study comparing software reuse with defect density and stability [Moh+04]. Their study is based on historical data on defects, modification rate, and software size. These metrics are compared to the reuse rates in a telecom system. Mohagehghi and Conradi [MC08] also present a case study on the benefits of software reuse in a large telecom product. In this study they evaluated and explored reuse benefits and found that significant benefits in terms of lower fault density and code modification. They also found that reuse reduced the risks and lead time of the second product as well as that reuse can help reduce organizational restructuring risks.

Malaiya and Denton provide an analysis of a number of studies and identify the component partitioning and implementation as influencing, competing factors to determine the “optimal” component size with regard to fault density [MD00]. Graves et al. have studied the software change history of components to create a fault prediction model [Gra+00]. Metrics such as time elapsed since the last changes, change times, and number of changes were used in the model, while size and complexity metrics were not deemed useful. Our experiment and these studies have in common,

that they make a link between component views and software quality, but in contrast to our experiment they only study aspects that can solely be studied using the software systems and their historical data. In contrast, we consider the (novice) architect's perception of understandability as well as expert opinions on their results. In addition, in those other studies, components are understood as implemented software modules, rather than architectural abstractions.

3.5.3 Studies and Approaches on Design Understanding

There are a number of research directions related to improving design understanding of software architectures. In particular, regarding architectural component views, software architecture literature suggests that architectural component views are central architectural abstractions [Cle+02], and hence it can be assumed that they have a positive effect on the architectural understanding of software systems (see e.g. [RW05; Cle+02]). A number of approaches suggest that other aids are needed to gain a better understanding of the design or architecture, such as architectural views [ISO11; Cle+02; Hof+00] or architectural decision models [TA05; Kru+06; Zim+09], which would contain or augment component views. Both research directions only focus on complementing component views with additional knowledge, but do not research on the effects of the component views on the understandability of a software architecture or design but neither of them can fully resolve understandability issues related to the component views themselves. Other literature suggests that it might be hard to understand the source code only with models, and traceability links [Iee] between components and code are needed to make the connection [Gal11].

Hungerford et al. [Hun+04] present a study on how 12 software developers use entity relationship diagrams and data flow diagrams for defects search. They found that search strategies and defect detection rate varied among the participants. Their results indicate that strategies that rapidly switch between the two design diagrams are the most effective.

3.5.4 Studies Focusing on UML Diagram Understandability

A number of papers focus on the comprehension of UML diagrams. Some focus on dynamic models [OD04], while others focus on specific diagrams or models like sequence diagrams [Gen+08], state charts [CL+09], or class diagrams [Pur+01]. The influence of the level of detail in class and sequence diagrams on the maintainability of software has been studied by Fernández-Sáez et al. [FS+12]. In a paper by Fernández-Sáez et al. [FS+12] the effect of UML models with high and low level of detail on software maintainability are studied. However no statistically significant result could be obtained due to various factors. Only a slight tendency towards a better results with low level of detail was found.

Ricca et al. [Ric+07] report the results of three experiments they performed. In the experiments they studied the effectiveness of UML stereotypes for Web design with regard to various comprehension tasks for different subjects. Their results show that Web design stereotypes are particularly helpful to subjects with low experience. However they did not find a significant effect on the mean performance, as the performance of experienced subjects did not change significantly.

Another set of experiments regarding UML stereotypes was performed by Staron et al. [Sta+06] with students and participants from industry. They evaluated the role of stereotypes in improving the comprehension of UML models. Their results show that stereotypes play a significant role in the comprehension of models achieved by students as well as by professionals.

Tilley and Huang [TH03] performed a qualitative assessment of the efficacy of UML 1.4 diagrams as graphical documentation in aiding program understanding. In their experiment they asked participants to analyze a series of UML diagrams and answer a questionnaire on a hypothetical software system. They identified three limiting factors for the efficacy of UML class diagrams: ill-defined syntax and semantics of UML 1.4 itself, the role of spatial layout in fostering program understanding, and the importance of domain knowledge in supplying necessary information to aid the software engineer.

Torchiano [Tor04] studies if UML static object diagrams can improve program comprehension. He asked 17 graduate students to answer questions about a software system which was either described with a class diagram or with both a class diagram and an object diagram. The results show a significant improvement for the experiment group for some, but not all tasks.

Otero and Dolato [OD04] evaluate the comprehension of designs in behavioral UML diagrams. In this study, they conducted two experiments in which they assessed the time spent and the scores obtained for answers to a questionnaire for different diagrams and different application domains. In their studies they found that for the different application domains, different combinations of diagrams support the comprehension best.

A number of studies have been conducted that focus on the effects of different UML models on the understanding of a software system. An industrial case study focusing on the impact of UML on software quality [NC09], as well as a controlled experiment studying the impact of OCL in combination with UML find that the usage of UML (and OCL) improves the detection rate of defects as well as program comprehension and impact analysis. While these approaches are similar to our controlled experiment in their intent to study the impact of additional models, they do not study the effect of architectural component diagrams in particular as they focus on the effects of other UML models, or on comparing the effects of different diagrams while we study whether (UML) component diagrams are beneficial to architecture understanding when provided to the experiment group in addition to the source code.

3.5.5 Studies Focusing on Traceability Links

A number of different studies examine the effects of traceability links on design understanding. Mäder and Egyed [ME12] investigate the usefulness on traceability links between requirements and the source code. In their experiment 52 students were given eight maintenance tasks on given software projects. For half of the tasks traceability information was provided. Their results show that the subjects found solutions faster and found better solutions when provided with traceability information.

In a controlled experiment and its replication De Lucia et al. [De +09] assessed the usefulness of an Information Retrieval-based traceability recovery tool during the link identification process. In these two experiments they found that using the tool ADAMS [Bru+06] significantly reduced the time spent by the participants to identify links. However they observed differences between participants with high and low experience. It seems that participants with lower experience profited more as their achieved results came closer to the results achieved by participants with more experience.

In two other controlled experiments with bachelor and master students, De Lucia et al. [DL+11; DL+06] investigate the quality of identifiers and comments produced when using COCONUT. COCONUT is an Information Retrieval-based traceability recovery tool implemented as an Eclipse Plugin that recommends candidate identifiers built on high-level use case information. Their results indicate an improved quality of comments and variable names when COCONUT is used.

Two controlled experiments investigating the effects of the IR-based traceability recovery tool SCOTCH [Qus+11] were conducted by Qusef et al. [Qus+12]. They asked 32 bachelor students to perform traceability recovery tasks for two different systems in which the students had to find links between test code and source code and compared the results to two base-line techniques (NC and LCBA). Their results indicate an improvement when using SCOTCH over the base-line techniques as participants identified a higher number of correct links with higher accuracy.

These studies show the positive effects of traceability links for finding faster and better solution to given tasks. As such, these studies have a different focus then the one presented in Chapter 4, but based on these findings, we ensured that the approach we present in Chapter 5 does not only support the easy creation of traceability links between architecture and code, but our prototype also provides them in a navigable manner in order to make switching between architecture and code as simple as possible.

Part II

Supporting the Architect During Evolution: Semi-automated Architectural Component Model Abstraction and Pattern Identifaction

4 Controlled Experiment on the Supportive Effect of Architectural Component Diagrams for Design Understanding of Novice Architects

4.1 Introduction

Today a software architecture description is usually comprised of multiple views [ISO11; Cle+02; Hof+00]. The component and connector view (or component view for short) of an architecture is a view that is often considered to contain the most significant architectural information [Cle+02]. This view deals with the components, which are units of runtime computation or data-storage, and the connectors which are the interaction mechanisms between components [PW92a; Cle+02]. An architectural component view is a high-level abstraction of the entities in the source code of the software system, as the software architecture concerns only the major design decisions about a software system, and abstracts from irrelevant details [JB05].

While much research work has been done in component-related research areas such as modeling languages for component and connector models, component implementation technologies, component composition, and the formal semantics of components, only a very few rigorous empirical studies relating to the use of component views in architectural descriptions of software systems have been conducted. However, such foundational research is essential to provide guidelines and tools to software architects, based on sound evidence, to help them understand how to design component views that are appropriate for the architectural understanding of a software system.

We use the empirical data gathered in this controlled experiment as foundation for our other research work that we present in this thesis, especially the approach for creating architectural component views that we present in Chapter 5.

The goal of the experiment was to find an answer to Research Question 1 and determine whether architectural component diagrams, provided in addition to a non-trivial software system's source code, have a supportive effect on the ability of novice architects to answer design and architecture related questions about that system. This goal is interesting to study, as today it is unclear whether

component diagrams alone are sufficient to help architects to understand complex architectural relationships in a given system in a better way than just studying the source code of that system. While the software architecture literature suggests a supportive effects of component diagrams (see e.g. [RW05; Cle+02]) for design understanding, there is little empirical evidence so far.

In addition, many existing approaches seem to assume seasoned architects as their main target group. Assuming that component diagrams alone are a useful help to gain a better architectural understanding of a system, as some of the software architecture literature suggests, it is unclear whether this effect can also be observed for novice architects. As software architecture has the goal to convey the big picture of a software system and novices who start on a new project especially require help to gain such a big picture quickly, it is highly interesting whether there is indeed a supportive effect on design understanding for them. Hence, in our experiment we particularly focus on novice architects with medium programming experience.

The experiment presented in this chapter studies the experiment goal by letting 60 students with medium programming experience answer seven questions about the design and architecture of a given software system (the computer game FreeCol). One half of the participants, the control group, received the source code of that system as the main source of information, while the other half of the participants, the experiment group, additionally received architectural component diagrams for FreeCol. The answers of the participants were rated by independent analysts. By showing that the quality of the answers improves for certain questions, our study provides initial evidence on how architectural component diagrams help in understanding the design and architecture of software systems. The results indicate that architectural component diagrams are especially useful if a direct link from the component diagram's elements to the problem that requires understanding can be made. That is, if the component diagrams provide some guidance or help to answer a question. In these cases, component diagrams indeed have a supportive effect for software design and architecture understanding. In contrast, if no such direct link can be made, we found evidence that it should not be assumed that architectural component diagrams help in design understanding, for instance only by providing a big picture view or some general kind of orientation.

In addition to the empirical study, the participants were also asked to complete a short questionnaire after they completed the study. In this questionnaire we asked all participants for a self-evaluation regarding their understanding of the given software system. In addition we asked the experiment group if they found the component diagrams helpfull for the given task.

This chapter is organized as follows: In Section 4.2 we introduce our experiment, including the goal, the hypotheses, the parameters and variables, the experimental design, and the execution. The following Section 4.3 describes the statistical analysis and the testing of the hypotheses. In Section 4.4 we present the results from the participants' self-assessment. Section 4.5 provides the validity evaluation. Finally, Section 4.6 concludes and discusses possible future research directions.

4.2 Experiment Description

For the design of the experiment we followed the guidelines by Kitchenham et al. [Kit+02] and Wohlin et al. [Woh+12]. In our experiments, the guidelines by Kitchenham et al. were primarily used in the planning phase of the experiments, while the advice by Wohlin et al. was used as a reference for the analysis and interpretation of the results.

4.2.1 Goal and Hypotheses

As already mentioned in Chapter 2, the goal of the experiment was to determine whether architectural component diagrams, provided in addition to a non-trivial software system's source code, have a supportive effect on the ability of novice architects to answer questions about the design and architecture of that system. Depending on the question asked, the guidance or help provided by architectural component diagrams can vary greatly. The two extreme cases are that component diagrams readily provide the answer without any need to study other information (like the source code) and that component diagrams provide no clue for answering the question. Intentionally we left out these two extreme cases and studied the shades of grey in between. In particular, we further distinguished the following three types of questions in our experiment:

- *QT*1: A question about the software system's design and architecture for which the component diagrams provide some guidance or help, but the information in the component diagrams alone is not enough to answer the question fully. This means that a question is assigned this question type if the following criteria are fulfilled:
  - The question focuses on one or more classes (objects) that are relevant for the architecture of the system.
  - The source code elements that are referenced in the question have the same or a very similar name as an element (component, connector, or interface) in at least one of the provided component diagrams.
  - The answer to the question is at least partially provided by the component diagrams, for example, because the component diagrams provide architectural information that is required to answer the question.
  - Finding the answer to this question without component diagrams requires substantial effort as multiple classes in the source code have to be understood in order provide a correct answer. For this experiment we classified a source code study as substantial effort, if we estimated that it would take longer than 5 minutes to find the answer.
- *QT*2: A question about the software system's design and architecture for which the component diagrams provide some guidance or help, but the same information is easily visible from

the source code. This means that a question is assigned this question type if the following criteria are fulfilled:

  - The question focuses on one or more classes (objects) that are relevant for the architecture of the system.
  - The source code elements that are referenced in the question have the same or a very similar name as an element (component, connector, or interface) in at least one of the provided component diagrams.
  - The answer to the question is at least partially provided by the component diagrams. E.g. because the component diagrams provide architectural information that is required to answer the question.
  - Finding the answer to this question in the source code without the component diagrams requires only a limited effort (a source code study for which we estimated that it would take less than 5 minutes to find the answer to the question).

- $QT3$: A question about the software system's design and architecture for which the component diagrams provide no direct guidance or help, only vague orientation in related components and connectors; digging in the source code is required for answering the question. This means that a question is assigned this question type if the following criteria are fulfilled:
  - The referenced source code elements cannot directly be connected to any element in any of the component diagrams.
  - The component diagrams provide no help in answering the question by providing additional architectural information.
  - The focus of this question is not architecture related but rather related to the design or implementation of the system.

The experiment goal has led to one null hypothesis that we tested for each question type:

Hypotheses

We postulate the following hypothesis about the effects of architectural component diagrams (in addition to the source code) on the quality of answers that novice architects provide to questions about a software system's design and architecture.

With respect to the quality of the answers given by the control group and the experiment group,

- the null hypothesis is that the quality does not improve, $H_0 : \mu \leq \mu_0$;
- the alternative hypothesis is that the quality improves, $H : \mu > \mu_0$.

We tested this hypothesis for the each of the question types separately and expect differences in the results.

Expectations

Our expectations for the results of the three tests are:

- For design questions of Type $QT1$, we *expect that the null hypothesis can be rejected*. That is, component diagrams have a supportive effect on the answers that novice architects provide to questions about a software system's design and architecture, if the component diagrams provide architectural guidance for answering the question.
- For design questions of Type $QT2$, we expect that the *null hypothesis can not be rejected*. That is, component diagrams are helpful, but that novice architects with medium software development experience are able to see the same information in the source code, if it is easily visible. However, this expectation might be wrong as possibly the visual information in the component diagrams might be more readily accessible to novice architects than the easily visible information in the source code.
- For design questions of Type $QT3$, we expect that the *null hypothesis can not be rejected*, as there is no direct relation between the question and the additional information provided by the component diagrams. This expectations might be wrong however. The insight gained through component diagrams might have an indirect supportive effect, for instance by providing some kind of general orientation that helps in answering this type of questions.

4.2.2 Parameters and Variables

Dependent Variable

One dependent variable was observed during the experiment, as shown in Table 4.1: the quality of the answer to the question. In this context, we see the quality of the answer as the degree of insight that is shown in the given answer. The quality of the answers was assessed by three independent software architecture researchers with multiple years of practical software development and architecture experience (later also referred to as analysts) using an interval scale, ranging from 0 (worst) to 10 (best). The interval scale nature of the rating system was explained to the analysts before their analysis (i.e., that equal distances between the points on the scale can be assumed), as this is important for applying parametric statistical tests [Ste46]. The analysts were assigned per question, and each analyst rated each of the assigned questions for all participants in both experiment and control group, to make sure that each question is homogeneously assessed (e.g. the

first analyst rated Question 1 for all participants). Two analysts rated two of the questions, and one analyst rated three of the questions. Each of the analysts studied the software system used in the experiment before their analysis in depth. A catalog of reference answers that was agreed upon by the analysts was created before the evaluation to ensure fair evaluation. In addition to this the analysts were asked to review the ratings given by their colleagues. The ratings were left to the analysts' own experience and interpretation, but they were asked to specifically take the displayed architectural understanding into account. The analysts were *not* part of the authorship of this work and had no knowledge on the participants' identities or whether a participant was part of the control group or experiment group. The participants of the experiments were also reminded before the beginning of the experiment that they should focus on the architectural dimension of the questions.

Independent Variables

Table 4.1 shows the factor (the participants group) that we changed in order to observe its effect on the dependent variable (the quality of the answers). The treatment for the control group was to perform the task based on the source code, while the treatment for the experiment group was that the task was performed based on the source code and component diagrams.

Table 4.1 also shows a number of other independent variables that could influence the dependent variable. They mainly concern characteristics and previous experiences of the participants. We observed these variables to ensure similar prior experiences in the control and the experiment group to prevent the prior experiences from influencing the results of this experiment.

In addition to the prior experience, the size of the system used in the experiment might influence the results of the experiment. We eliminated this variable by only asking questions about one system.

4.2.3 Experiment Design

To test the hypotheses, we conducted the experiment in the context of the Software Architecture course at the Faculty of Computer Science, University of Vienna in spring 2012.

Subjects

The subjects of the experiment were 60 students of the Software Architecture course. The subjects were randomly assigned into two equally sized groups of 30 students: experiment group and control group. The software architecture course is in the 4^{th} semester or later of the bachelor computer science program. In the software architecture course, prior to the experiment, all participants

Type	Description	Scale Type	Unit	Range
Dependent Variable	Quality of the answer to design question	Interval	Points	0 (worst) to 10 (best)
Independent Variable (Factor)	Group	Nominal	N/A	Possible values (treatment): *experiment group* (source code and component diagrams), *control group* (source code only)
Other (observed)	Programming experience	Ratio	Years	Positive natural numbers including zero.
Independent Variables	Programming experience in Java	Ordinal	Years	4 classes: 0, 0-1, 1-3, >3
	Programming experience	Ordinal	Years	4 classes: 0-1, 1-3, 3-6, >6
	Commercial programming experience in industry	Ordinal	Years	4 classes: 0, 0-1, 1-3, >3
	Experience in programming computer games	Ordinal	Years	4 classes: 0, 0-1, 1-3, >3
	Result in previous assignments	Interval	Points	0 (worst) to 34 (best)
	System size	Ratio	Lines of source code	Positive natural numbers without 0.

TABLE 4.1: Observed Variables

learned to model and read component diagrams with multiple practical assignments and had to perform an advanced implementation task. All of the participants have taken at least a basic programming course prior to the Software Architecture course. 87 percent of the students in the control group and 90 percent of the students in the experiment group have taken a course on object oriented modeling where they also learned UML. 17 percent of the students in the control group and 27 percent of the students in the experiment group have also attended a course on distributed systems design and programming.

Objects

The basis of the experiment is the source code of the Freecol computer game[1], an open source version of the classic computer game Colonization (a turn-based strategy game) with multi-player support, implemented in Java.

We used the tool SonarQube[2] to measure the size and complexity of FreeCol and the results of this analysis are shown in Table 4.2. Complexity measurement in SonarQube is based on the branches in the control flow of a system[3].

Both experiment group and control group were provided with access to the complete source code of Freecol. In order to avoid any bias caused by participants having in-depth knowledge of complex Integrated Development Environments (IDEs) or code editors, source code access was only

[1]See http://www.freecol.org/.
[2]See http://www.sonarqube.org/.
[3]See http://docs.codehaus.org/display/SONAR/Metrics+-+Complexity.

Artifact	Count
Lines of source code	95425
Lines	175689
Statements	47032
Classes	850
Functions	6797
Accessors	980

Avg. complexity per Artifact	Complexity
Function	3.5
Class	28.4
File	39.3

TABLE 4.2: Results of the code quality analysis for FreeCol with the open source tool SonarQube(Version 4.3)

provided using the Browser-based code navigation tool that is integrated with our locally hosted installation of Gitorious[4]. While all participants are familiar with one of the different existing IDEs, participants having an in-depth knowledge of the IDE used in the experiment might have introduced a bias.

Instrumentation

The participants of both groups received the following materials: The Browser-based access to the source code of Freecol was provided in a Lab environment on prepared computers. All other materials were provided on paper. The participants received a questionnaire about the independent variables regarding the participants' experiences. Both groups also received 7 different questions about the design and architecture of Freecol (see below). In addition, the experiment group received an additional document with the following 6 UML component diagrams:

- FreeCol Architecture Overview (Figure 4.1): This diagram gives an overview of the FreeCol components on a very high abstraction level. It shows the three executable parts Client, Server, and MetaServer and how these parts are connected. In addition it visualizes that the Client and the Server both make use of a (partly) common Model.
- FreeCol Server Architecture (Figure 4.2): This view shows the architecture of the FreeCol server in more detail. It depicts that the FreeColServer component handles the communication over the network. Received messages are then forwarded to the ServerController via InputHandler. It shows that the Server has its own Model that uses an externally provided model and that the AI component is integrated into the FreeCol server using an AIPlayer interface.
- FreeCol Client Architecture (Figure 4.3): This view shows the architecture of FreeCol's client in more detail. The graphical user interface (GUI) component uses ActionHandlers and Actions to communicate to the ClientController which updates the Model and communicates with the Server using the IServerAPI provided by the Clients network implementation that

[4]See `http://www.gitorious.org/`.

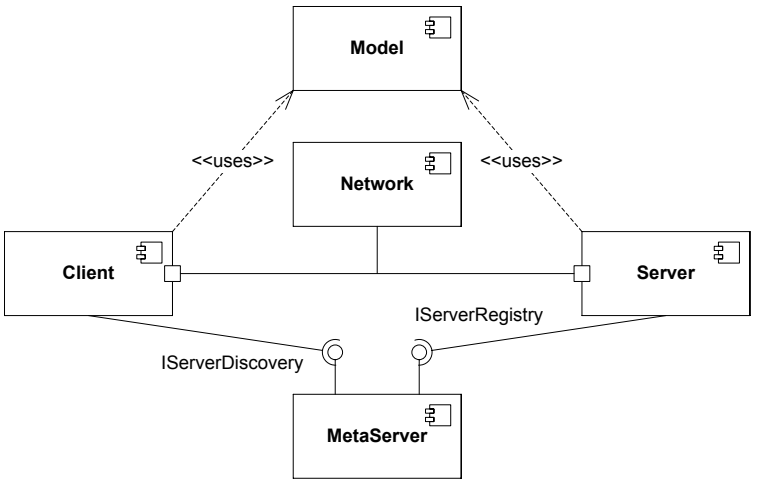

FIGURE 4.1: Freecol architecture overview

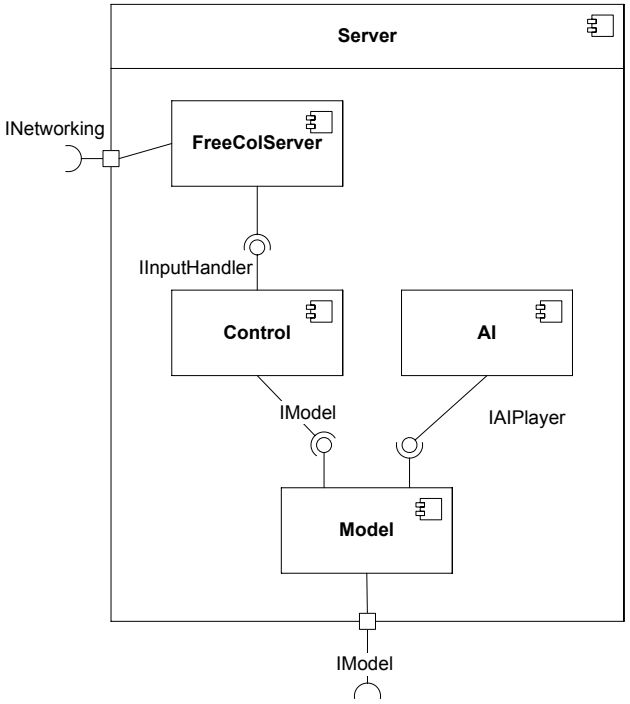

FIGURE 4.2: View showing the architecture of the FreeCol server

is encapsulated in FreeColClient. Furthermore it shows that the client has two components for playing audio and video files that are used by the GUI.

- FreeCol MetaServer Architecture (Figure 4.4): This view shows the two main components of the MetaServer: the registry component that provides the functionality to (de)register and browse open servers and the NetworkHandler that implements the network protocol that is used for server discovery and server (de)registration.

- Detailed View: FreeCol Server – Control (Figure 4.5): This view is on an even lower abstraction level than Figure 4.2. It shows the detailed inner workings of the ServerControl component.

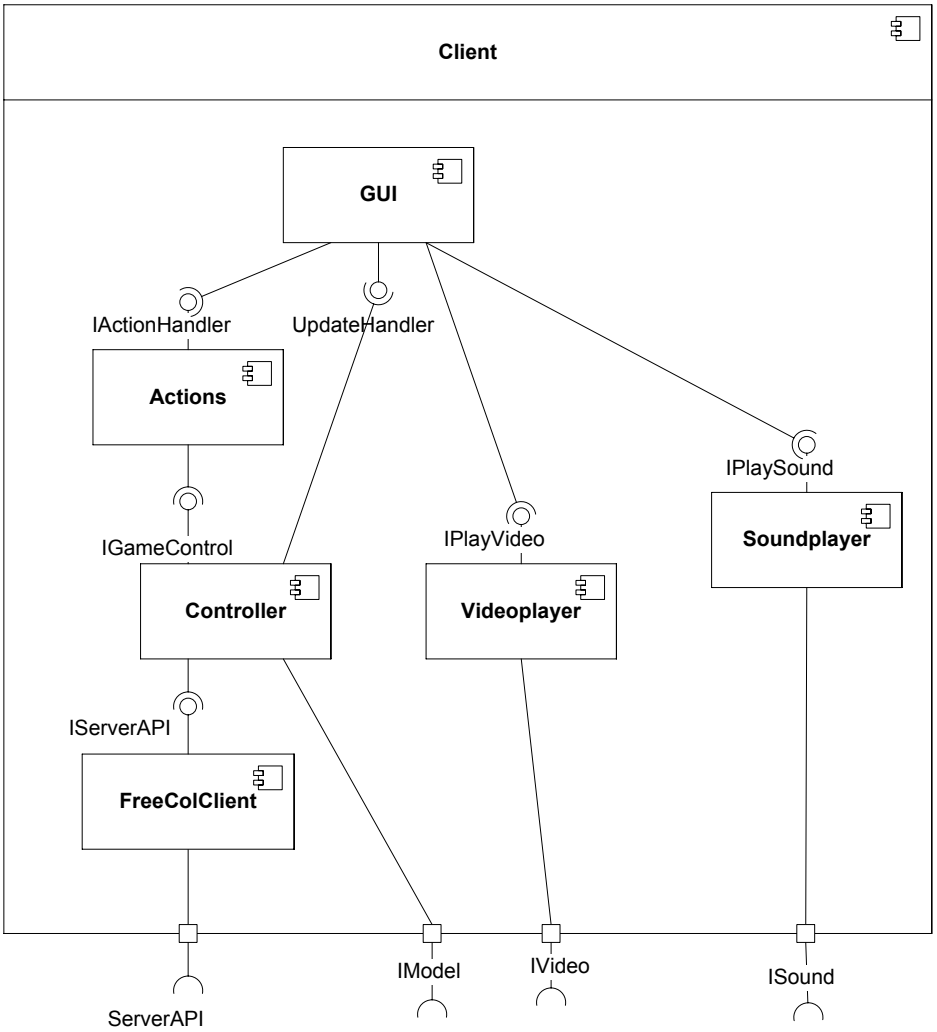

FIGURE 4.3: View showing the architecture of the FreeCol client

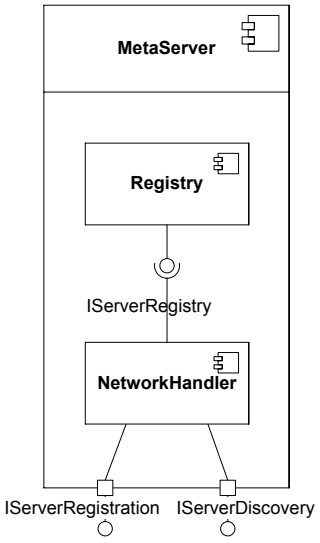

FIGURE 4.4: View showing the architecture of the FreeCol meta-server

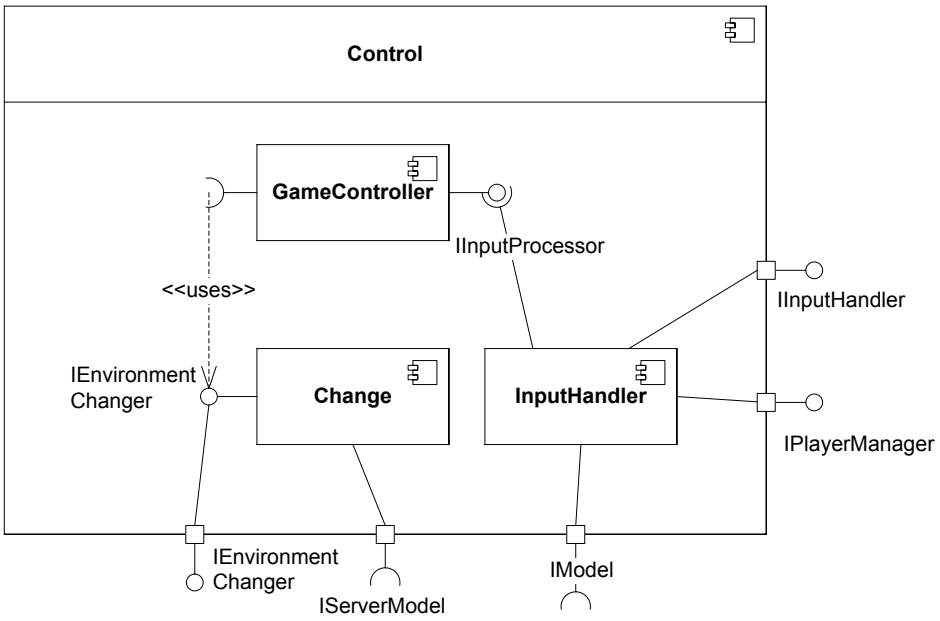

FIGURE 4.5: Detail view for the FreeCol Server showing the server-side Control component

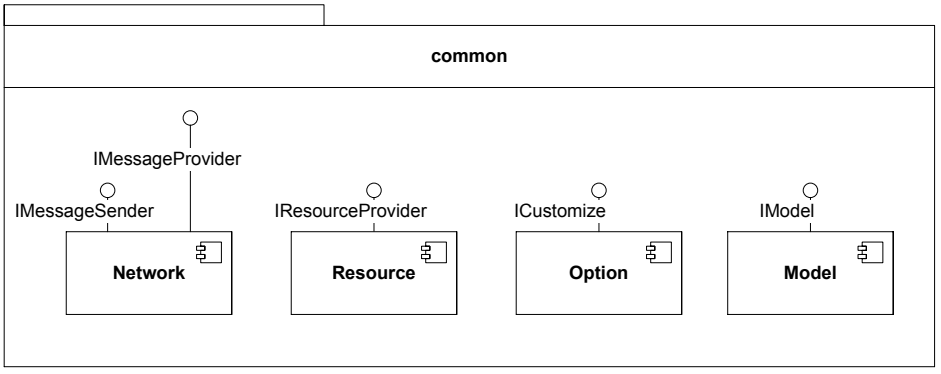

FIGURE 4.6: Detail view: FreeCol Commons

- Detailed View: FreeCol Commons (Figure 4.6): This view shows a number of components that are used by the FreeCol client as well as by the FreeCol server. The most important of these is the common model that is used by the FreeCol client and is also extended by the FreeCol Server.

The component diagrams have been created in an architectural reconstruction of FreeCol that took place before the experiment and was independent of the experiment.

In the experiment we have tested 7 different questions about the design and architecture of Freecol. The questions have been confirmed by the independent analysts as being relevant questions for understanding Freecol's design and architecture. The questions and their classifications are shown in Table 4.3. We have classified the questions into the question types from Section 4.2.1 as follows:

- $QT1$: For 3 questions ($Q1$, $Q5$, $Q7$) the component diagrams provide direct guidance or help to better understand Freecol's design and architecture, but the information in the component diagrams alone is not enough to answer the question fully.
- $QT2$: For 1 question ($Q2$) the answer can be deduced both from the component diagrams and the source code organization (in packages) alike.
- $QT3$: For 3 questions ($Q3$, $Q4$, $Q6$) the component diagrams provide no obvious information, only vague orientation, and digging in the source code is required to answer the question.

We asked 3 questions of type $QT1$ and 3 questions of type $QT3$, so clearly those two question types are our main focus. We also checked $QT2$, but only once, as it is a rather seldom occurring option somewhat in between $QT1$ and $QT3$ that some design aspects are directly visible from the source code organization. To illustrate the difference between the two extremes in our experiment, $QT1$ and $QT3$, and let us briefly explain the difference in the level of detail modelled:

- *Example for Type $QT3$:* For question $Q3$ of type $QT3$ there is only a component *AI* visible in the FreeCol Server Architecture model. It has a single connector with the interface *AIPlayer* to the *Model* component. This provides only *vague orientation* for answering question $Q3$, as it enables the participants to know that AI concerns are implemented in the server packages and that there is a link to the model classes, but it does not provide details for answering the question.
- *Example for Type $QT1$:* For question $Q1$ of type $QT1$ there are significantly more details and links to all important aspects of the question in the component diagrams. First of all in the FreeCol Client Architecture model, we can see a component *Controller* that is linked through a connector with an interface *GameControl* to the *Actions* component and through another connector with an interface *UpdateHandler* to the *GUI* component. This enables participants to understand how GUI Actions use *InGameController* (and another class *ConnectController*) to perform model, game, etc. updates using basic controls. It also makes it easy to find various basic control tasks in the game's client, which can then easily be found in the source code. The *Controller* also has a connector to a port with the *Model* interface, which links to details in the Commons and Server Architecture component views. In the server, the *Model* component is linked to another *Control* component which is modelled in a detailed view, the FreeCol Server: Control component diagram. This enables participants to understand (1) the client-server relationship for control and the synchronization through models and (2) the event handling for changes through model messages.

Clearly, the level of detail for question type $QT1$ is much higher. We hope this example helps to illustrate what we mean by *"providing direct guidance or help"* for $QT1$ in contrast to *"vague orientation"* for $QT3$.

ID	Type	Question	Classification Details
Q1	QT1	Explain the role of the class "InGameController" in the Package "net.sf.freecol.client.control". What is its purpose?	Component diagrams provide detailed orientation through related components and connectors, and also hint at interesting architectural concerns not easily seen from the source code. Connection to the source code must be made for providing the full answer.
Q2	QT2	How many and which independent executable programs belong to this game?	Component diagrams provide detailed orientation. The answer can be deduced from the component view, but also from the package structure in the source code.
Q3	QT3	Explain how the computer players (AI) are integrated into the game. Which classes are responsible for the integration and implementation of the AI players? In which of the executable program(s) (see Q2) do they run?	Component diagrams provide vague orientation through components. The source code is the main source of information.
Q4	QT3	What is the role of the class "DOMMessage" in the "net.sf.freecol.common.networking" Package? How is it used in the game?	Component diagrams provide no details for answering the question, only vague orientation. The source code is the main source for getting the required information.
Q5	QT1	What is the role of classes in the package "net.sf.freecol.metaserver"?	Component diagrams provide detailed orientation through related components and connectors. Connection to the source code must be made for providing the full answer.
Q6	QT3	What are the roles of the classes in the packages "net.sf.freecol.server.model", "net.sf.freecol.client.control", "net.sf.freecol.common.model"? How are they related to each other?	Component diagrams are useful for basic orientation, but not for answering the question. The source code is the main source of information.
Q7	QT1	In order to show a consistent game state to all players, the programs of the different players must be updated regularly. How and by which classes is this mechanism realized? Sketch the control flow from one class (or object) to the next one.	Component diagrams show related components and connectors, and a few details helpful for answering the question. Component diagrams also hint at the architectural big picture not easily seen only from the source code.

TABLE 4.3: Questions and Classification of Questions

4.2.4 Execution

The experiment was executed in the context of the Software Architecture course at the Faculty of Computer Science, University of Vienna in the summer semester 2012/2013 where it was executed as one of the tasks of the practical course. Before the experiment took place, the participants were randomly assigned to experiment group and control group. Each of the two groups consisted of 30 participants (total participant number: 60).

Figure 4.7 shows the previous experience of the participants for the control group and the experiment group. In particular, the figure shows the programming experience of the participants, which is quite comparable in the two groups, with slightly more participants with longer experience in the experiment group. The industry programming experience is low in both groups, with a few participants with 1, 1-3, or even more than 3 years of industry experience in both groups. Finally,

most participants have no game programming experience; the very few participants with game programming experience are equally present in both groups.

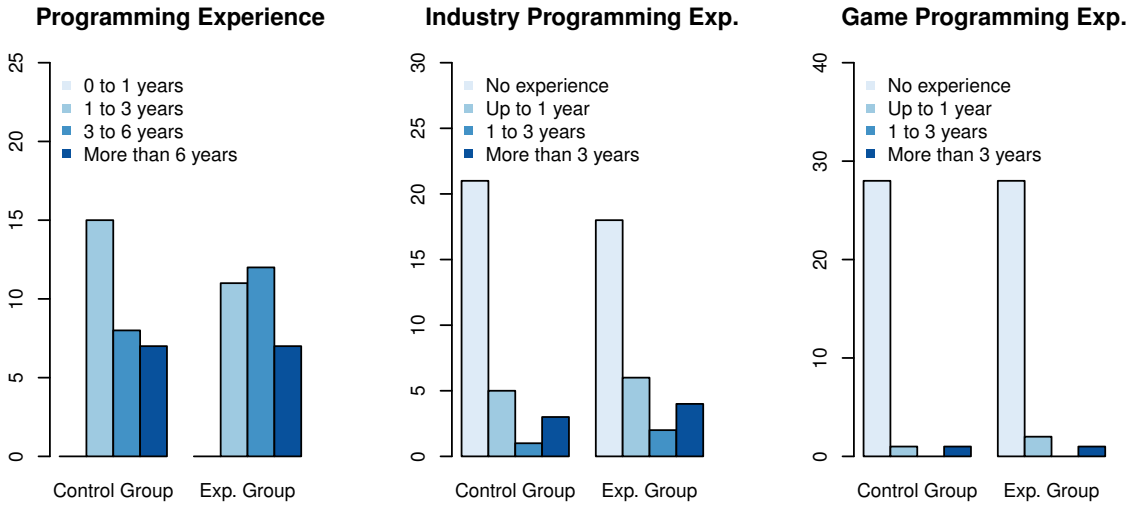

FIGURE 4.7: Experience of the Participants

Before the experiment started, the materials explained in Section 4.2.3 were handed out to the participants and the tasks were briefly explained to both groups. After 15 minutes of introduction, the participants were given time to fill out the questionnaire about their experiences. After all participants were ready, the answering of the questions started. The answers were provided by the participants on paper. This main phase of the experiment lasted for two hours.

The data collection was performed as planned in the design. No participants dropped out and no deviations from the study design occurred.

The experiment took place in a controlled environment. The experiment was conducted for both groups in different rooms, equipped with computers to which the participants had logins. At least one experimenter was present in each room during the whole experiment time to assure that participants behaved as expected. After the experiment, all materials were collected by the experimenters before any of the participants left the room. There were no situations in which participants behaved unexpectedly.

4.3 Analysis

4.3.1 Descriptive Statistics

Figure 4.8 shows the means for the quality of the answers given to the seven questions $Q1$–$Q7$ for both control group and experiment group (the values can also be seen below in Table 4.5). As can be seen, the means for questions of type $QT1$ ($Q1$, $Q5$, $Q7$) are always higher in the experiment group than those in the control group. For the question of type $QT2$ ($Q2$) the control group yields

a slightly better result. The means for questions of type $QT3$ ($Q3$, $Q4$, $Q6$) of the experiment group show the same or slightly better results than those of the control group.

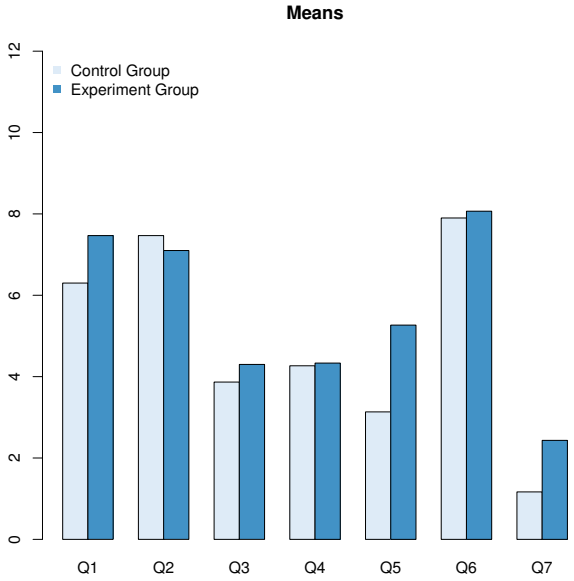

FIGURE 4.8: Means of control and experiment group for the seven questions

4.3.2 Data Set Reduction

Outliers are potential candidates for dataset reduction, i.e. data points that are either much higher, or much lower than other data points. Our analysts did not report any participants that might have caused outliers by intentional or motivated misreporting. The deviations from the means for the ratings of all questions are in a corridor that roughly corresponds to our previous experiences from other exercises with participants in our courses. Hence, we did not want to exclude individual participants from the data set, as excluding data points would have introduced a potential vulnerability for the study results.

An interesting outlier in the means for the seven questions is Question $Q7$, where both groups performed rather poorly. Hence, we studied the answers for this question in depth to understand whether it is necessary to exclude Question $Q7$ from the further analysis, for instance because it was too hard to answer or simply because the participants did not have enough time for answering the question (which was the last question). First we checked the protocols of the experiment and most participants have finished before the end of the 2 hours slot, so the limited time frame does not seem to be the cause of the poor results. To study whether the question was too difficult, we did an in depth study of answers without knowledge whether the individual answers were from the control group or the experiment group. The results are: Indeed, Question $Q7$ seems to be a difficult question that requires complex design and architecture understanding and making connections across multiple parts of the FreeCol system's design and architecture. Most participants failed and reached 0-3 points. Very few participants are in the middle ranks of 4-6 points. Only 6 out of the

60 participants managed to provide a sufficient answer to the question (with a score > 6 points). We checked all sufficient answers, and all of them have understood the problem addressed by the question. As this means that 10% of the participants were able to answer the question sufficiently, it does not seem impossible for novice architects to answer Question $Q7$, just difficult. Hence, we decided to not exclude the question from further analysis but rather view it as a case to study a difficult question of type $QT1$.

4.3.3 Hypotheses Testing

Testing the Normality of the Data

As a first step in analysing the data, we tested the normality of the data by applying the Shapiro-Wilk test [SW65], in order to see whether we can apply parametric tests like the t-test that assume the normal distribution of the analysed data. The null hypothesis H_0 for the Shapiro-Wilk test states that the input data is normally distributed. H_0 is tested at the significant level of $\alpha = 0.05$ (i.e., the level of confidence is 95%). That is, if the calculated p-value is lower than 0.05 the null hypothesis is rejected and the input data is not normally distributed.

Group	**N**	**Shapiro-Wilk test p-value**						
		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**
Control Group	30	0.06505	6.528e-05	0.07345	0.0852	0.005865	7.255e-05	1.362e-06
Experiment Group	30	0.01998	9.035e-05	0.04852	0.3658	0.0002576	3.028e-05	0.0007023

TABLE 4.4: Results of the Shapiro-Wilk normality test

In the Table 4.4 the p-values for the Shapiro-Wilk normality test for the seven questions $Q1$–$Q7$ for both control group and experiment group are shown. As can be seen, most questions do not have a normal distribution (i.e., hypothesis H_0 is rejected). Based on these considerations of normality, we decided to pursue non-parametric statistical tests with our data.

Comparison of the Means Between Two Variables

To compare the means of the variable for the control group and experiment group of a question, we applied the Wilcoxon rank-sum test [MR47]. The Wilcoxon rank-sum test is a non-parametric test for assessing whether one of two data samples of independent observations is stochastically greater than the other. The null hypothesis of the one-tailed Wilcoxon test (appropriate for the hypotheses in our experiment) is that the means of the first variable's distribution is less than or equal to the means of the second variable's distribution,so that we can write $H_0 : A \leq B$. The Wilcoxon rank-sum test tries to find a location shift in the distributions, i.e., the difference in means of two distributions. The corresponding alternative hypothesis H_A could be written as

$H_A : A > B$. If a p-value for the test is smaller than 0.05 (level of significance), the null hypothesis is rejected and the distributions are shifted. If a p-value is larger than 0.05, the null hypothesis can not be rejected, and we can not claim that there is a shift between the two distributions.

ID	**Control Group: Mean**	**Experiment Group: Mean**	**p-value**
Q1	6.3	7.466667	0.02021
Q2	7.466667	7.1	0.4818
Q3	3.866667	4.3	0.263
Q4	4.266667	4.333333	0.494
Q5	3.133333	5.266667	0.01512
Q6	7.9	8.066667	0.2212
Q7	1.166667	2.433333	0.006899

TABLE 4.5: Results of the Wilcoxon rank-sum test

In the Table 4.5 the p-values for the Wilcoxon rank-sum test are shown, together with the means for control group and experiment group. The raw material for these results is presented in the Appendix in Table A.1. Based on the obtained p-values, we can assess that the following distributions show a statistically significant shift between each other: $Q1$, $Q5$, and $Q7$. None of the other variables shows a statistically significant shift.

Testing Hypothesis H for QT1

In our experiment, we have introduced 3 questions related to QT1: $Q1$, $Q5$, and $Q7$. Each of the three questions shows a significant location shift in their distributions, and in each of them the experiment group shows better results than the control group in their means. This provides evidence that H_0 *can be rejected for QT1.* That is, indeed there is evidence that augmenting the source code with architectural component diagrams *improves the quality* of answers that novice architects provide to questions about a software system's design and architecture, if the component diagrams provide architectural guidance for answering the question.

It is interesting to note that the difficult Question $Q7$ shows the same result (even with the highest significance level) as Questions $Q1$ and $Q5$ (of medium difficulty). While many of the participants in the experiment group failed as well, all but one of the sufficient answers were in the experiment group. This result seems to indicate that component diagrams can be especially helpful for problems that require making complex design and architecture connections across multiple parts of the system.

Testing Hypothesis H for QT2

In our experiment, we have introduced 1 question related to QT2: $Q2$. For this question we can observe higher means for the control group than for the experiment group, however the location

shift is not statistically significant. This provides evidence that H_0 *can not be rejected for QT2.* As expected, there is *no evidence* that augmenting the source code with architectural component diagrams does improve the quality of answers that novice architects provide to questions about a software system's design and architecture, *if the component diagrams provide architectural guidance for answering the question, but the same information is visible from the source code.*

Testing Hypothesis H for QT3

In our experiment, we have introduced 3 questions related to QT3: $Q3$, $Q4$, and $Q6$. The means of the experiment group show the same or slightly better results than those of the control group. None of the three questions shows a significant location shift in their distributions, so H_0 *can not be rejected for QT3.* As expected, there is *no evidence* that augmenting the source code with architectural component diagrams does improve the quality of answers that novice architects provide to questions about a software system's design and architecture, *if the component diagrams provide no direct guidance or help, only vague orientation in related components and connectors.*

4.4 Discussion of the Post-study Questions

In this section we present the results from one post-study question that all participants were asked to complete, and one post-study question that was posed only to the experiment group. The idea behind the questionnaire was to capture the participants own opinion on the supportive effect of component diagrams as well as their opinion on how well they understood the architecture of the given software system and whether their opinions are reflected in the achieved results. Therefore we asked all participants:

> *Question 1: On a scale of 1 (worst) to 10 (best) – how would you rate your understanding of FreeCol after this session?*

We report the data for this question in Table 4.6. The medians and means show that the experiment group, which achieved slightly better scores then the control group, also give themselves a slightly better score then the control group. However these two samples are not normally distributed and thus we used a Wilcoxon Rank Sum test to determine if the two samples differ significantly. With a resulting p-value of 0.3941 the groups do not differ significantly. Like any result based on the personal opinion of the participants, it is hard to draw any conclusions as a number of different interpretations are possible.

One possibility is that component diagrams have no effect on the self-evaluation of the participants, as the empirical score of the participants shows a similar difference between the experiment and

	Median	**Means**
All participants	6	5.809
Experiment Group	6.5	6
Control Group	5	5.625

TABLE 4.6: Median and means for post-study Question 1 for all participants, the control group, and the experiment group.

the control group, and on average, the participants estimate their own performance as quite good and thus come to a similar result as our empirical study.

The results also seem to underline the results of our empirical study in so far, as we were able to show a supportive effect of the component diagrams only for one type of question. Hence, it cannot be expected that the experiment group has the impression to have gained a significantly better understanding than the control group with regard to the whole experiment. It would be interesting to study if the results would be significantly different if we would only provide questions of type $QT1$.

Another possible conclusion from this result with respect to the rest of the experiment could be, that component diagrams do not provide a false sense of improved understanding. As a significant difference in the self-evaluation could be interpreted as a distortive effect that component diagrams have on how participants rate their own understanding of the system when the experiment only showed significant results for $QT3$ and not $QT1$ or $QT2$.

In addition we asked all participants of the experiment group to answer the question whether they found component diagrams helpful:

> *Question 2: On a scale of* 1 *(worst) to* 10 *(best) – how helpful were the component diagrams?*

The statistical data for this question is shown in Table 4.7. One of the participants gave the lowest possible score and the highest score given was an eight (given by three participants). However, the average score is 5.2407, which is close to, but slightly below the average for the defined scale of 1 to 10 which is 5.5.

It seems that although the empirical data suggests a beneficial effect of component diagrams under specific circumstances ($QT1$), this effect is not reflected in the participants opinions. Again, as we asked the question for component diagrams in general and not for specific questions, this result can be expected after the results of the empirical study, as it is unlikely that the questions of type $QT2$ and $QT3$ made no impression on the participants or that the questions of type $QT1$ had such an impact on the participants' opinions that they would effect the overall rating of the component diagrams.

Means	Median	Minimum	Maximum
5.2407	5	1	8

TABLE 4.7: Statistical data for the question whether the participants deemed component diagrams helpful

With respect to the two post-study questions, we can conclude that the answers are consistent with the results from the study, as the study only showed a supportive effect for one of the three question types and thus statistically significant results for the post-study questions were not be expected.

4.5 Validity Evaluation

Several levels of validity have to be considered in this experiment. We consider the classification scheme for validity in experiments by Cook and Campbell [CC79].

Internal Validity

The internal validity is the degree to which conclusions can be drawn about cause-effect of independent variables on the dependent variables.

- The subjects' experiences in the two groups have approximately the same degree with regard to programming, industrial, and game programming experience. Of course, a certain difference in experience between the two groups can not be entirely excluded.
- All subjects have at least medium programming experiences and have passed several courses on programming and design at our university. Hence, we consider their responses as valid, keeping in mind that our goal was to analyze the supportive effects component diagrams have on novice architects.
- As the experiment lasted less than 2 hours fatigue effects were not considered relevant.
- No randomization of questions has been used and thus the participants might have gained insight while carrying out the first tasks that might have influenced their performance in the later questions. No particular order of the questions was enforced by the experimenters and the participants received all questions at the same time and could choose the order in which they answered the questions freely. Furthermore participants could revise answers to previous questions at any time. Thus it is not likely that any insight gained would affect the later tasks more than the tasks performed in the beginning.

- The experiment happened in a controlled environment in separate rooms under supervision of at least one experimenter. While it is not possible to completely exclude misbehavior or interaction among participants, it is not very likely that misbehavior or interactions have had a big influence on the outcomes of the experiment.
- Possibly the analysts could have been biased towards the experiment group in some way. We tried to exclude this threat to validity by not revealing to the analysts the identity of the participants or in which of the two groups they have participated. Hence, it is rather unlikely that this threat occurred.
- Regarding the post-study questions: The participants were not told that the tasks they were given, were part of an experiment. In addition, they were explicitly told that the questionnaire at the end of the experiment was not part of their grade but that we asked for their personal opinions. This way we tried to reduce risk that the participants' opinions were influenced by the experimenters. However this thread cannot be totally ignored.

External Validity

The external validity is the degree to which the results of the study can be generalized to the broader population under study. The greater the external validity, the more the results of an empirical study can be generalized to actual software engineering practice.

- We used students of our software architecture lecture as subjects. As discussed above, they have medium programming and design experience, but limited professional experience. Hence, we believe them to be well representative for the target group of novice architects, but if and how the results can be translated to more experienced architects is open to future study. We plan to replicate the experiment with other target groups to find out more.
- The instrumentation and object in the experiment might have been unrealistic, not representative, or too simple to allow generalization. For instance, FreeCol as an open source game might be too simple or no representative software system for typical architectural studies. The component diagrams used might not be representative or unrealistic. Or the questions asked might not be typical design or architecture questions. All these considerations might impede the generalizability of our results. We do not think that this is the case as FreeCol is a widely used, non-trivial software system implemented in Java. The component diagrams were created in an architectural reconstruction effort that took place before the experiment and was independent of the experiment. The questions have been confirmed by the independent analysts as being relevant questions for understanding the design and architecture of FreeCol.

- The experimenters could have biased the measurements of the independent variables. We mitigated this risk by assigning the quality ratings to independent analysts that had no knowledge about the goals of the experiment. Furthermore the analysts did not know the identities nor the groups of the participants.
- While all of the participants had seen component diagrams before, their prior experience with component diagrams was limited and thus their personal opinions on the helpfulness of those diagrams primarily stemmed from this experiment. Participants with other backgrounds, e.g. seasoned architects who have prior experiences with using component diagrams, might give answers that are not or less influenced by the tasks in the study.

Conclusion Validity

The conclusion validity defines the extent to which the conclusion is statistically valid. The statistical validity might be affected by the size of the sample (60 participants, 30 in each group). The size can be increased in replications of the study in order to reach normality of the obtained data. We plan to replicate the study with different systems and by engaging subjects who work in industry in our future work.

Construct Validity

The construct validity is the degree to which the independent and the dependent variables are accurately measured by their appropriated instruments. As only one object, the FreeCol implementation and associated component diagrams, was used in the experiment, there is the risk that the cause construct is under-represented. Possibly, the results could look different if multiple systems and multiple sets of diagrams would be used for the recovery. In case of bigger and more complex systems, diagrams might be easier to understand than source code. We assume that the used system is representative for medium-size object-oriented systems. This threat, however, can not totally be ignored. Another potential threat to validity is that we only used one variable to measure the quality of answers. This does not allow cross-checking the results with different measures.

4.6 Conclusions

Our study provides initial evidence on how architectural component diagrams help in understanding the design and architecture of software systems. The results indicate that architectural component diagrams are especially useful if a direct link from the component diagram's elements to the problem that requires understanding can be made. Component diagrams seem to help in such cases to

understand the bigger architectural connections that are hard to see from studying the source code alone and/or they provide orientation in the source code to understand such problems. However, there is a different situation for problems that are readily solvable by looking at the source code (like the question of type $QT2$ in our experiment) or for problems that are only vaguely linked to what is depicted in the component diagrams (like the question of type $QT3$ in our experiment). As expected, we found no evidence that architectural component diagrams help with respect to $QT2$ and $QT3$, for instance, just by providing a big picture view or providing some general kind of orientation. This data is in line with the observation from the post-study questions that the participants' self-assessment of their architecture understanding is slightly but not significantly higher for the experiment group and that the participants gave an average score of 5.2 out of 10 on the question how helpful the component diagrams were.

We can conclude for the design of architectural component diagrams that they should be designed with specific important architectural problems in mind and that elements of the component diagrams should explicitly represent links to those problems. That is, components, connectors, and other model elements for providing an abstract understanding of the design that resolves the problem should be shown in the diagrams. The component views related to questions of type $QT1$ in our experiment achieve this by leaving out irrelevant details and showing high-level connections of system parts that are hard to reconstruct by just studying the low-level source code classes. It also seems to be important that these links from component views to the problem in focus are modeled in enough detail. Only vaguely showing a problem-related component in its context of other components and connectors that are not related to the problem is not enough.

It seems plausible, based on our results, that such details could also be provided through other architectural views or through architectural knowledge models. Further, it seems that making links to the source code is important for the supportive effect revealed in our study. Such links can be made explicit through traceability links. Hence, it also seems plausible that establishing traceability links between architectural component views and code might have a further supportive effect. We plan to study these aspects in further studies in our future work. From the combined results of this and future studies we plan to develop design guidelines for architectural component diagrams.

Regarding generalizability, our results are strictly limited to the target group of novice architects with medium programming experience. We expect that similar results will also show for seasoned architects, but potentially they can make more use of vague information in architectural component diagrams. Again, we plan to investigate this in our future research.

In the following chapter (Chapter 5), we use the findings from this Chapter as a basis to propose our own approach for documenting architectural component views, that links architectural components to the source through architecture abstraction specifications and takes the findings from this controlled experiment into account, as it provides navigable traceability links directly in the

prototype and, if necessary, as HTML tables that link architectural components to source code and vice versa.

5 Semi-automated Architectural Abstraction Specifications for Supporting Software Evolution

5.1 Introduction

The approach presented in this chapter focuses on architectural abstractions from the source code in a changing environment while still supporting traceability. It is motivated and based on the findings of the controlled experiment we presented in Chapter 4 on the one hand, and on the other hand by the problem that in many software projects the design and the implementation drift apart during development and system evolution [Jan+07]. In some small projects this problem can be avoided, as it might be possible to understand and maintain a well written source code without additional architectural documentation. For many larger systems, this is not an option, and additional architectural documentation is required to aid the understanding of the system and especially to comprehend the "big picture" by providing architectural knowledge about a system's design [Bro13]. One way to provide this information are automatically generated diagrams of the systems (e.g. in form of class diagrams) [Kos03]. However these diagrams usually do not represent higher-level abstractions, and hence they hardly support the understanding of the big picture. First of all, the sheer size of the automatically generated diagrams is often a problem. In addition, creating an automatic layout or partitioning that is understandable is still an open research topic [SW05; Eig+03]. Clustering approaches from the reengineering research literature (e.g. [Abr+00; Die+08; DB11]) can help to obtain an initial understanding and make sense of such diagrams. However the case study by Corazza et al. [Cor+10] shows that in five out of seven cases it is necessary to make manual corrections for about half of the entities of the analyzed source code.

As a consequence, today the documentation of the system's architecture is usually maintained manually. To model architectural knowledge, often models using box-and-line-diagrams [RW05], UML [Obj10], architecture description languages (ADLs) [MT00], or similar modeling approaches are used. In many cases, such models are created before the actual implementation begins. Later, during implementation and system evolution, they loose touch with reality because changes to the software design are only made in the source code while the architectural models are not updated [Zim+08]. This problem is known as *architectural knowledge evaporation* [Jan+07].

A considerable number of works exist that focus on abstractions from source code [Men+02; Mur+95a; DB11; Egy04]. However, to the best of our knowledge, so far none of these approaches targets architectural abstractions at different levels of granularity, traceability between architectural models and the code, and the ability to cope with the constant evolution of software systems. Our approach introduces the semi-automatic abstraction of architectural component and connector view models from the source code based on an architectural abstraction specified in a domain specific language (DSL) [Fow10; Gro09]. In contrast to the related works, this approach specifically targets architectural abstractions and requires changes to the architectural abstraction specifications only in the rare case that the architecture of the system changes, but not for the vast majority of non-architectural changes we see during a software system's evolution (see Section 5.2). Please note that in the literature the term "component model" is often used to describe metamodels for component-based development [LW07]. In this chapter, we (only) use the term *architectural* component and connector view (or component view for short) to describe a model that contains architectural components (as in [Ive+04]).

We chose a semi-automatic approach to enable the software architect to provide information which system details are relevant for getting the right level of abstraction – as software architecture is usually described in different views at different levels of abstraction. Our goal is to let the software architect specify this information with minimal effort in an easy-to-comprehend DSL that provides good tool support. Our approach allows architects to create different architectural abstraction specifications that represent different levels of abstraction and thus supports views ranging from high-level software architectural views to more low-level software design views. Once the software architect has defined an architectural abstraction in the DSL, we can automatically generate architectural component views from the source code using model-driven development (MDD) techniques and check whether architectural design constraints are fulfilled by these models.

As our approach focuses on defining stable abstractions in the architectural abstraction specification, it can cope with many changes to the underlying source code without changing the architecture description (i.e., an instance of the DSL). Only changes to the architecture itself, which usually require a substantial modification of the source code, require the architectural abstraction specification to be updated.

By creating different versions of the architectural component view over time, we are able to use a delta comparison to check and reason about the changes of the architectural component view. The generated models can be compared to a design model, to check the consistency of an implementation and its design, and to analyze the differences. To support the iterative nature of our approach, it also supports automatically checking the consistency between the source code model and the architecture abstraction specification on the fly.

Once the architectural component views have been abstracted, another problem is to identify which parts of the source code contribute to a specific component, i.e., to support traceability between

architectural models and code. Today, this usually requires substantial and non-trivial manual effort to identify which code elements are related to which model elements. In contrast, in our approach, traceability can be automatically ensured, as model-driven development (MDD) [BG05] is used to generate the required traceability links between the model elements and the source code directly from the architectural abstraction specification.

The remainder of this chapter is organized as follows: Section 5.2 explains the research problem addressed by this chapter in more detail, as well as the research method that was applied to design and evaluate the DSL. Section 5.3 gives an overview of our approach. Section 5.4 provides details about our architectural abstraction DSL and its implementation. In Section 5.5 we present the evaluation of our approach based on five cases and a performance evaluation.

In Section 5.6 we discuss open issues and lessons learned. We conclude this chapter in Section 5.7.

5.2 Research Problem

During the software development life-cycle, the source code and the architecture of a system evolve and change. This often results in architectural knowledge evaporation [Jan+07]. One of the reasons for this is that in today's software development processes the software architects often have to manually capture and maintain the architectural knowledge, which is a tedious task that is often forgotten in the daily business [Zim+08]. Additionally, when using conventional means for architecture documentation like abstracted UML models or box-and-line diagrams, the traceability between the architecture and the source code is lost. This can also lead to architectural knowledge evaporation, when architects and developers lose track of the correspondences between code and architecture.

A number of approaches have been proposed to address this research problem by providing automatically or semi-automatically produced abstractions from the source code [Men+02; Mur+95b; DB11; Egy04]. In contrast to these related works, our approach *specifically targets architectural abstractions*. That is, we have designed our DSL to only require changes of the architecture abstraction specification once the architecture of the underlying software system has changed, but not for other kinds of changes. In case of non-architectural changes, an updated architectural documentation can automatically be re-generated from the altered source code without manual changes in the architectural abstraction specification.

To reach this goal we have designed and implemented the DSL using an incremental refinement process, following the design science research method [Hev+04].

Based on our original Research Question 2 (see Chapter 2) for this approach, we formulated the following research questions:

RQ5.1 Is it possible to create architectural abstractions based on generic filters that are stable and that only have to be changed when architectural changes occur, but do not have to be changed when non-architectural changes occur in order to prevent knowledge evaporation and provide up-to-date architecture documentation throughout the evolution of a software system?

RQ5.2 Is it possible to automatically generate traceability links that can be used to support the architects in the definition of the architectural abstractions?

RQ5.3 Is it possible to automatically check the consistency between design and code for different versions to support the architect during the evolution of a system?

RQ5.4 What is the effort to create and maintain architectural abstractions under the assumption that knowledge about a system's architecture already exists?

RQ5.5 Is it possible to do consistency checking and generate an architectural component view with traceability links in an acceptable amount of time on a development machine?

In our design science research, these research questions emerged incrementally. We started with RQ5.1 and incrementally refined it through the other research questions. In particular, we learned during our research that traceability links (RQ5.2) are important for creating architectural abstractions. Our focus on evolution in RQ5.1 later led us to also study consistency checking problems during system evolution (RQ5.3). Finally, our early prototype and usage experiences showed that development effort (RQ5.4) and execution time (RQ5.5) are important for acceptance and usability of our approach.

To address the research questions, while developing our DSL, we have studied the evolution of various software systems and their architecture documentations. We have classified changes in these systems into architectural changes and non-architectural changes. In an incremental refinement process, we have improved the DSL and its DSL tools to only require changes to the architectural abstraction code for changes classified as architectural abstractions in the studied samples of architectural evolution. In each incremental design cycle we have added more samples of architectural evolution and continued the iterations until only architectural changes required changes in the architecture abstraction specification.

Finally, we have evaluated the resulting DSL for all changes that can be observed in a number of consecutive versions of five open source projects. As can be seen in this study, reported in Section 5.5, the vast majority of changes are non-architectural changes, and they can be tolerated by the architectural abstractions defined in our DSL without changes to the architectural abstractions. Only when changes that are classified as architectural changes are introduced in the open source systems, updates to the architectural abstraction code in the DSL are necessary.

5.3 Approach Overview

In this section we present an overview of our approach for supporting the semi-automated architectural abstraction of architectural component views throughout the software life-cycle. Our approach allows software architects to compare the abstract model with a previously defined architectural model and to maintain that model in correspondence with the source code over time. For this purpose we have introduced a DSL that defines architectural abstractions from class models, which can be automatically extracted from the source code, into architectural component views. We believe that in a lot of cases it should be possible to create architectural abstractions that are stable during the implementation process and only need to be changed when architectural changes occur (e.g., leading to significant restructuring of the architectural design).

Once an architectural abstraction specification is defined, we can automatically generate the architectural component view. The workflow for the generation process is depicted in Figure 5.1. First, a class model is extracted from the source code. The extraction of a class model from source code decouples our approach from a specific source language since the approach works on language independent UML class models. For instance, for Java different tools exist that can perform this extraction [Spi11; Soy11]. Then, a model transformation is used to generate a UML component view. This model transformation uses the architectural abstraction specification defined in the DSL code and the class model as inputs, and it generates UML component views from this input. The architectural abstraction specification is needed here as it describes the relation between the abstract model and the source code. The model transformation also generates bi-directional traceability information that links the DSL, the class models, and the architectural component views. During the model transformation, consistency checks are applied to identify potential discrepancies between design and code.

As the software system evolves over time, we use our architectural abstraction in the DSL to create multiple architectural component views, e.g. one for each version of the software systems. Our approach can also be used to compare the created models to each other and to architectural component views created manually during early software architecture design by the software architects. This way, software architects can identify where the implementation differs from the original design or from previous versions. They can then argue whether these changes are intended (e.g. flaws in the design) or not (e.g. the implementation is not in accordance with the design). The comparison of these very similar models with only minor differences is a straightforward task. Approaches for advanced model comparison and a variety of frameworks that implement this functionality already exist (see e.g. [Ecl11a; KP]). Based on this comparison, model consistency between a model and consecutive versions can be checked. For example, if the original design model is compared to models that were generated based on the existing implementation, the comparison can indicate

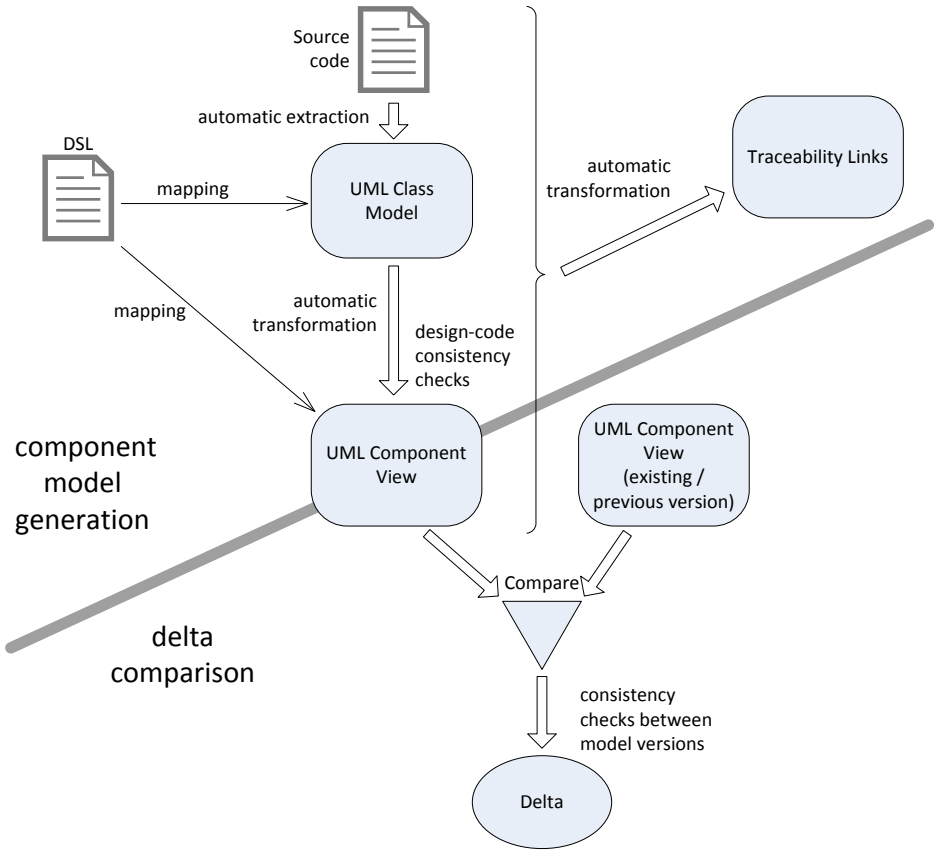

FIGURE 5.1: Generating architectural component views from source code and comparing different model versions

which components are not yet implemented or how communication between components works in the current implementation with respect to the intended design.

This approach enables developers to maintain an architecture documentation by providing an "up-to-date" architectural component view that reflects the source code. In order to support the developers in the definition of architectural abstractions and throughout the development our approach generates traceability links between the source model and all generated artifacts. These links provide direct navigation between the different artifacts in our tool.

In addition to the already mentioned consistency checking between different versions of the software, our approach provides automatic consistency checking between the different artifacts of the same version. These checks are based on the automatically generated traceability links and are automatically triggered whenever one of the artifacts is changed. The implemented consistency checks cover the following artifacts: source model, the architectural abstraction specification, and the architectural component view.

If this approach is used in a software development project from the beginning, the software architect drafts an architecture of the system as an abstraction specification that describes components that

do not yet exist and that will be implemented over time. This way the architect can keep track of the already implemented components using consistency checks as well as generate an abstraction model whenever she desires to do so. As the architecture of the system evolves, adjustments to the architecture abstraction specification are likely, while non-architectural changes should not have an effect on our architecture abstraction specification.

Our approach also eases another frequently discussed problem in software projects: Often, the connection between design and source code is lost during development. Using our architectural abstraction approach, developers can keep track of which parts of the source code correspond to which architectural components, introducing traceability links from the architectural model to the source code and vice versa.

Multiple architectural abstractions can be defined for the same source code to create different views at different levels of abstraction, where one architectural abstraction provides an overview of a system and other architectural abstractions provide detailed views of different parts of the system on varying levels of abstraction.

Our proof-of-concept implementation[1] uses the EMF implementation of UML [Obj10] for its class and architectural component view. This way it is possible to leverage architectural component views created during design time and repeatedly compare them to the current status of the implementation.

5.4 Domain Specific Language for Specifying Architectural Abstractions

To support the architectural abstraction from the automatically created class models to the architectural component views, we define a DSL based on Xtext 2.3 [Ecl11b]. This DSL provides rules for abstracting the detailed UML classes into architectural components. The rules for defining the abstractions can be grouped into three categories:

- *Name- or ID-based filters*: This category of filters selects classes based on the name or ID of an object; for example all classes that reside in a specific package or all classes that contain the string “message” in their name.
- *Relation-based filters*: These filters select classes based on their relationships to a selected class or interface; for example all classes implementing a specific interface.
- *Compositions*: This category contains set operations instead of actual filters. Using set operations one can manipulate the result sets from other filters in order to combine a number of resource sets or define exclusions from more general filters.

[1] This prototype is available at https://swa.univie.ac.at/ArchitectureAbstractionDSL.

```
ComponentDef returns ComponentDef:
'Component' name=ID
'consists of' (expr=OrComposition)
connectors+=ConnectorAnnotation*;

ConnectorAnnotation:
{ConnectorAnnotation}
'connector to' targets+=[ComponentDef] (',' targets+=[ComponentDef])*
('implemented by'
(implementingExpression+=OrComposition)?
('relation: ' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME]
(',' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME])*)?)?
;
OrComposition returns Expression:
ExcludeComposition (
{OrComposition.left=current} 'or' right=ExcludeComposition)*;

ExcludeComposition returns Expression:
AndComposition (
{ExcludeComposition.left=current}
'and not'
right=Primary
)*;

AndComposition returns Expression:
Primary ({AndComposition.left=current}
'and'
right=Primary)*;

Primary returns Expression:
NameFilter | RelationFilter |
ExtensionFilter | '{' OrComposition '}';
[...]
```

Listing 1: Excerpt of the Xtext grammar of our architectural abstraction DSL

We provide a number of different clauses that map groups of class model elements to components in the architectural component view and to define exceptions to these rules. For the manipulation of sets we provide three basic operations (`union`, `intersection`, and `complement`). For more flexibility, we also added custom filters implemented in Java or Xtend [Eff+12]. For this purpose we introduce two special clauses. The Java extension is supported using a filter that is implemented as a static Java method. This method has to accept two parameters: the DSL object of type `JavaExtensionFilter` and a List of `Package` objects. The method is expected to return all UML classifiers that passed the filter. A similar clause exists for using custom filters defined in Xtend.

A complete list of all the clauses that we defined for architectural abstractions can be found in Table 5.1.

The required and provided interfaces of a component are automatically deduced from the UML-class model by defining all external interfaces, used by the component's implementing classes, as

required interfaces. All interfaces that are implemented by a component's implementing classes and used by another component are deduced as provided interfaces. The interface details can be hidden by aggregating interfaces in ports as dedicated interaction points and using assembly connectors between the ports.

Additionally our DSL supports the definition of connectors between components. A connector in the architectural abstraction specification supports the definition of realizing objects by using a) the same clauses that are available for components and or b) by specifying a UML dependency relation from the UML class model as the implementing realization. If realizing objects are specified they are stored in the generated UML component view in the form of a UML `Realization` relation that can later be used for consistency checking.

In our examples and studies that we have used to incrementally refine and evaluate the DSL, this set of language elements has been proven to be sufficient to express architectural abstractions in a way that tolerates all kinds of non-architectural changes (see Sections 5.5.1 and 5.5.2 for details on five cases of open source projects).

An excerpt of the DSL specification is depicted in Listing 1 as an Xtext grammar. It shows the definition of the infix operators for union (`and`), intersection (`or`), and complement (`and not`). `{,}` can be used to change the operator precedence. The complete Xtext grammar can be found in Appendix B.

The transformation is implemented in Xtend [Eff+12] which is first defined for the abstract type `Expression` and then refined for each of the DSL's clauses.

5.4.1 Illustrative Example

Let us illustrate the use of our architectural abstraction DSL with a simple example. Figure 5.2 shows a high-level architectural component view for the Frag project [Zdu11]. Frag is a dynamic programming language implemented in Java, specifically designed for supporting building DSLs and supporting MDD. An excerpt of an architectural abstraction specification in the DSL which is used to generate the component view depicted in Figure 5.2 can be found in Listing 2.

5.4.2 Automatic Generation of Traceability Links

The generation of traceability links is integrated in the architectural component view transformation. As the transformation is executed, the architectural abstraction specification (i.e., the DSL code) is evaluated for each component. This evaluation yields a set of realizing classes and interfaces for each component. This set is stored in the generated UML component view using a UML Realization relation which is defined by the UML-Standard [Obj10], as a relation between two sets

Filter	Parameters	Description
class name	String	all classes, who's name matches the regular expression
package name	String	all classes residing in packages with names matching the regular expression
contained in package	ID	all classes residing in the package identified by the ID
uses	ID	all classes using the class identified by the ID
used by	ID	all classes used by the class identified by the ID
child of	ID	all child classes of the class identified by the ID
super type	ID	all super classes of the class identified by the ID
instance of	ID	all instances of the interface identified by the ID
Java extension	String	String that points to a static Java method which takes the filter object and a List of UML packages as parameters and returns a list of matching classifiers
Xtend extension	String	String that identifies an Xtend function which has the same as the aforementioned Java method.
and	two clauses	infix operator for intersecting the results from two clauses
or	two clauses	infix operator for uniting the results from two clauses
and not	two clauses	infix operator for the difference between two results

Table 5.1: Architectural abstraction DSL clauses

```
Component Interpreter
consists of {
Class (".*Interp") or {
{ // all classes the interpreter uses
// that reside within the core package
UsedBy (root.frag.core.Interp)
and Package (root.frag.core)
} and not Class (root.frag.core.Dual)
}
}
connector to Parser
connector to CommandObjects
implemented by Class(root.frag.core.CommandDispatcher)
connector to FileCommandObjects
implemented by Class(root.frag.core.CommandDispatcher)
Component CommandObjects
consists of Package (root.frag.objs)

Component Shell
consists of
Class (".*Shell") or {
{ UsedBy (root.frag.Shell)
and Package (root.frag.core)
}
and not {
Class (root.frag.core.Interp)
or Class(root.frag.core.Dual)
}
}
connector to Interpreter
```

Listing 2: Code samples for architectural abstraction of the three main components of the Frag (v0.91) example

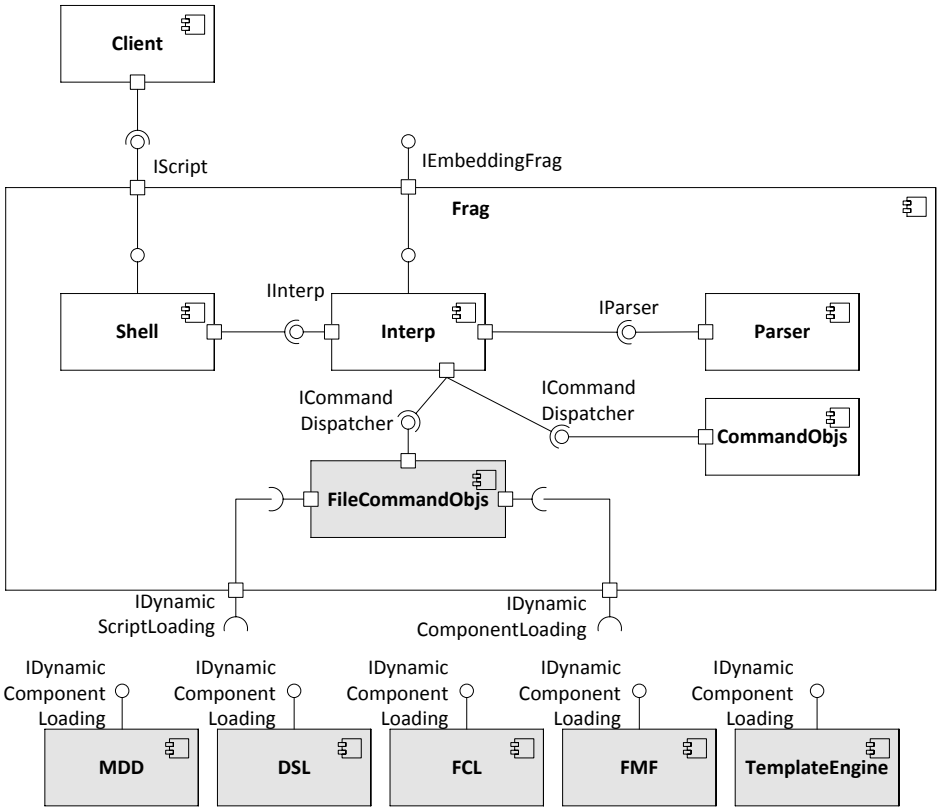

FIGURE 5.2: Visualization of the Frag (v0.91) example for an architectural component view generated from an architectural abstraction specification. Components that were newly introduced between Frag (v0.6) and Frag(v0.9) are colored in grey

of model elements: The supplier (component) specifies the behavior that is realized by the client (a set of classes and interfaces). This relationship is navigable in both directions and thus able to provide the answers to the question "Which classes and interfaces realize component X?" as well as to the question "Which component does this class (partially) realize?". For example, when looking at the "Interpreter" component shown in Figure 5.2 the list of realizing source code elements that are stored in the component realization contains 5 classes (`Interp, FragObject, FragMethod, CodeLine, Callframe`).

Using traceability links, our prototype provides navigable links from the architecture specifications to the source model and vice versa. For example, when the user clicks on `root.frag.objs` in Line 4 of Listing 2, the according element in the source model is automatically opened and the clicked element is highlighted.

The traceability links are used in our prototype to provide the developer with a direct navigation from the source model to a generated architectural component view. For instance, when the developer opens the context menu for the `Interp` class in the Eclipse UML2 editor our prototype provides a context-menu entry that, when clicked, opens the according component in architecture

abstraction specification as well as the generated component in the generated UML2 component view.

While the automatic generation of traceability links from source code is nothing new, our approach uses them specifically to provide interactive tool support to aid the developers and architects in the definition of the architectural abstractions. In addition, the traceability links are the basis for most of our consistency checks, as explained in the next section.

5.4.3 Consistency Checking During Model Transformation

Once an architectural abstraction is defined, it is important to identify discrepancies between design and code. To aid this task, our approach supports design constraint checking. Constraints that have to hold for the class model and the architectural component view can be checked, and then discrepancies can be identified by determining which parts of an implementation are not visible in the design and vice versa. At the moment we have implemented checks for the following constraints and plan on implementing further checks in the future:

- Mapping of a particular class to multiple components
- Components where the DSL clauses do not map to any realizing elements in the source model
- DSL clauses not matching any classes in the class model
- Particular classes that are not mapped to any component
- Components that have a connector in the architecture abstraction specification but do not have a relation in the source model
- Components that have a relation in the source model but do not have a connector in the architecture abstraction specification. E.g. components that make use of an interface that is defined in another component.

Constraint checking is realized in our prototype using Xtext2's [Eff+12] validation framework. This framework distinguishes between *cheap*, *normal*, and *expensive* constraint checks. While *cheap* checks are automatically executed whenever a change in DSL-editor happens, *normal* checks are executed when the model is saved, and *expensive* checks are manually triggered. The majority of our consistency checking constraints are realized as cheap checks, and a few more expensive constraints are realized as normal checks; expensive constraints are not used.

For a number of constraints our prototype automatically provides possible solution options. This ranges from the creation of a new component for classes that are not mapped to any component, over modifying clauses that do not match any source code elements in a way that they possibly

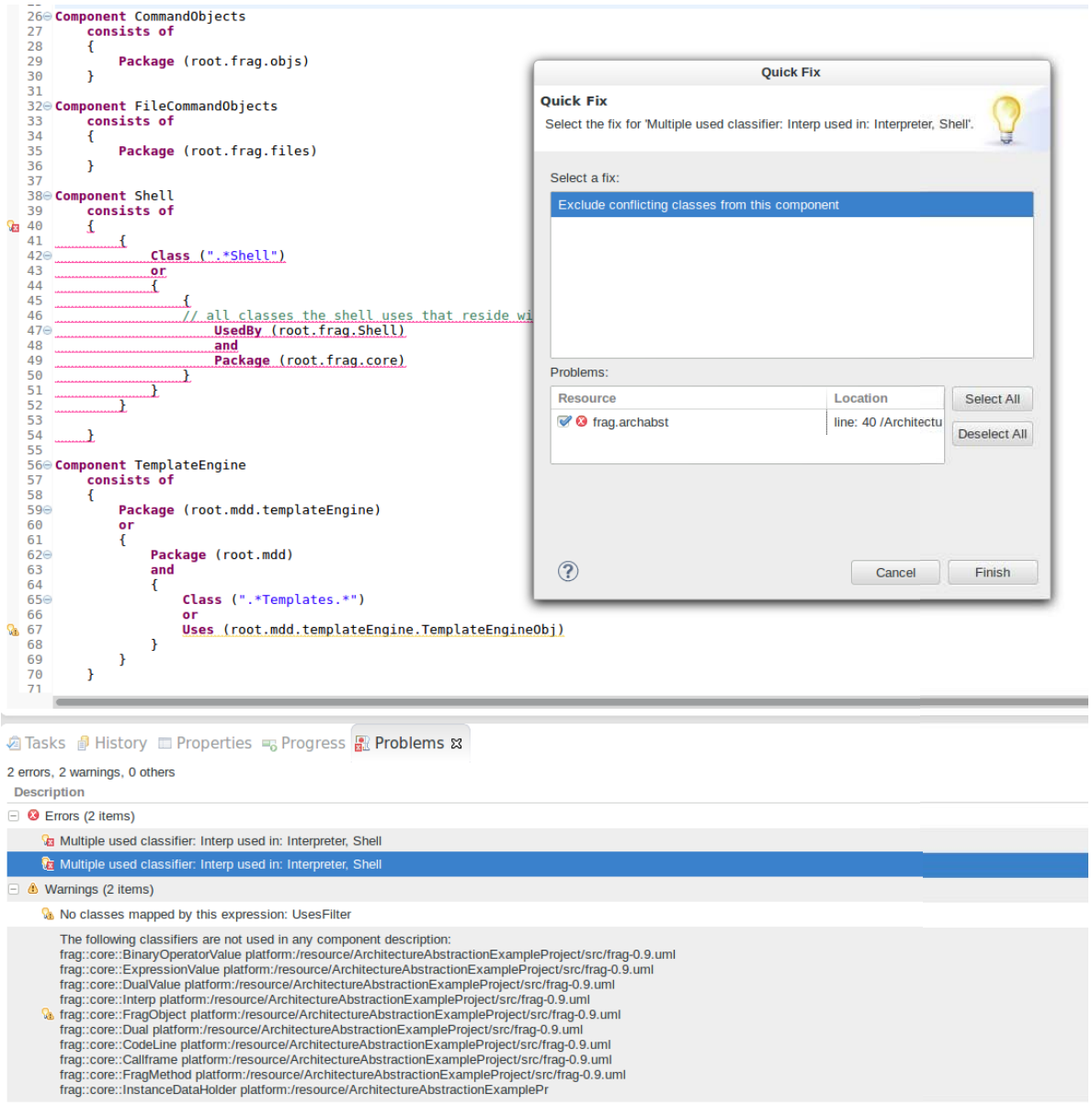

FIGURE 5.3: Exemplary reports of inconsistencies

target a larger number of source code elements, to removing classes that are mapped by multiple components from one of the mapped components.

Figure 5.3 shows two exemplary inconsistencies that were reported for the Frag [Zdu11] example. The first inconsistency is an error reporting that one or more source code elements where added to more than one component in the architectural abstraction specification. Using the already mentioned traceability links, the consistency checks found that the class `Interp` is contained in the architecture abstraction specifications of the components `Shell` and `Interpreter`. This inconsistency is reported with a marker on every abstraction specification the conflicting class is part of. In Figure 5.3 you can also see a proposed solution for this problem: The exclusion of the class `Interp` either from the component `Shell` or the component `Interpreter`.

The other inconsistency reported in Figure 5.3 is a warning that shows the lists of elements from the source code that are not part in any of the defined components. This features provides valuable information to the software architect as it provides information about possible starting points for the next iteration in the architecture recovery process. While source code elements that are used in multiple components are reported as errors, unused source code elements are only implemented as a warning as there might be cases where the software architect does not want to include some source code elements in the architecture abstraction specification.

5.5 Evaluation

To evaluate our approach with regard to the required effort in creating and maintaining architectural abstractions we conducted five case studies. We have performed these case studies as replications of the same research steps (explained below) for five open source projects of different size and in different application areas. The case studies illustrate how the features of the approach like consistency checks and traceability links are used during an architecture recovery.

As approaches like the one introduced in this chapter can potentially require a lot of computational effort, we measured the performance of our tool-suite while creating and maintaining the architectural abstraction specifications for our cases and present the results from this evaluation in Section 5.5.2.

5.5.1 Detailed Cases of Architectural Abstraction Evolution

In this section we discuss the five open source projects Apache CXF [Apa], Frag [Zdu11], Hibernate [LM10], Cobertura [M. 11], and Freecol [The11] that we have used in our evaluation. During the incremental refinement of our DSL design, we started with scenarios from these projects and extended the set of scenarios step-by-step to cover all changes observed in multiple versions of the five cases studied in this section. The lessons learned from this examples are discussed in Section 5.6.

We have performed the five case studies as replications of the same research steps: First, we automatically generated a UML class model from the source code using our parser. Then we tried to gain an initial understanding of the program. In order to ease this task we imported the source code in an Eclipse IDE. After an initial study of the source code, we created a first, incomplete architecture abstraction specification. The time that was required to create this initial specification heavily depended on the size and the previously existing architectural knowledge about the example cases. Then we utilized the consistency checks to further improve the abstraction specification by removing the reported inconsistencies step-by-step. The inconsistencies at this point usually were source code elements that had not been considered in the abstraction specification.

When we were satisfied with the resulting architecture abstraction specification, we updated the source model to a newer version. After that we checked the architecture specification and the new source model for inconsistencies. Any reported inconsistencies were fixed before we continued with the next version of the program.

For all of the examples, we report the following data points:

- The estimated lines of source code give an impression of the projects size[2].
- The number of changes in the source code between different versions together with the number of changes in our architecture abstraction specification necessary to account for the changes in the source code, to indicate how stable our abstractions are[3]. This is directly related to answering RQ5.1 and indirectly to answering RQ5.2 and RQ5.3 as the number of changes between different versions of the examples is related to the number of reported inconsistencies and therefore also to the number of used traceability links.
- The number of components in the architecture abstraction specification as well as the average, standard deviation, and median for the number of classifiers per component provide measures for the abstraction level of our architecture abstraction specification and show that the resulting components are roughly of similar size. This is important, as a single component holding all classes would not have been representative for finding the necessary changes to the architecture abstractions. Therefore this data is related to answering RQ5.1.
- The time to create and to update the architecture abstraction specifications helps to find an answer to RQ5.4 as it indicates the amount of effort that is necessary for creating and maintaining architectural abstraction specifications.
- The estimated time required to gather the architectural knowledge indicates the necessary effort if this approached is used for software architecture reconstruction and not from the beginning of a software project.
- The execution time of our prototype for all example cases, which indicates an answer to RQ5.5, is reported and discussed in Section 5.5.2.

In the following section we report the data for each example case and discuss the lessons learned in Section 5.6.1.

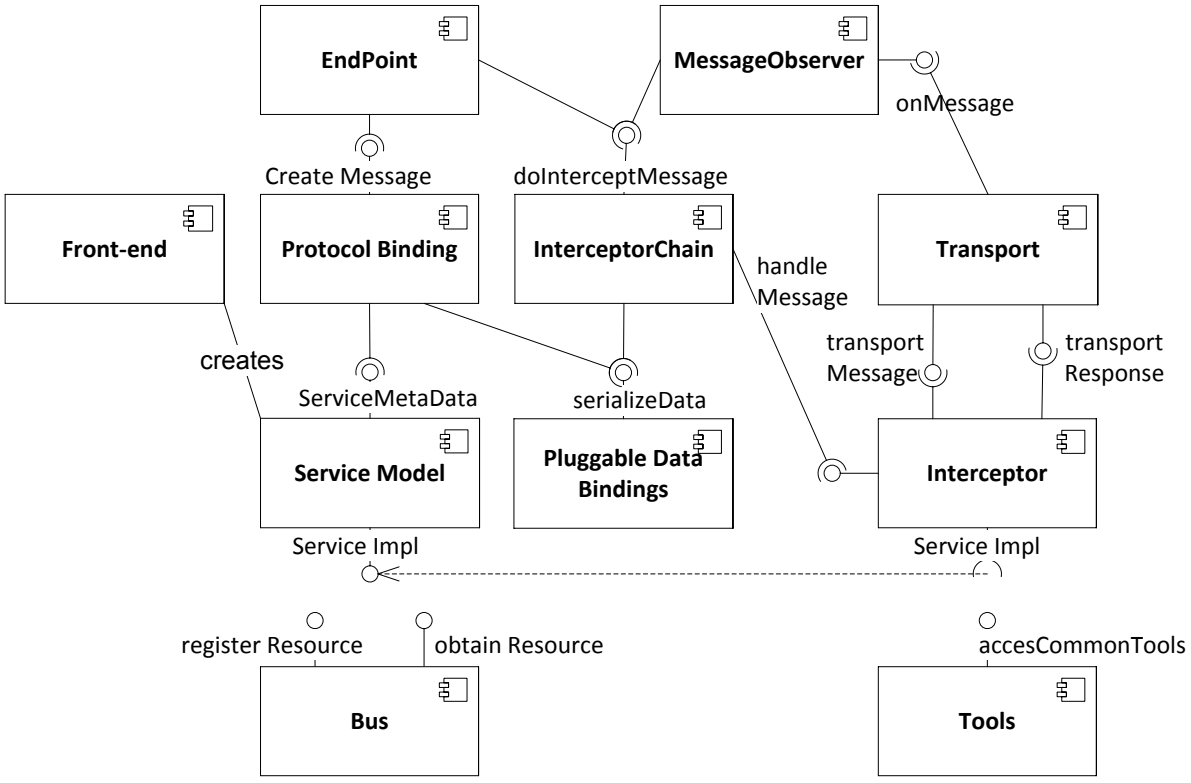

FIGURE 5.4: Apache CXF (v2.4.3) architecture overview [Apa]

5.5.1.1 Case 1: Apache CXF

Apache CXF is an open source services framework that helps developers build and develop various kinds of Web services. In the Apache CXF case, we used the architecture overview that is available from the CXF web-site[4] as a basis for our source code study which required about 2 hours. After this, the first architectural abstraction specification took 15 minutes to create. However it was not complete and the consistency checks reported a relevant number of source code elements that were not considered in the specification. We then spent another two and half hours studying the source code while incrementally improving our architectural abstraction specification and all the time reducing the number of missing source elements that our consistency checks reported. We continued to the point where the consistency checks did not report any inconsistencies.

Next, we updated the source model to a newer version and adapted our architecture abstraction specification to the changed source model. We repeated the last step for all versions of Apache CXF. We estimate the total time required for updating the specification to accommodate the changes in the source models with 45 minutes.

[2]We used the tool `SLOCCount` [Whe09] to estimate the lines of source code.
[3]We used the Linux command line tool `diff` [HM76] to estimate the changes between the different versions of the cases.
[4]`http://cxf.apache.org/docs/cxf-architecture.html`

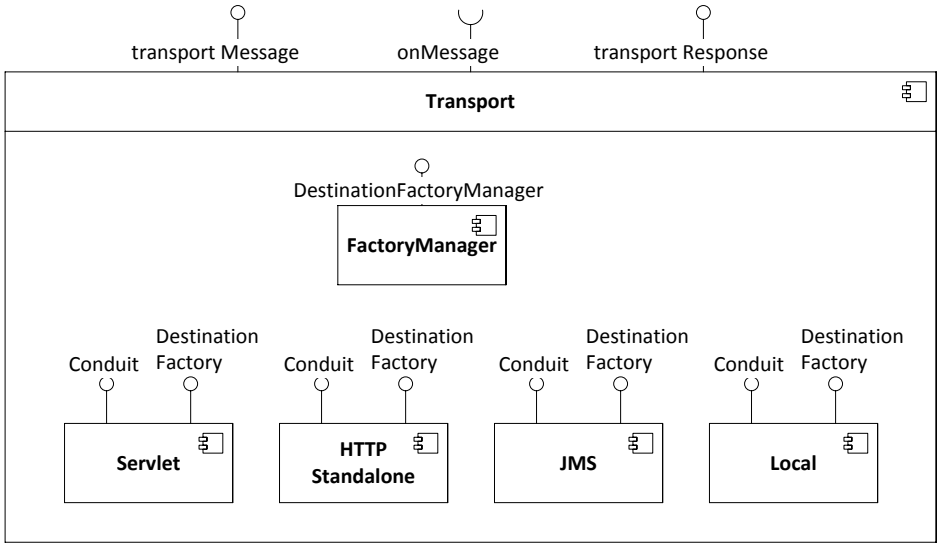

FIGURE 5.5: Detail view for Apache CXF (v2.4.3) transports

Apache CXF version	Files added	Files removed	Files changed	Total changes	DSL changes
Overview 2.0.10 ⇒ 2.2.12	299	83	1040	1422	4 new packages; 2 removed packages
Overview 2.2.12 ⇒ 2.3.7	133	19	923	1075	3 new packages
Overview 2.3.7 ⇒ 2.4.3	115	62	739	916	1 new package
Transport 2.0.10 ⇒ 2.2.12	29	4	90	123	1 new component
Transport 2.2.12 ⇒ 2.3.7	17	3	117	137	1 new component
Transport 2.3.7 ⇒ 2.4.3	20	23	120	163	1 component removed

TABLE 5.2: Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Apache CXF

To show the ability to provide different views for a system we created a detail view for the “Transport" component in the CXF architecture overview (see Figure 5.5.).

The results in Table 5.2 show that in order to keep the Apache CXF abstraction up-to-date hardly any changes were necessary. In the course of the evolution of Apache CXF from version 2.0.10 to version 2.4.3 more than 5000 changes were implemented but only ten changes to the architectural abstraction specification were necessary. These modifications constitute eight new and two removed packages that were introduced between the different versions. This result might be caused by the fact that we only compared minor revisions (no older version than Apache CXF 2.0.10 was available at the time of the case study) during which no major changes to the architecture were made.

When looking at the detail view for the transport component in Table 5.2, three changes were necessary. The package “http_osgi” was added in version 2.2 and removed in version 2.4, and the

CXF Version	sLoC	# of Com-po-nents	Avg. # of classifiers per component	σ for the # of classi-fiers	Median for the # of classi-fiers
Overview 2.0	198k	11	123.64	93.04	98,00
Overview 2.2	342k	11	180.18	150.15	159,00
Overview 2.3	370k	11	194.23	154.11	185.00
Overview 2.4	390k	11	212.73	174.22	193.00
Transport 2.0	1196	5	26.40	14.69	25.00
Transport 2.2	1399	5	31.40	16.89	25.00
Transport 2.3	1715	5	38.20	20.10	30.00
Transport 2.4	1229	5	38.00	22.03	30.00

TABLE 5.3: Apache CXF: Average, median, and standard deviation (σ) for the number of classes per component

package "jaxws_http_spi" that was added in version 2.3.

Table 5.3 contains additional data for this case study. It shows the average number of classifiers per component, the standard deviation for the number of classifiers per component, the median number of classifiers per component, and the lines of source code (in thousands) for the specific version of the program. The standard deviation indicates that the components in the CXF case vary significantly in size (number of classifiers). While in CXF 2.4 the smallest component has only 36 realizing classifiers, the largest component is realized by 534 classifiers. However as the median indicates, the size of the different components is dispersed between these extremes.

The details for the Transport view are also summarized in Table 5.3. As can be seen, 5 components with an average of 26.4 to 38 classifiers have been introduced, again with substantial deviations between different components.

5.5.1.2 Case 2: Frag

Frag is a dynamic programming language implemented in Java, specifically designed for supporting building DSLs and supporting MDD. The high-level architecture of Frag in Version 0.91 was shown already in Figure 5.2.

Frag version	Files added	Files removed	Files changed	Total changes	DSL changes
0.6 ⇒ 0.7	48	44	32	124	2 new components, 2 minor changes
0.7 ⇒ 0.8	9	1	148	158	2 new components, 6 minor changes
0.8 ⇒ 0.91	7	40	36	83	no changes

TABLE 5.4: Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Frag

Frag Version	sLoC	# of Components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
0.6	10k	7	21,57	25,20	6,00
0.7	12k	10	16,00	23,83	6,00
0.8	14k	11	18,73	23,93	10,00
0.9.1	13k	11	14,18	11,29	11,00

TABLE 5.5: Frag: Average, median, and standard deviation (σ) for the number of classes per component

The author of Frag could provide a UML component diagram that he created within half an hour. Using this architectural information as a starting point, the effort required to create an architecture abstraction specification for Frag took about 15 minutes while updating the specification to the newer versions took about 40 minutes.

During this evolution of Frag's architecture, we identified a number of differences when comparing the architecture of Version 0.91 to the architecture of Version 0.6, which is missing the components DSL, FCL, FMF, and TemplateEngine. Figure 5.2 highlights these differences. Components that were part of Frag 0.6 have a white background, while components that where introduced in the newer versions have a grey background.

The architectural abstraction specification for Frag 0.6 has less than fifty lines of DSL code and shows a very straightforward architecture. The changes necessary to conform to Frag 0.7 are shown in Listing 3. They constitute a substantial modification to the architectural abstraction specification. This was expected, since in this revision the architecture of Frag had been reworked to use the Java Reflection API for dynamic dispatching of Frag method calls. Also a number of new features were introduced that led to new components. These components were grouped in a new package called `mdsd`.

Prior to these changes, our consistency checks reported 3 missing packages and 85 classes that were not considered in the architectural abstraction specification for Frag.

```
Component Parser
consists of {
Package(root.frag.parser)
- and not
- Package(root.frag.parser.predefinedObjs)
}
Component CommandObjects
consists of {
+ Package(root.frag.objs)
- Package(root.frag.parser.predefinedObjs)
- or Package(root.frag.predefinedObjects)
}
+Component DSL
+ consists of {
+   Package(root.mdsd.dsl) or {
+     Package(root.mdsd) and Class(".*DSL.*")
+   }
+ }
+Component FCL
+ consists of {
+   Package(root.mdsd.fcl) or {
+     Package(root.mdsd) and Class(".*FCL.*")
+   }
+ }
+Component FMF
+ consists of
+   Package(root.mdsd) and Class(".*FMF.*")
```

LISTING 3: Architectural abstraction specification modifications for the changes in Frag 0.7

For the following version of Frag (0.8) 7 inconsistencies were reported by our prototype. Another new component (TemplateEngine) was introduced which required twelve lines of DSL code and the top-level package `mdsd` was renamed to `mdd`, which required updates to the architectural abstraction specification at six places that were automatically highlighted by the consistency checks. The integration of partial support for such automatic architectural abstraction specification updates are a topic for future research.

Another change was that the code for the FMF component was moved into a package of its own, with only one class remaining outside this package. These changes account for 5 new lines of DSL code.

For the following release (Frag 0.91) the number of changes halved and no changes to the architecture were made. Because of this, no inconsistencies are reported and no updates to the architectural abstraction specification are required. A summary of all the changes that occurred during the evolution of Frag is shown in Table 5.4.

The data on how classifiers are distributed to components in this case can be found in Table 5.5. As the average and standard deviation suggest, the number of classes per component varies. However

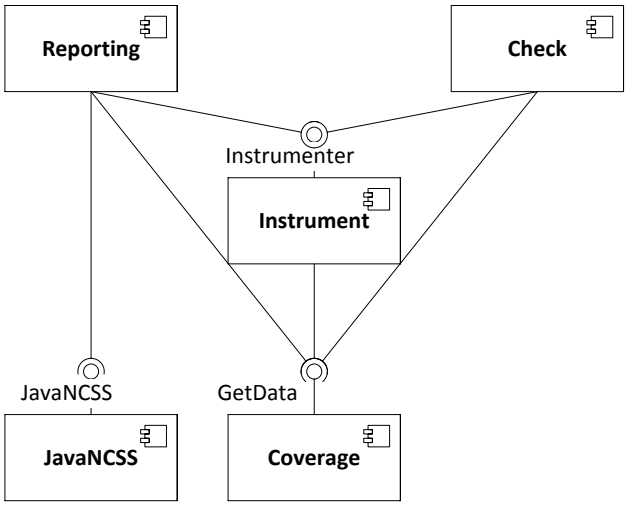

FIGURE 5.6: Simplified architecture overview of Cobertura 1.1

as the median is close to the average, the size of the components is evenly distributed between the minimum and maximum.

5.5.1.3 Case 3: Cobertura

Cobertura is a Java code coverage analysis tool that can be used to determine what percentage of your source code is exercised by a unit test suite. For Cobertura, we created an initial architectural abstraction specification on the basis of Version 1.0 through source code study.

With no prior architectural knowledge available, the effort to conduct the source code study and create an initial architectural abstraction specification was roughly 40 minutes and another 15 minutes of improving this specification. The total required time to adapt the architecture abstraction specification from Cobertura 1.0 step by step until Cobertura 1.9.4.1 was about 70 minutes.

A simplified version of the architecture of Cobertura is depicted in Figure 5.6.

Table 5.6 summarizes the evolution of architectural abstraction specification until the most recent Version 1.9.4.1. While initial versions only contained about 4000 lines of source code, Version 1.9.4, the last available version, has about 50000 lines of source code.

As Table 5.7 shows, the last version only has three more components than the initial version. All three introduced components were predated by a reported inconsistency that listed newly introduced classes and the only other reported inconsistency was a package that was removed in version 1.3 and thus could no longer be used in the architecture abstraction specification.

While the components are of similar size from Version 1.0 through to Version 1.9, in Version 1.9.4 the `JavaNCSS` component triples in size. This is caused by the new package `net.sourceforge.cobertura.javancss.parser`. As a result, the standard deviation increases from 3.24 to 14.06. This

Cobertura version	Files added	Files removed	Files changed	Total changes	DSL changes
1.0 ⇒ 1.1	18	10	23	51	no changes
1.1 ⇒ 1.2	23	1	2	26	no changes
1.2 ⇒ 1.3	12	5	8	25	1 new component and 1 minor change
1.3 ⇒ 1.4	25	3	3	31	no changes
1.4 ⇒ 1.5	20	2	0	22	no changes
1.5 ⇒ 1.6	17	3	1	21	no changes
1.6 ⇒ 1.7	18	1	0	19	no changes
1.7 ⇒ 1.8	47	11	1	59	1 new component
1.8 ⇒ 1.9	16	10	1	27	no changes
1.9 ⇒ 1.9.4	26	14	9	49	1 new component
1.9.4 ⇒ 1.9.4.1	1	0	0	1	no changes

TABLE 5.6: Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Cobertura

Cobertura Version	sLoC	# of Components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
1.0	4239	5	11	7,66	10,00
1.1	3491	5	8	5,87	7,00
1.2	3421	5	8	5,81	7,00
1.3	3406	6	5	2,61	4,50
1.4	3556	6	6	2,95	4,50
1.5	4003	6	6	2,77	5,50
1.6	3956	6	7	2,51	6,50
1.7	3997	6	7	2,17	6,50
1.8	17165	7	7,86	2,67	9,00
1.9	18179	7	9,14	3,24	10,00
1.9.4	51343	8	13,25	14,06	10,50
1.9.4.1	51342	8	17	17,10	10,50

TABLE 5.7: Cobertura: Average, median, and standard deviation for the number of classes per component

new parser might be a candidate for a component in its own right. However, as this parser is only used within `JavaNCSS`, we decided against interpreting this as an architectural change, as this parser is an internal implementation detail of the `JavaNCSS` component and in our opinion not relevant to the overall architecture of Cobertura. This is why we accepted the increased standard deviation and kept the, compared to the other components, large `JavaNCSS` component.

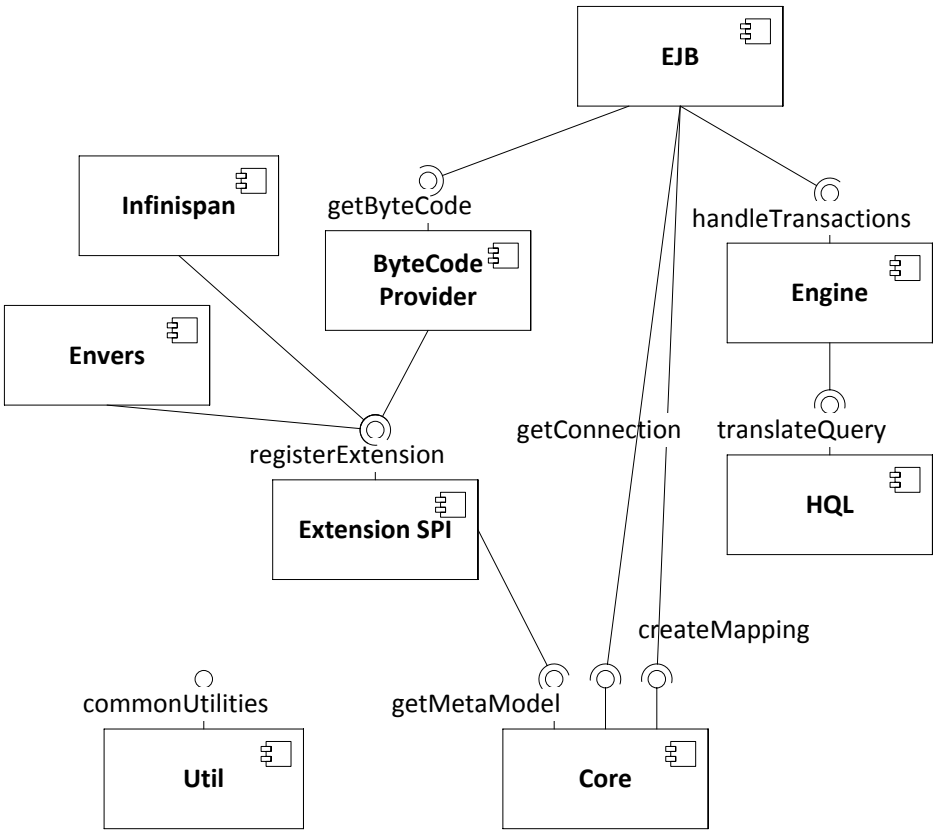

FIGURE 5.7: Simplified architecture overview of the Hibernate 4.1.10

5.5.1.4 Case 4: Hibernate

Hibernate is an open source Java persistence framework that supports object-relational mapping and querying of databases.

Although we found some information about the Hibernate architecture in the projects documentation, we still required a substantial amount of time for the initial source code study, which required about 3 hours. We then required 30 minutes for the creation of the initial architecture abstraction specification, another 150 minutes for improving the specification, and after that about 75 minutes for updating the specification according to the newer versions of Hibernate.

A simplified version of the architecture of Hibernate is depicted in Figure 5.7 and Table 5.8 presents the changes we encountered over the different versions. Especially interesting are the changes from version 3.6.10 to version 4.0.0alpha1 and from this version to 4.0.0.final, since these two changes constitute the transition between two major versions. While this version's changes consisted of a huge number of modified files and more than 150 new classes, only eight inconsistencies were reported which led to eight minor changes (added or removed packages) in the DSL code but no added or removed component. In our opinion, a part of the reason for this low level of change is the high level of abstraction that is used in the architectural component view for this case.

Hibernate version	Files added	Files removed	Files changed	Total changes	DSL changes
3.6.6 ⇒ 3.6.10	22	0	89	111	no changes
3.6.10 ⇒ 4.0.0.alpha1	20	86	2597	2703	3 removed packages; 3 new packages
4.0.0.alpha1 ⇒ 4.0.0.final	174	420	1270	1864	2 removed packages
4.0.0 ⇒ 4.0.1	19	2	70	91	no changes
4.0.1 ⇒ 4.1.0	54	5	221	280	no changes
4.1.0 ⇒ 4.1.1	12	5	192	209	no changes
4.1.1 ⇒ 4.1.2	15	5	720	740	no changes
4.1.2. ⇒ 4.1.3	20	3	391	414	no changes
4.1.3 ⇒ 4.1.4	12	2	87	101	no changes
4.1.4 ⇒ 4.1.5	132	1	92	225	no changes
4.1.5 ⇒ 4.1.6	22	0	87	109	no changes
4.1.6 ⇒ 4.1.7	8	0	54	62	no changes
4.1.7 ⇒ 4.1.8	34	11	147	192	1 new package
4.1.8 ⇒ 4.1.9	16	1	118	135	no changes
4.1.9 ⇒ 4.1.10	42	1	112	155	no changes

TABLE 5.8: Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Hibernate

As the standard deviation reported in Table 5.9 indicates, the components in this case are of varying size. The smallest component is realized by approx. 40 classifiers while the three largest components have more than 500 realizing classifiers.

5.5.1.5 Case 5: Freecol

Freecol is an open source game implemented in Java that is based on the popular game Colonization. While Version 0.4 has only 30.000 lines of code, the last version we examined has more than 100.000 lines of code.

We found no existing architectural information for FreeCol. This is why the necessary source code study required about 3,5 hours before we could create a first architectural abstraction specification. Updating the specification to the newer versions of FreeCol then required another 60 minutes.

The results for this example are presented in Table 5.10 and an architecture overview is shown in Figure 5.8.

Although the size of this project more than tripled, no inconsistencies were reported by our prototype and therefore no changes to the architectural abstraction specification (the DSL code) were necessary. The main cause for this stable architecture abstraction is that most changes to the game were improvements to the graphical user interface, improvements to the gameplay, or bug fixes.

Hibernate Version	sLoC	# of Components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
3.6.6	344k	9	225,89	219,91	174,00
3.6.10	351k	9	226,78	220,60	174,00
4.0.0.alpha1	343k	9	293,33	240,14	210,00
4.0.0.final	387k	9	298,67	261,32	189,00
4.0.1	389k	9	295,00	256,85	184,00
4.1.0	400k	9	296,33	258,13	184,00
4.1.1	402k	9	296,22	257,96	185,00
4.1.2	405k	9	301,11	262,16	191,00
4.1.3	407k	9	297,11	257,24	186,00
4.1.4	409k	9	297,22	257,48	186,00
4.1.5	410k	9	302,44	263,30	192,00
4.1.6	413k	9	298,11	258,78	187,00
4.1.7	416k	9	298,11	258,78	187,00
4.1.8	421k	9	306,22	249,67	189,00
4.1.9	423k	9	306,33	249,63	189,00
4.1.10	426k	9	308,22	252,77	190,00

TABLE 5.9: Hibernate: Average, median, and standard deviation (σ) for the number of classes per component

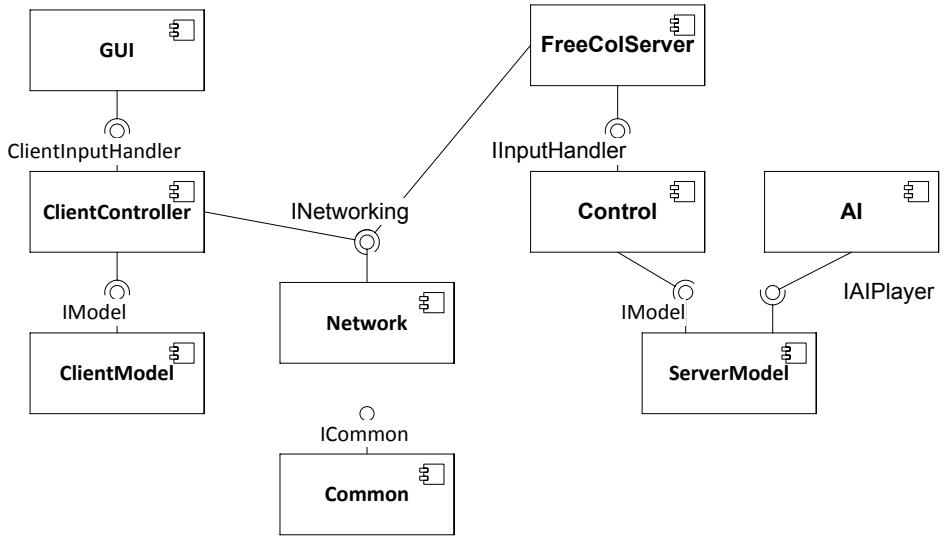

FIGURE 5.8: Simplified architecture overview of the FreeCol 0.10.7

Freecol version	Files added	Files removed	Files changed	Total changes	DSL changes
0.4 ⇒ 0.5.0	190	55	0	245	no changes
0.5.0 ⇒ 0.5.3	155	9	1	165	no changes
0.5.3 ⇒ 0.6.0	185	72	1	258	no changes
0.6.0 ⇒ 0.6.1	60	6	0	66	no changes
0.6.1 ⇒ 0.7.0	217	31	0	248	no changes
0.7.0 ⇒ 0.7.4	356	31	1	388	no changes
0.7.4 ⇒ 0.8.0	284	86	18	388	no changes
0.8.0 ⇒ 0.8.4	117	10	0	127	no changes
0.8.4 ⇒ 0.9.0	280	72	14	366	no changes
0.9.0 ⇒ 0.9.5	370	18	18	406	no changes
0.9.5 ⇒ 0.10.0	468	95	71	634	no changes
0.10.0 ⇒ 0.10.7	542	88	22	652	no changes

TABLE 5.10: Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Freecol

Freecol Version	sLoC	# of Components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
0.4	31k	9	25,44	40,51	9,00
0.5	42k	9	28,00	44,00	9,00
0.5.3	47k	9	28,22	44,20	9,00
0.6	52k	9	33,67	57,08	9,00
0.6.1	54k	9	37,56	61,42	9,00
0.7.0	60k	9	41,44	67,81	9,00
0.7.4	63k	9	42,56	70,48	9,00
0.8.0	74k	9	51,22	83,90	9,00
0.8.4	76k	9	52,00	84,83	9,00
0.9.0	84k	9	57,89	94,07	10,00
0.9.5	82k	9	56,89	91,57	10,00
0.10.0	92k	9	64,44	103,43	11,00
0.10.7	101k	9	73,44	119,64	12,00

TABLE 5.11: Freecol: Lines of java source code (in thousands), average, median, and standard deviation (σ) for the number of classes per component

Table 5.11 presents the data about component sizes for this case. The high averages of classes per component (compared to the medians) indicate that this architectural abstraction consists of a number of small components and a very few larger components. In this case the large components are `GUI` and `Common`. While the purpose of the first component is rather self-explanatory, the component "Common" provides functionality that is used on the client and on the server. That is, it provides functions that the client and server have in common. We focused our architectural abstraction on the "big picture" architecture of the system to convey an understanding for the whole system.

Project	#Clauses	Avg. Exec. Time (in ms)	σ (in ms)	Median Exec. Time (in ms)
Cobertura 1.9.4.1	19	176	257	116
Frag 0.9.1	41	283	281	219
FreeCol 0.9.5	21	704	647	455
Apache CXF 2.4	35	3010	1534	2353
Hibernate 3.6.6	96	3117	956	2757

TABLE 5.12: Execution times, standard deviation (σ) and other key data for implemented cases

5.5.2 Performance Evaluation

To validate our approach, we realized architectural abstraction specifications for the five open source projects explained in the previous section (see Table 5.12). As approaches like the one introduced in this chapter can potentially require a lot of computational effort, we measured the performance of our prototype tool-suite for the transformations in the five open source projects. Performance problems can be introduced for instance through the exponential growth of the execution time of the analysis according to the size of the model and the architectural abstraction specification. However, for regular usage of the approach an execution time below two minutes is acceptable. We measured the time it takes to execute the constraint checks and the transformations. Table 5.12 shows the execution times for the most recent version of each of the five open source cases which we obtained on a developer notebook (Intel i7 L620, 4 Gb RAM). We measured each execution time 100 times and calculated the average value. We also measured the minimum and maximum values, but as we observed only small deviations around the average values, so we only report the averages here.

The results from Table 5.12 suggest that the execution time increases with the number of clauses in the architectural abstraction specification and with the number of classes in the source code. The results also suggest that the approach is well applicable for even larger projects like Hibernate in the normal flow of software development. Thus we can state that for the example cases it is possible to generate an architectural component view with traceability links and check all artifacts for consistency within an acceptable amount of time (RQ5.5).

Please note that we did not report the time that is needed for extracting the class model from the source code, since this algorithm converts every class in the source code into an instance in the model. That is, this algorithm has a time requirement of $O(n)$, where n is the number of classes in the project.

5.6 Discussion

In this section we discuss the lessons learned from the five example cases from the case study as well as the results of the performance evaluation of our prototype. Furthermore we discuss the limitations of the approach and open issues in Section 5.6.2.

5.6.1 Lessons Learned

The cases discussed in Section 5.5.1 confirm that it is possible to create abstractions based on the generic filters defined in our DSL (RQ5.1). In all cases, from version to version a large number of changes have been applied, often in many different files. Still only small changes to the DSL were necessary, even for major architectural changes in the projects.

The times required to create an initial architectural abstraction specification differ between the systems where architectural knowledge was already available in other forms (Apache CXF, Frag) and the systems where we had no or only very limited architectural knowledge about the system prior to our case study (Cobertura, Hibernate, FreeCol). In our opinion, the time that was necessary to come from the initial specification to a satisfactory one depends on factors like the quality of existing architectural information and system size. For instance, while we could create an initial specification for Apache CXF based on an existing box diagram within 15 minutes, it took us long to create a specification that we were satisfied with. For Frag, on the other hand, we had first hand architectural knowledge as one of the authors is also involved in the development of Frag. Thus the creation of a satisfactory architecture abstraction specification took only 20 minutes before we could start comparing the different versions of Frag. Regarding RQ5.4 we can conclude that the effort of creating a suitable architectural abstraction specification varies heavily depending on the existing knowledge of the source code and architecture. If it exists, a small architectural abstraction specification can be created in about 15-20 minutes.

The consistency checking rules are an important means to automatically indicate that changes to the DSL might be necessary. In our finalized version, all necessary changes between different versions of the examples were automatically detected by the consistency checking. This reduced the necessity for manually searching for changes or the use of other tools like *diff*. In addition the inconsistencies are reported directly for the violating parts of the specification and thus direct the software architect to the origin of the problem at hand. The inconsistency that was reported the most while creating the example cases, and thus was the most helpful, were source code elements that were not considered in the architectural abstraction specification. In summary, in the example cases the consistency checks were very helpful in creating and maintaining architectural abstraction specifications (RQ5.3).

Traceability links are an important aid to find and understand the links between components and realizing classifiers between two version quickly. That is, without support through consistency checking and traceability links, in our 5 cases, the same low number of updates would have been necessary in our DSL, but the amount of manual searching and understanding of change impacts would have been substantially higher. We used them throughout all the examples to navigate between the architectural abstraction specification and the source code elements whenever an inconsistency was reported. The traceability links provide the foundation for all consistency checks and are especially helpful when source code elements are not considered in the architecture abstraction specification. So with respect to RQ5.2 we can state that: For these five examplary cases we are able to generate traceability links that were useful during the definition of architectural abstractions.

Once the architectural abstraction specification is defined, we are able to automatically create abstraction models from the source code. We noticed many of our component definitions are based on one to five *Package* rules. Packages are a major way of grouping multiple Java classes (besides Tagging interfaces and so on). The advantage of component definitions based on existing groupings like packages is that the architectural abstraction specifications can cope with many kinds of changes, as in an established software project the coarse grained (package) structure usually is stable. For this reason, only major changes require a change of the architectural abstraction specification. For example, the introduction of a new subpackage or a new class do not require any changes. Only the introduction of new major packages or new components requires architectural abstraction specification updates. However, while in our use cases the grouping based on *Package* rules was beneficial, this might not be always the case.

Our approach supports the creation of architectural abstraction specifications on different levels of abstraction. The data in Table 5.12 supports this claim. While we needed 41 clauses to map the 13k lines of code from Frag, we only needed 21 clauses to map the 103k lines of code from FreeCol and 35 clauses for mapping the 386k lines of code of Apache CXF to components. This indicates that the Apache CXF architectural abstraction specification is on a higher abstraction level than the one for Frag.

The five cases indicate that creating and maintaining architectural abstractions is easier for high-level abstractions and that generic filters like package-based or name-based filters are less likely to be changed. For example, name-based filters are unaffected by changes as long as the regular expression for matching the name is not affected. A `Package` rule that uses a regular expression like ".*model.*" only is affected if this exact part of the name is modified, while a `Package`-rule based on the fully qualified name of the package needs to be updated as soon as one of the packages on its path is modified.

However, it is not always possible to define architectural abstraction specifications solely using name-based filters like `Package` name filters and the union of their results (`or`). One example is

the Shell component in Listing 2 that consists of all the classes that contain the name "Shell" and all elements in the `root.frag.core` package that are used by `root.frag.Shell`. The definitions based on the relationships between classes have two disadvantages: Firstly, they are hard to read because it is unclear which classes match the specified filter. Secondly, relationship-based filters can have side-effects. A related class can reside within a package that is also targeted by a package-based or a name-based filter. This can be avoided by defining an exception in one of the filters. The evolution of relationship-based filters is similar to the already mentioned `Package`-filters that is based on fully qualified names. They need to be updated only if the class that is defined in the filter is moved, renamed, or deleted.

When we compare the statistical data for the different versions of the cases to the corresponding number of lines of source code, one can see that they develop in a similar manner. As the lines of code grow, the average size of classifiers and the median for the number of classifiers per component also grow. Because of this, we think that the added source code gets distributed among the different components. This negates the potential threat to validity that in our cases, all classes could be aggregated in a single component. In this case obviously no changes to the architectural abstraction would ever be necessary. As already mentioned, this is not the case in any of the presented cases.

5.6.2 Limitations and Open Issues

Our approach has limitations when being applied to architectural knowledge recovery and no prior knowledge about a software project exists. Under these circumstances our approach is only applicable after initial architectural knowledge has been acquired, since it does not provide an automated abstraction that can be used for refinement. This limitation does not reduce the applicability in a software development project where the focus lies on preserving architectural knowledge. In such cases, the required knowledge usually is created in an early stage of a software development project (i.e. this problem will not arise in the first place).

While we demonstrated the approach we presented in this chapter on cases that use programming languages supporting the structuring of source code (e.g., via packages in Java), our approach is also applicable for other languages that do not offer such features. The limitation that arises from the missing structuring of source code features is that the `Package` filter cannot be used. All other rules are still available and can be used instead. However, this limitation often increases the number of rules necessary to define an architectural abstraction specification, though.

In our five case studies, we documented the architectures on a high abstraction level. It is likely that a more detailed architectural descriptions of the same example cases lead to more changes during the evolution of the systems. This is partly reflected in the Apache CXF Transport case. However the main goal of our approach is to support the understanding of the "big picture". This is

the reason why we chose a high abstraction level for our case studies. We would like to investigate in future research in how far our approach is applicable for detailed design as well.

Furthermore, while our approach is designed to be independent of the source programming language by using UML as the respresentation of the source model and a number of tools exist for different programming languages that transform from source code into UML2 model elements, we currently use examples that are implemented in Java. The reason for this is that we currently only have implemented a transformation from Java to our input representation in EMF UML2.

5.7 Conclusion

In this chapter, we presented a semi-automatic approach for supporting architectural abstractions of source code into architectural component views. Our approach, supported through a DSL and MDD tooling, can automatically generate architectural component views from source code and supports traceability between the mapped artifacts. By creating architectural abstraction specifications with the DSL on different levels of abstraction, we are able to generate different abstracted views for one project. A major feature of our approach is its ability to cope with change. Only major changes, like newly introduced components, require an update of the architectural abstraction specification. Overall, our evaluations and experience show that architectural abstraction specifications can be created and maintained with low effort. This is due in part to the traceability links and inconsistency checking support. That is, we can check for inconsistencies between abstraction and code and identify the participating source code and architectural components in case constraints are violated.

Our approach has limitations when used for reengineering as knowledge about the source code and the design of a project are needed to create the architectural abstraction specification; in many reengineering approaches the main assumption is that such knowledge does not yet exist. Hence, our approach can be used together with these approaches: The reengineering approaches can be used for acquiring an understanding of a project, and our approach can be used to maintain and evolve an architectural view on the system once it has been sufficiently understood. We plan to investigate this relation in our future work.

In our future work we also plan to increase the usability of the presented approach by implementing support for other popular programming languages by providing the necessary transformations from source code to the UML2 class model. For this purpose we will focus on languages that are already supported in the Eclipse IDE like C++.

Based on the answers to the more detailed research questions in this chapter, we can give the following answer for RQ 2: With the approach we proposed in this chapter, we are able to support the architect during the creation and maintenance of architectural component views and through

the automatic generation of traceability links as well as automatic consistency checking we are able to reduce the risk for architectural drift and erosion of architectural component views and source code. However, while the architectural components view is very important, it is limited in the information it conveys.

In Chapter 6 we extend this approach towards the architectural pattern discovery and documentation. For this purpose we allow the developer to annotate components and connectors from the architecture abstraction specification with architectural primitives [ZA08] and then search pattern candidates that are built of these architectural primitives.

6 Semi-automatic Architectural Pattern Identification and Documentation Using Architectural Primitives

6.1 Introduction

As already mentioned in Chapter 5, during maintenance and evolution of a software system, a deep understanding of the system's architecture is essential. This knowledge about a system's architecture tends to erode over time [Jan+07] or even get lost. In a recent study Rost et al. [Ros+13] found that architecture documentation is frequently outdated, updated only with strong delays, and inconsistent in detail and form. They also found that developers prefer interactive (navigable) documentation compared to static documents. This also reflects our personal experiences as well as those of others. For instance, our colleague Neil Harrison shared the following story from his experiences with large-scale industrial systems: "Once upon a time I worked on a large system that was already a few years old. It had a well-defined architecture. When I started, I was given copies of three or four documents that described the architecture. In addition, I watched several videotapes in which the architects described the architecture. As a result, I gained a good understanding of the architecture of the system. I felt comfortable working in most parts of the system, including some of the more arcane parts of the code.

After a few years, I left the project to work on other things. But several years later I returned. I think it was about ten years later. The system was still being used and was under active development. Of course, it had changed greatly to add new capabilities and support changes in technology. Underneath it all, the original architecture was largely intact, but it was much more obscure. I wanted to refresh my architectural memory, so I asked around for the original memos and videotapes. Nobody had even heard of them. Critical architectural knowledge had been lost. People were actually afraid to change the original code, because they didn't understand how it worked. A very few old timers knew the original system, and they were the ones who dared change the early code. (I soon became one of them.)"

Software architecture documentation or, in case of lost architectural knowledge, software architecture reconstruction [DP09] techniques can be used to (re)establish the proper architectural documentation of the software system. An essential part of today's architectural knowledge is information about the patterns used in a system's architecture. Patterns can be seen as building blocks for the composition of a system's architecture [Bus+96; BJ94]. This is especially valid for architectural patterns or styles which describe a system's fundamental structure and behavior [LN95]. A considerable number of software architecture reconstruction approaches support software pattern identification [BJ94; BP00; Shu+96]. Most of these approaches (see e.g. [Heu+03; KP96; BP00; Phi+03]) focus on automatically detecting design patterns in the source code. Such pattern identification approaches are often restricted to design patterns that were identified by Gamma et al. [Gam+95] (GoF patterns). Architectural patterns, in contrast, convey broader information about a system's architecture as they usually are described at a larger scale than GoF patterns.

There are a number of important problems in automatic pattern identification in general and especially in architectural pattern identification. Existing approaches often only focus on the task of identifying a system's design patterns while the documentation of the reconstructed patterns and the future evolution of the system are not considered (which is just as essential as identifying an architectural pattern).

In addition, architectural patterns are often much harder to detect directly in the source code than GoF design patterns, as there is often a large number of classes involved in the implementation of the pattern and the variations between different instances of the patterns are very large. As a consequence of the large number of involved classes, there is a possibly huge search space for these patterns that grows with every class and increases execution times [DP09].

A big problem of pattern identification is the variability in pattern implementations. Only a very few pattern identification approaches consider pattern variations at all, and they are usually focused on GoF design patterns only [Wen+01; Wen03]. For instance, hardly any implementation of a system strictly adheres to the Layers pattern [Bus+96] as described in the textbook, but a huge number of systems are designed based on Layers. To give a concrete example, in the definition of the Layers pattern, a layer only has access to the functionality provided by the layer below it. However, this rule is often violated for cross-cutting concerns like performance, security, or logging. As a consequence, many layered architectures contain parts that do not strictly adhere to the Layers pattern. In addition to this, the Layers pattern suggests but does not in any way enforce clean interfaces between the layers. For these reasons, it is hard to automatically detect architectural patterns like Layers.

Another problem of automatic pattern identification is the accuracy of the approaches, which is often not sufficient. That is, some approaches treat pattern instances they find as candidates [Wen03]. However the likelihood of false positives increases with system size and can lead to precision values

around 40 percent [KP96] which means that 60 percent of the found pattern instances are false positives. This requires substantial manual effort to review the found pattern instances.

In the light of the aforementioned problems, we have split our original Research Question 3 and formulated the following, more detailed research questions:

RQ6.1 How far can a semi-automatic architectural pattern approach go toward the goal of identifying the patterns in architectural reconstruction?

RQ6.2 How far can a semi-automatic architectural pattern approach go toward the goal of maintaining documented pattern instances during the further evolution of a reconstructed architecture?

RQ6.3 In how far are the concepts and tools applicable in existing real-life systems?

RQ6.4 How efficient are the actual pattern instance matching algorithms that are based on primitives?

RQ6.5 Are primitives and an adaptable pattern catalog adequate means to handle the variability inherent to architectural patterns?

The main contributions of this chapter are, first, to suggest a novel semi-automatic architectural pattern identification approach that tackles the aforementioned problems that arise during the documentation and evolution of architectural patterns like the variability inherent to patterns, consistency between the documented architecture and the source code, and the large number of source code artifacts that are related to the implementation of architectural patterns. This is based on the architectural component views that we presented in Chapter 5.

Second, we show the approach's feasibility in terms of tool support (in the context of three open source case studies), and to study the performance of the approach (also in the context of these cases). We aim to assist the software architect during the reconstruction of architectural knowledge as well as supporting the architect in the documentation of the reconstructed architectural knowledge. After the architectural knowledge has been reconstructed and documented with our approach, we support the software architect in keeping the created architectural documentation in sync with the source code of the application. As Clements et al. [Cle+02] state, a strong architecture is only useful if it is properly documented in order to allow others to quickly find information about it.

Our proposed solution is an interactive approach for the semi-automatic identification and documentation of architectural patterns based on a set of Domain Specific Languages (DSLs). It consists of the following main components:

- *Architecture Abstraction Specification Language:* In the *Architecture Abstraction Specification Language*, which we discussed in detail in Chapter 5, the software engineers can semi-automatically create an abstraction of an architectural component view based on design models or during architecture reconstruction. To address the rich concepts and variations of patterns, we propose to use architectural primitives [ZA05] that can be leveraged by software engineers for pattern annotation during software architecture documentation and reconstruction. Architectural primitives are primitive abstractions at the architectural level (i.e. defined for components, connectors, and other architectural abstractions[1]) that can be found in realizations of multiple patterns.

- *Pattern Instance Documentation Tool:* Using the architectural primitive annotations, our approach provides a *Pattern Instance Documentation Tool* which automatically suggests possible pattern instances based on the architectural component view of a system and a pattern catalog.

- *Pattern Catalog DSL:* The pattern catalog contains templates of the architectural patterns to be identified. It is customizable, reusable and integrates support for pattern variability. Our approach leads to a reduced search space for patterns, as we search for patterns only in the created architectural component view instead of the source code.

- *Pattern Instance DSL:* Identified pattern instances are documented using the *Pattern Instance DSL* which uses the artifacts defined in the *Architecture Abstraction Specification Language* and the *Pattern Catalog DSL* to permanently store pattern instance documentations.

We automatically generate traceability links between the architectural abstractions and the source code (more specifically, the automatically generated class models of the source code), the architectural abstractions and the selected pattern instances, and the pattern instances and the pattern catalog. When artifacts are changed, the traceability links are used to automatically check the consistency of all the artifacts. Automated consistency checking aids the software engineers during the incremental architecture documentation process, when new artifacts are identified and documented. For example, the system automatically detects when the pattern catalog is used to customize an existing pattern and these changes cause an existing instance of this pattern to be no longer valid. The consistency checks are used throughout the evolution of the documented system and report any occurring violations within seconds.

[1] Today, the component and connector view (or component view for short) of an architecture is a view that is often considered to contain the most significant architectural information [Cle+02]. Taylor et al. [Tay+10] define *components* as architectural entities that encapsulate a subset of a systems functionality and/or data. Each component has an explicitly defined interface that restricts access to the component's functionality and data as well as explicitly defined dependencies on its required execution context. They define a *connector* as an architectural building block that is tasked with effecting and regulating interactions among components.

This chapter is structured as follows: In Section 6.2 we briefly explain architectural patterns and architectural primitives as our background. We give an overview of our approach in Section 6.3, and present it in detail in Section 6.4. In Section 6.5 we present three case studies of open source systems in which we have applied our approach to test its applicability. As it is crucial for our approach that it works smoothly in the working environment of the software designer during software design and development, we evaluate the execution time of our prototype in Section 6.6. In Section 6.7 we discuss lessons learned from the case studies and the performance evaluation as well as limitations of our approach. We conclude this chapter in Section 6.8.

6.2 Background: Patterns and Architectural Primitives

A significant aspect of documenting software architectures is the representation of architectural patterns [Bus+96; AZ05] and the closely related architectural styles [SG96]. In general, a pattern is a problem-solution pair in a given context. A pattern does not only document 'how' a solution solves a problem but also 'why' it is solved, i.e., the rationale behind this particular solution. Architectural patterns help to document architectural design decisions, facilitate communication between stakeholders through a common vocabulary, and assist in analyzing the quality attributes of a software system.

Common approaches for modeling architectural patterns are Architecture Description Languages (ADLs) [MT00], the Unified Modeling Language [Med+02], and formal or semi-formal approaches for the formalization of pattern specifications [EH99; Mik98]. As discussed in detail by Zdun and Avgeriou [ZA05], none of these approaches succeeds in effectively modeling architectural patterns, as they (1) are too limited in their abstractions to cover the rich concepts found in the patterns and (2) do not deal with the inherent variability of architectural patterns. To solve these problems Zdun and Avgeriou then proposed in [ZA05] to remedy the problem of modeling architectural patterns through identifying and representing a number of *architectural primitives* that can act as the participants in the solution that patterns convey. We use the term 'primitive' because they are the fundamental modeling elements in representing a pattern, and they are the smallest units that make sense at the architectural level of abstraction (e.g. specialized components, connectors, ports, interfaces). Our approach relies on the assumption that architectural patterns contain a number of architectural primitives that are recurring participants in several other patterns [MM03]. These primitives are common among the different patterns even if their semantics demonstrate slight variations from pattern to pattern.

In the previous work of our group, we provided modeling abstractions for each type of elicited architectural primitive [ZA05]. In this work, we propose a semi-automatic architectural pattern identification and documentation approach based on the architectural primitives. The benefit of using this approach during architecture documentation and architecture reconstruction efforts

is that the primitives can capture the rich concepts found in patterns as well as their inherent variability.

In the remainder of this chapter we use the term primitive in two different contexts. First, we use the term primitives to annotate the architectural component view with the primitive information. In this context we use the primitives, as described above, as fundamental modeling elements. Throughout the chapter we refer to this as primitive annotations. The second context is the description of architectural pattern templates. In this context we use the term primitive as a parameterizable version of the definition above to describe the properties of a pattern participant.

6.3 Approach Overview

Figure 6.1 shows the most important steps and tools in our approach. The central tool is the *Pattern Instance Documentation Tool*. Its goal is to document architectural pattern instances based on an architectural component and connectors view of a system that is annotated with architectural primitives.

The tool is semi-automatic, as it also receives manually edited inputs developed using the *Architecture Abstraction Specification Language*. In Chapter 5 we developed a basic version of this DSL that can be used to provide architecture abstraction specifications to incrementally create an architectural component view which abstracts over source code. In order to provide language independence we first automatically create a UML class model from the source code and then the software architect uses our DSL to manually create abstractions from this UML class model to create an architectural component view. This component view is then permanently stored as a UML components and connector model.

Traceability links between the class model and the architectural component view are automatically generated. Essentially, the DSL supports the specification of architectural components and connectors based on source code elements. In this chapter, we extended the *Architecture Abstraction Specification Language* to also enable software architects to incrementally annotate the created components and connectors with architectural primitives during architecture documentation or reconstruction. The final input for the *Pattern Instance Documentation Tool* is a reusable pattern catalog. Usually the pattern catalog is defined once and can then be reused many times. The pattern catalog contains a number of templates for architectural patterns. For the task of creating and editing pattern catalogs we defined the *Pattern Catalog DSL* (details in Section 6.4.1) that uses architectural primitives as the basis for pattern descriptions.

Based on the information from the architectural components and the pattern catalog, the *Pattern Instance Documentation Tool* (see Section 6.4.3 for details) automatically computes which patterns

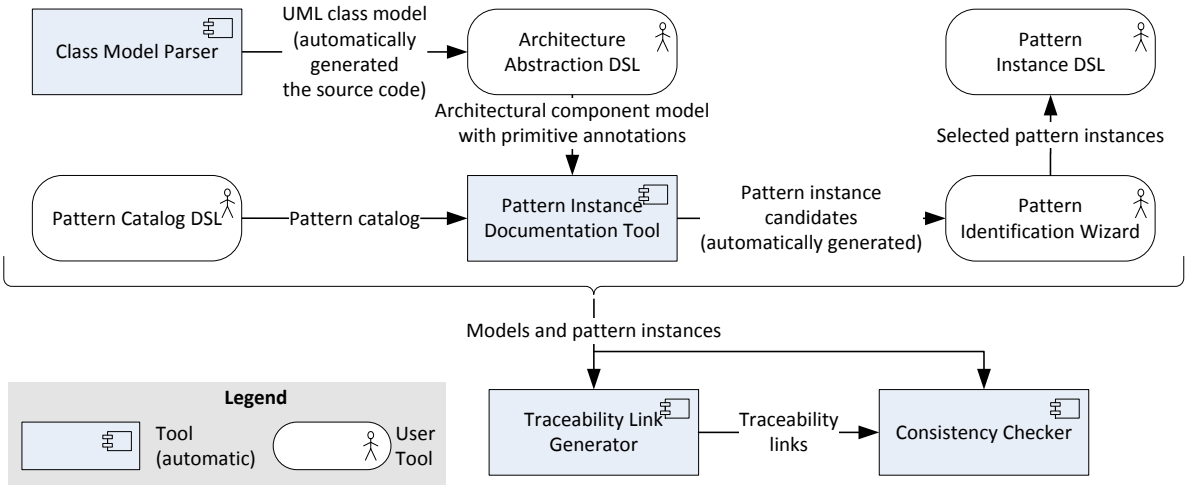

FIGURE 6.1: Overview of the approach

from the pattern catalog can be instantiated based on the architectural component view specification. Using a *Pattern Identification Wizard*, all pattern instance candidates are presented to the software architect. The software architect then chooses the candidates she wishes to document. The created pattern instance description, as well as the pattern instance candidates, are instances of another DSL, the *Pattern Instance DSL* (see Section 6.4.4 for details). This DSL uses elements from the *Pattern Catalog DSL* and the *Architecture Abstraction Specification Language* to describe pattern instance documentations. It is used to review and edit the documented pattern instances and pattern instance candidates, or add missing information, e.g. within the *Pattern Identification Wizard*.

Once a pattern instance is documented, it is automatically checked for consistency throughout further iterations of architecture documentation effort and the future evolution of the system using the *Consistency Checker*. Whenever the source code, the architectural components, or the pattern catalog are changed, our tools test if any consistency rules, defined by the architectural primitives, are violated or if any relations and constraints that are defined in the pattern catalog are no longer satisfied. As shown in Figure 6.1 the Pattern-Architecture Traceability Generator automatically maintains traceability links between all elements that are shared between the *Pattern Catalog DSL*, the *Architecture Abstraction Specification Language*, and the *Pattern Instance DSL*. To maintain traceability with the source code, the Code-Architecture Traceability Generator automatically generates traceability links between the architectural components and the source code based on the *Architecture Abstraction Specification Language*.

For all manual steps in the approach, our tools provide a number of features that ease the life of the software architect. This includes code completion for the DSLs, auto-complete for names of existing artifacts, and automatic generation of traceability links. Furthermore the system provides detailed information about violated constraints and the violating artifacts.

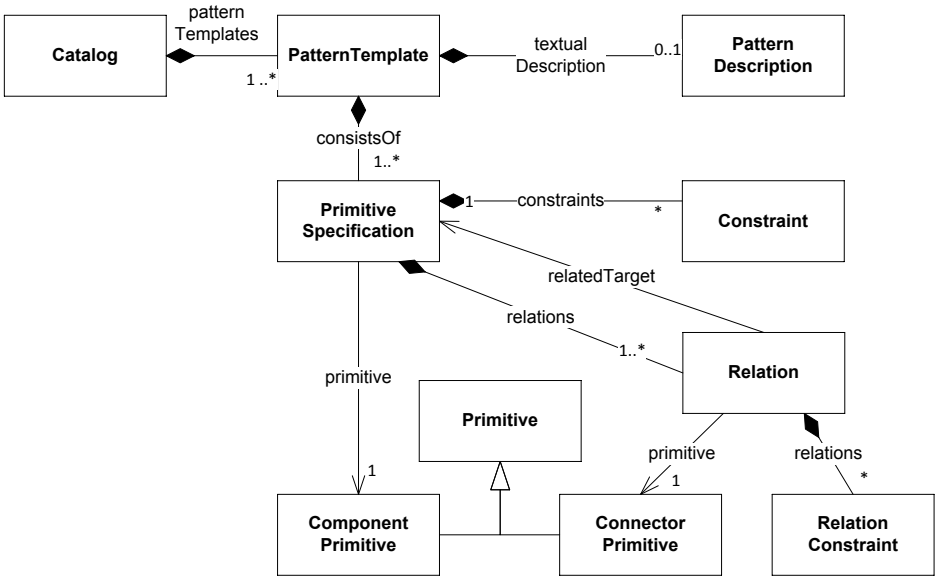

FIGURE 6.2: Excerpt of the Ecore model for the Pattern Catalog DSL

6.4 Detailed Description of the Approach

Our approach introduces a reusable pattern catalog that contains architectural patterns, an architectural component view that is annotated with architectural primitives, and pattern instances based on the pattern catalog and the architectural component view. In this section we describe the concepts and languages used for realizing these different parts of our approach in more detail. To illustrate our approach, we use a running example based on the open source game *FreeCol* [The11] which is a turn-based strategy game based on the old game Colonization, and similar to Civilization. The objective of the game is to create an independent nation.

The central tool in our approach is the *Pattern Instance Documentation Tool* as it is responsible for the computation of possible pattern candidates that are presented to the architect in order to document the architectural patterns found in the source code. It is described in Section 6.4.3; however, we describe the *Pattern Catalog* and the changes and extensions to the *Architecture Abstraction Specification Language* first, as their concepts are important for understanding the details of the *Pattern Instance Documentation Tool*.

6.4.1 Pattern Catalog

The basis of our approach are patterns that can be defined in a reusable pattern catalog. Figure 6.2 shows the most important parts of the Ecore meta-model for this *Pattern Catalog DSL*[2]. It supports the definition of patterns based on architectural primitives.

[2]All our DSLs are implemented using Eclipse Xtext 2.3.1 utilizing Eclipse Xtend and Java for model-transformation and constraint checking (using the Eclipse Xtext validation framework).

FIGURE 6.3: Example showing the MVC pattern in the Pattern Catalog DSL

Figure 6.3 shows an exemplary definition of the Model-View-Controller pattern as described by Fowler et al. [Fow02]. This pattern aims at decoupling the presentation from the business logic and the data. It consists of three different parts: The Model that holds the data, the Controller that manipulates that data, and the View(s) that display the data. This pattern defines that the View and the Controller have a relationship with each other and also that both have a relationship to the Model. Please note the usage of the *Grouping* primitive that, in the context of the pattern template, defines the role *Controller* as a group of components that together fulfill this role.

A pattern specification (i.e. the `PatternTemplate` in the meta-model) consists of a textual description of the pattern and one to many `RoleDefinitions`. Each `RoleDefinition` describes one role and its relations to other roles that are part of the pattern. Thus an instance of `RoleDefinition` has a *Rolename*, a *ComponentPrimitive*, an arbitrary number of constraints, and a number of `Relation` objects. Each `Relation` requires a `ConnectorPrimitive` and a target role and optionally holds a `RelationConstraint`. Currently three types of constraints are supported:

- *RangeConstraint*: this allows to define a lower bound (0 or greater) and an upper bound (1 to many - which is denoted by a *) for the occurrences of a role in a pattern instance description.
- *ExclusionConstraint*: this allows to define an exclusion: if a role is assigned in the pattern instance description one or more other roles must not be assigned.
- *RequiresConstraint*: this allows to define a requirement: if a role is assigned in the pattern instance description,one or more other roles have to be assigned as well.

This way `RoleDefinitions` can describe optional roles, by using a *RangeConstraint* with a lower bound of 0. Using the *RequiresConstraint* it is possible that a whole group of roles can be defined which all have to be assigned in the pattern instance documentation or none of the roles is assigned.

In Figure 6.3 three roles are shown: One with the name *Model* which specifies that a group of components belongs to this pattern and that the this group has no connector to elements fulfilling the roles *View* and *Controller*. One called *View* which specifies that one or more components are

TABLE 6.1: Overview of the defined primitives (excerpt)

Primitive Name	Annotation Target	DSL Keyword	Description
Component	(Component)	(Component)	Supertype for component primitives
Grouping	Component	Group	Component is part of a group of components
Layering	Component	Layering	Component belongs to a layer
Connector	(Connector)	connector to	Supertype for connector primitives
Callback	Connector	is callback for	Register a callback with another component
Indirection	Connector	indirection to	Indirection to another component
Aggregation Cascade	Connector	aggregates	Component aggregates other components
Composite Cascade	Connector	composition of	Component is a composite of other components
Virtual-Connector	Connector	virtually connected to	An indirect connection to another component
Shield	Connector	shield for	Prevents direct access to a set of other components

part of this pattern which have connectors to the *Model* and the *View*, and finally a group of components that are named *Controller* with connectors to *Model* and *View*. While the version of the MVC pattern that is shown in Figure 6.3 only allows a single controller, there might exist versions of MVC that utilize multiple *Controllers*. In order to extend the pattern template to allow this variants, only the *RangeConstraint* that follows after *Controller* has to be modified from (1) to e.g. (1 .. *) to allow the unbounded assignment of *Controller*s in the pattern instance description.

6.4.2 Architecture Abstraction Specification Language

In Chapter 5 we introduced a DSL which we now extend and evolve. It is intended to describe a software system's architectural component view with its connectors and primitive annotations through architectural abstraction specifications.

We now build on this approach by extending the architectural component view and allowing the architect to (manually) annotate the abstractions for architectural components with architectural primitive information.

Table 6.1 provides an overview of the primitives that are used in the examples and cases in this chapter. In our prototype we have implemented all primitives defined in the previous work of our group [ZA05].

In the FreeCol example (Section 6.5.1) we identified 10 architectural components. Figure 6.4 shows the definition of one of these architectural components, `ClientController`, which contains the source code package `root.net.sf.freecol.client.control`. The `or`-statement in the code represents the

```
Component ClientController
consists of
{
  Package ([...].client.control)
  or
  //[...]
}
is in Group ( Controllers )
connector to Server
implemented by
  //[...]
  or
  Class([...].server.FreeColServer)
connector to GUI
connector to Model
connector to ServerModel
//[...]
```

```
Component Interpreter
consists of
{
  Class (".*Interp")
  or
  {
  Package(root.frag.core)
  }
}
is a Shield for CommandGroup
is a Shield for Parser
indirection to CommandObjects
indirection to FileCommandObjects
connector to Parser
```

FIGURE 6.4: Example for an architectural abstraction for the `ClientController` component of the FreeCol system [The11] as well as an example for an architectural abstraction for the `Interpreter` component of the Apache CXF [Apa] case study (Section 6.5.3).

union operation and indicates that more source code elements are contained in this component which we do not show here for brevity reasons.

In addition, this architectural component has been manually annotated with the *Grouping* primitive and been added to a group called `Controllers`. This definition also contains a set of connectors for this architectural component (also abbreviated). Similar to the ball and socket notation in UML 2, connectors in the *Architecture Abstraction Specification Language* have direction, as they pertain to an architectural component and can target either a single architectural component or a group of components that are annotated with the *Grouping* primitive. The connectors of the `ClientController` component shown in Figure 6.4 were automatically generated based on relations in the source code. This component, respectively the group it is annotated with, is a candidate for the role `Controller` in the Model-View-Controller pattern (Figure 6.3) as this architectural component has connectors to an architectural component `GUI` (which fulfills the constraints for the role View) and to the components `Model` and `ServerModel` which form a group called `ModelComponents`. This group fulfills the constraints for the role `Model`.

The components shown in Figure 6.4 also exemplify the annotation of architectural components with primitives. While the `ClientController` component is annotated with the *Grouping* primitives, which indicates that it is part of a group of components that belong together, the `Interpreter` component is annotated as a *Shield* that shields the `Parser` component and a *Grouping* called `CommandGroup` from direct access. The Xtext grammar of the *Architecture Abstraction Specification Language* can be found in Appendix C.

6.4.3 Pattern Instance Documentation Tool

In this section we describe the *Pattern Instance Documentation Tool* of our approach. When architects intend to create a pattern instance description based on the pattern catalog, the architectural

primitive annotations are used to automatically search for possible patterns that are then presented in an Eclipse Wizard. There the architect can select the pattern she deems appropriate. The wizard then creates a pattern instance description based on the selected pattern and prefills all values that can be automatically determined.

The *Pattern Instance Documentation Tool* first reads the architectural component view and the pattern catalog. The tool then checks for each pattern template in the pattern catalog if the pattern's constraints are satisfied in the architectural component view. The basic algorithm for this task is shown in Algorithm 1. It is invoked for every pattern template in the pattern catalog and returns a pattern candidate if the pattern's constraints can be satisfied.

Algorithm 1 Pattern Evaluation Algorithm

```
 1: procedure evaluatePattern(patternTemplate,model)
 2:     candidate := new Candidate()
 3:     for each primitive in patternTemplate do
 4:         comps := getAnnotatedComponents(model, primitive)
 5:         if checkComponentConstraints(primitive, comps) then
 6:             updateCandidate(candidate, primitive, comps)
 7:         else
 8:             return null
 9:         end if
10:     end for
11:     for each primitive in candidate do
12:         if ¬checkRelationConstraints(primitive, candidate) then
13:             return null
14:         end if
15:     end for
16:     return candidate
17: end procedure
```

In Algorithm 1, for each architectural primitive of the type `ComponentPrimitive` that is used in the pattern specification, the method getAnnotatedComponents is used to find all architectural components that are annotated with the specified primitive. After this, the method checkComponentConstraints is used to check if the constraints for the primitive (as specified in the pattern catalog) can be satisfied. When all component primitives of a pattern have been satisfied, the algorithm checks if the `ConnectorPrimitives` defined in the pattern template and all constraints defined for these `ConnectorPrimitives` are satisfied. If any constraint or primitive of a pattern template cannot be satisfied using the primitive annotations in the architectural component view, the evaluation of the pattern template is aborted. If all constraints are satisfied and all the pattern template's primitives exist as primitive annotations in the architectural model, the pattern template is accepted and a candidate is created. This pattern candidate is not a complete pattern instance description, as precomputed complete pattern instances would lead to a potentially huge number of pattern candidates. Each pattern candidate holds information about a single pattern template that can be found in the architecture and contains a map that holds, for each role, the architectural components that can be assigned to this role (based on the role's primitive and the architectural component's primitive annotations). Algorithm 1 has a worst-case runtime complexity of $O(N \times M)$, i.e., in simplified form we can write $O(N^2)$. Because of the quadratic complexity

of the algorithm we performed an evaluation of our approach's performance which is presented in Section 6.6.

In the Model-View-Controller(MVC) example (see Figure 6.3), this means that when creating an instance, it is necessary that the architecture abstraction specification matches the constraints for all `RoleSpecifications` that were defined in the pattern template. For example, to assign the `Model` role, at least one group primitive annotation needs to exist in the architecture abstraction, while any defined component is viable for the role `View`. To satisfy the connector primitives for the role `View` at least one architectural component is needed that has connectors to at least two distinct groups of components. Similar requirements are necessary for the role `Controller` which requires that a number of architectural components are grouped. To fulfill the role, this group needs to have a connector to the `Model` components and also needs to have at least one connector to the possible candidates for the `View` role. The pattern template for the MVC pattern also forbids a relation between the model and the view as well as the model and the controller. However it is not efficient to check for the absence of a relation during the search for possible pattern candidates. At this time no knowledge or limited knowledge (in case of roles that can only be fulfilled by one component or grouping) about the concrete pattern instance exists and it would be necessary to check all possible assignments (all possible pattern instances) for the absence of this relation.

The list of accepted pattern instance candidates is then presented to the software architect together with information about the pattern template that is the basis for the candidate. Once the software architect selects one or more pattern instance candidates, the *Pattern Instance Documentation Tool* tries to automatically assign components for the roles of the selected candidates. If not all the roles of a pattern instance candidate can be automatically assigned, the *Pattern Instance DSL* is used to present the unfinished pattern instance to the software architect and she needs to complete the pattern instance by selecting one of the viable architectural components for each unassigned role. Once all roles of a pattern instance are assigned, it is permanently stored for documentation and later use – again using the *Pattern Instance DSL*.

To support continuous consistency checking, all the checks that were performed in the *Pattern Instance Documentation Tool* during the creation of a pattern instance are performed for each selected pattern instance both during the following iterations of the architecture documentation and subsequent evolution of the system.

As discussed in Section 6.1 and 6.2, variability is inherent to all architectural patterns and their implementations. Two possible examples of pattern variations are models with pattern instances that contain additional architectural elements not described by the pattern or pattern instances where parts of the pattern have been omitted (an example for this case is shown and discussed in Section 6.5.3). The *Pattern Instance Documentation Tool* ignores additional architectural elements by default and is only influenced by additional architectural elements if the pattern template(s) in the pattern catalog explicitly forbid certain relations. If a pattern implementation omits parts that

```
Pattern Instance: ModelViewController
      Model : ModelComponents
      View : GUI
      Controller : Controllers
```

FIGURE 6.5: Example instance of the MVC-pattern for the program "FreeCol"[The11]

are specified in the corresponding pattern template in our pattern catalog, in order for the *Pattern Instance Documentation Tool* to identify the pattern, it is necessary to mark the missing part of the pattern template as optional. We plan to support the search for incomplete implementations using an heuristic approach in the future.

6.4.4 Pattern Instances

For creating and persisting pattern instances we implemented the *Pattern Instance DSL* that references elements from the *Pattern Catalog DSL* and the *Architecture Abstraction Specification Language*. The pattern instances hold the information which architectural components relate to which parts (roles) of the pattern from the pattern catalog. In Figure 6.5 we show an example instance of the Model-View-Controller pattern that we identified in the architectural components of FreeCol. This instance uses the group `Controllers` (which holds the `ClientController` from Figure 6.4) for the role with the same name. It assigns the `Model` role to the `ModelComponents` group and the `View` role to the architectural component `GUI`.

For every pattern instance documentation, our *Pattern Instance DSL* expresses and permanently stores traceability links between the elements from the pattern instance description, the pattern template and its roles from the pattern catalog, as well as the architectural components from the architectural component view that are assigned to roles. This is done implicitly as the existing artifacts from the *Architecture Abstraction Specification Language* and the *Pattern Catalog DSL* are directly referenced when a pattern instance is documented. For the example shown in Figure 6.5, this means that `ModelViewController` is a reference to the pattern template from the pattern catalog and `Model`, `View`, and `Controller` are references to the roles that are defined in this pattern template. While `ModelComponents`, `GUI`, and `Controllers` are references to components or `Groupings` specified in the *Architecture Abstraction Specification Language*. All of these traceability links are navigable by the architect in the tool and allow a quick navigation between the different artifacts of our approach. E.g. the architect can navigate from the documented Model-View-Controller pattern instance in the *PatternInstanceDSL* to the underlying pattern template specified in the *PatternCatalogDSL* by Ctrl+clicking the pattern template name in the pattern instance documentation. This also works for roles and assigned components, where a Ctrl+click on the role `View` will also open the template for the Model-View-Controller pattern, while a Ctrl+click on the assigned component `GUI` will bring up the specification of this component in the architectural component view.

Based on the traceability links, our tool suite checks consistency of pattern instance descriptions with the architectural components, which in turn are consistency checked against the source code artifacts, as well as the underlying pattern templates. These checks also include the aformentioned checks whether a pattern template defines constraints on the relations of its participating roles (see Section 6.4.3). If the consistency checks detect a violation, an error is raised and the affected part of the pattern instance description is highlighted.

6.5 Case Studies

In this section we present three open source system case studies to better illustrate our approach and to study the practical applicability of our approach to do architecture reconstruction and documentation for existing, non-trivial software systems. Finally, the case studies are also used as a basis for the performance evaluations in Section 6.6. In the first case study we documented the architecture of the open source game FreeCol, which was partly presented as a running example before. In the second case study, Frag, we explain the documentation of the architecture of an open source programming language implementation – which was developed in our group – for a number of evolution steps. In the third case, we study the documentation of architectural patterns on an open source system with approximately 390.000 lines of source code in more than 2000 classes, namely Apache CXF.

6.5.1 Case Study: FreeCol

FreeCol is a turn-based open source multi-player game implemented in Java. The implementation uses a Client-Server architecture. For all games the clients connect to a server to play. While a local server is started for single-player games, a dedicated server can be used for multi-player games.

Through our architecture reconstruction and documentation effort we identified the architecture shown in Figure 6.8. We identified 10 components and their relationships. The client consists of a graphical user interface `ClientGUI` that displays the game based on a domain model (`Model`). All actions, that are executed by the user, are forwarded to the `ClientController` which updates the model accordingly and also notifies the server about the changes. For this purpose the `ClientNetworking` exposes a server API to the client. Calls to this API are then forwarded to the server using a message-based protocol that is implemented by the `Networking` component. On the server-side received messages are forwarded from the `ServerNetworking` to the input handler which is part of the `ServerController` where `MessageHandlers` are used for handling the input. These update the server's game state which is realized in the `ServerModel` and notify other players if

necessary. This means that the API provided to the client actually does not have a direct counterpart on the server-side as the implementation of this fuctionality is distributed on the different `MessageHandlers`. On the client side, the `ClientController's InputHandler` reacts to messages that are received from the server and updates the GUI and model accordingly.

During the architecture documentation we decided to group the components that provide similar functionality for the client and the server. One obvious choice for grouping were the `ClientController` and `ServerController` which we grouped as `Controllers`. The second group we annotated was the `ModelGroup` which consists of the `Model` and the `ServerModel` component. Another set of components that together provide important functionality are all components concerned with networking which together provide the facilities for the communication between clients and the servers. Naturally we grouped them in a group called `Networking`.

During the source code study of this project we noticed that the `ClientNetworking` component as well as the `ServerNetworking` component both use the `Networking` component to handle the input they receive. Thus we annotated the connector between `ClientNetworking` and `Networking` as well as the connector between `ServerNetworking` and `Networking` with the *Indirection* primitive.

Using these primitive annotations, the Pattern Instance Documentation Tool then proposed a number of patterns from the pattern catalog. Among these are the already mentioned Model-View-Controller (MVC) pattern, as well as the Broker [Zdu+04], Application Controller, Page Controller, Proxy, Transform View, Template View, and Two Step View patterns (which are all discussed by Martin Fowler [Fow02]).

After manual analysis of the architectural component model, we concluded that the MVC pattern matches the architecture's user-interface best, as alternatives like the Page Controller pattern are similar to the MVC pattern but do not match FreeCol's intended architecture. For the MVC pattern we selected the `ClientGUI` as `View`, the `Controllers` grouping as `Controller` and the `ModelGroup` grouping for the `Model` role. A very similar option would have been the Page Controller pattern. Our current pattern templates for the MVC and Page Controller patterns only differ in one relation between the role View and the role Controller which is forbidden in the Page Controller pattern and required in the MVC pattern. As already discussed in Section 6.4.3 the absence of relations can only be viably checked for existing pattern instances and the difference for these two patterns cannot be detected by the Pattern Instance Documentation Tool which reports both patterns as possible pattern candidates.

In order to demonstrate this, we also documented an instance of the Page Controller with the same role assignments that we used for the MVC pattern. However the consistency checker raises a constraint violation for the Model role once this pattern instance documentation is checked. This is shown in Figure 6.6. A similar constraint violation might occur when the application changes over time and a new dependency between two components is implemented in the source code.

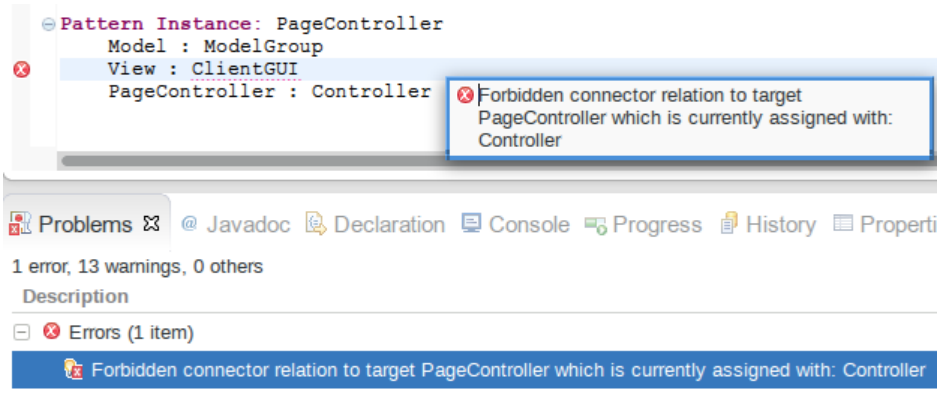

FIGURE 6.6: FreeCol case study: Page controller pattern instance with constraint violation[The11]

This first leads to a constraint violation in the *Architecture Abstraction Specification Language*. If the architect then decides that an evolution/change of the architecture is necessary (rather then changing the source code), she adds the according connector to the architecture abstraction specification. This, in turn, may lead to a constraint violation like the one shown before. This works in a similar manner if dependencies between different components and thus their connectors are removed.

After the attempt to document an instance of the Page Controller pattern we also documented the Broker pattern [Zdu+04]. This pattern's context are distributed objects and the transparent invocation of remote objects. For this purpose a client-side requestor and a server-side invoker are used that hide the implementation details of the network communication from the engineer using a marshaller. In order to provide type-system transparency the requestor usually is a proxy for the object that is invoked on the server.

Our pattern template, which is based on the description of Zdun et al. [Zdu+04], was able to describe the implemented architecture. For this pattern instance the assignment of roles had to be done manually as the *Pattern Instance Documentation Tool* found more than one possible option for each role.

As the description of the Broker pattern is based on the *Component* primitive, the architect has to select between all of FreeCol's components when assigning the first role. After the role `Client` was assigned to the `ClientController` component, the tool suggested the 3 components `Model`, `ClientGUI`, `ClientNetworking` for the `ClientProxy` role. After the selection of the `ClientNetworking` component as `ClientProxy`, our tool automatically suggested the `Networking` component for the `Transport` role. For the `ServerProxy` role, the tool provided a choice between the `ClientNetworking` and the `ServerNetworking` component. At the current point of time, however our tool does not use other means than structural information and thus cannot automatically select the `ServerNetworking`. The last remaining `Server` role was automatically assigned to the `ServerController` component. This results in the following assignment for the documented pattern instance: `ClientController` as Client, `ClientNetworking` as ClientProxy, `Networking` for the Transport role, `ServerNetworking`

```
Pattern Template Broker
consists of:
Client: Component
connector to ClientProxy
ClientProxy: Component
connector to Transport
Transport: Component
ServerProxy: Component
connector to Transport
Server: Component
connector to ServerProxy
```

```
Pattern Instance: Broker
Client: ClientController
ClientProxy: ClientNetworking
Transport: Networking
ServerProxy: ServerNetworking
Server: ServerController
```

FIGURE 6.7: FreeCol case study: Broker pattern template and Broker pattern instance description

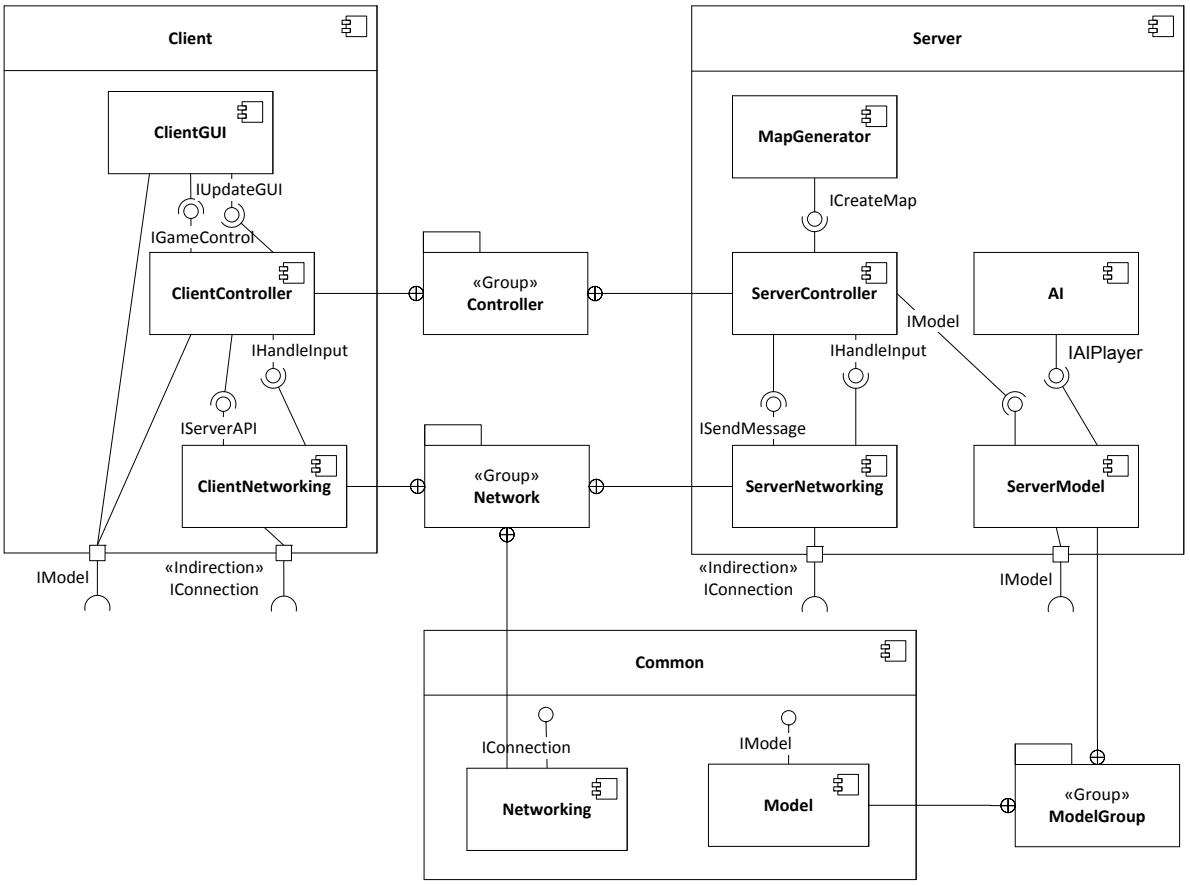

FIGURE 6.8: FreeCol architecture overview [The11]

as ServerProxy, and `ServerController` as Server. Figure 6.7 shows the template for the Broker pattern in the Pattern Catalog and the pattern instance documentation for the implementation found in FreeCol.

This case study shows the applicability of our tool for a medium sized system with about 100k lines of source code. The annotation of components with architectural primitives proved to be a straightforward task in this case, as the chosen annotations came quite naturally during the source code study. After the creation of the architectural component view, the documentation of the two architectural patterns required only little effort. Both were automatically suggested by the *Pattern Instance Documentation Tool*, and, while there were no alternatives found for the Broker pattern,

a number of alternatives were suggested for the MVC pattern. The case study also shows that the distinction of similar user interface patterns like Model-View-Controller or Page Controller on this level of abstraction can be a challenging task and requires human judgment. However, our approach aids the software designer in selecting the appropriate patterns by providing traceability links from the architectural information to the source code elements.

6.5.2 Case Study: Frag

Frag [Zdu11] is a dynamic programming language that is designed to be tailorable, support the creation of DSLs, support Model-driven Development, and the Frag interpreter is embeddable in Java, in which Frag also is written. We applied our approach to document Frag's architecture. We started by creating an architectural component view and pattern instance documentations for Frag version **0.6**. In this first iteration of our architecture documentation effort, we identified 4 architectural components and annotated these with primitive information.

As Frag is an interpreted language, the most important architectural component is the `Interp` component. It provides two interfaces. On the one hand, it allows the embedding of Frag in any Java program, and, on the other hand, it provides the functionality to execute the commands that were given as input via the components `Client` and `Shell` or via the `IEmbeddingFrag` interface. As the `Interp` uses different command objects to execute the received commands, we annotated the involved connectors with *Indirection* primitives. In addition the connectors to `Interp`'s provided interfaces are annotated with the *Shield* primitive as the `Interp` component shields the access to the `Parser` and `CommandObjs` components.

Pattern Documentation

On this architectural component view we used our *Pattern Instance Documentation Tool* to identify all pattern candidates. Based on our pattern catalog, our tool provided the following candidates: Facade [AZ05], Indirection Layer [AZ05], and Interpreter [AZ05]. The Facade pattern simplifies the access to a complex subsystem and decouples the client code from the actual implementation of the subsystem. While Facade is often used as design pattern, it can also be used on architectural level as sometimes access to a whole subsystem can be provided by an architectural component that provides a simplified interface for this subsystem to the rest of the application.

An Indirection Layer differs from the Facade as it is intended to hide the actual implementation of a subsystem and should not be bypassed, while a Facade still allows direct access to a subsystem. Similar to a Facade, it is possible that an Indirection Layer [AZ05] holds additional logic or performs additional tasks. The Interpreter pattern defines a class-based representation for a grammar along with an interpreter to interpret the language defined by the grammar [Fre+04]. Although the

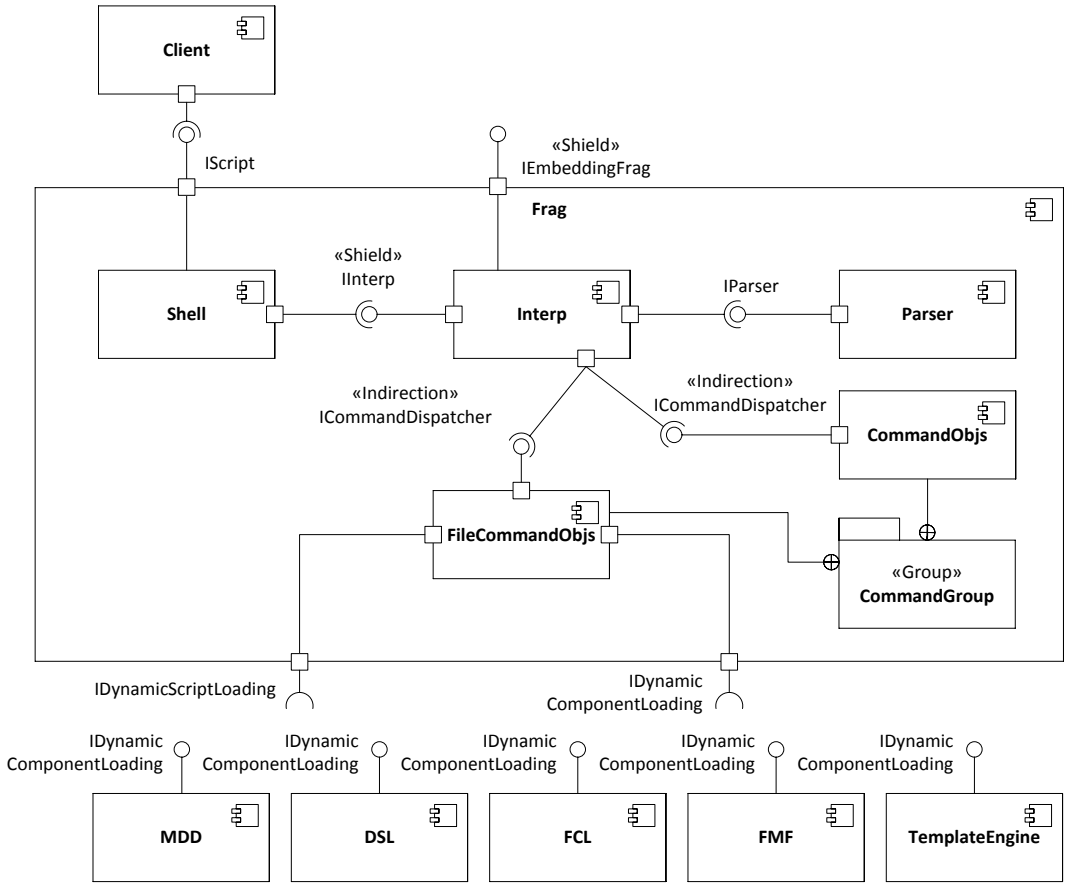

FIGURE 6.9: Architectural component view for Frag 0.91

Pattern Template
Interpreter
consists of:
Client: **Component**(0 .. *)
connector to Interpreter
Interpreter:**Component**(1)
shield for Expressions
indirection to Expressions
Expressions: **Component**(1 .. *)

Pattern Template
Indirection
consists of:
Client: **Component**
connector to Proxy
Proxy: **Component**
indirection to Target
shield for Target
Target: **Component**

Pattern Instance: Interpreter
Client : Shell
Interpreter : Interpreter
Expressions: CommandGroup

FIGURE 6.10: Pattern templates for the Interpreter and Indirection patterns as well as the pattern instance of the Interpreter pattern in the Frag example

patterns have different intents, from the perspective of the *Pattern Instance Documentation Tool*, all three candidates are plausible for the architectural model of Frag because the structural descriptions of the patterns (see Figure 6.10) are similar.

For both patterns in Figure 6.10 a number of possible variants for the description of the patterns exist. For the Interpreter pattern one variant would be that the Expressions that implement the language are grouped into one architectural component instead of a group of components. Another variant could be that the Expressions are implemented in the architectural component that fulfills the `Interpreter` role and the `Expressions` role is omitted. In the same way, variants of the Indirection Pattern are possible where e.g. the *Shield* might not be necessary.

The pattern instance we selected from these candidates is the `Interpreter` pattern because of multiple reasons. The first indication is the name of the reconstructed `Interpreter` component, and secondly this architectural component dispatches to the component, containing command objects, and this execution of commands matches the Interpreter pattern better than it matches the Facade or Indirection pattern.

After this selection, the *Pattern Instance Documentation Tool* automatically uses the `Interpreter` component for the role `Interpreter` from the pattern template and the `Shell` component as the role `Client`. The role `Expressions` had to be assigned manually as more then one possible assignment exists in our architectural component view. For the role `Expressions` we did select the component `CommandObjects`.

Architecture Evolution

In order to study consistency checking during architecture evolution, we then updated Frag's source code to version **0.7** and let the *Consistency Checker* test for inconsistencies. In a first step, our *Consistency Checker* reported a number of classes that existed in the source code but were not considered in the architectural abstraction as well as a package that was referenced in the architecture abstraction specification but no longer existed (as it had been renamed). The renamed package resulted in an update in the architecture specification where the reference to the package was changed accordingly. After a source code study of these new classes, we introduced four additional components to the architecture abstraction called `FileCommandObjects`, `MDD`, `DSL`, and `FCL`. Furthermore we added a connector from `Interpreter` to `FileCommandObjects`, one between `FileCommandObjects` each of the other new components. In addition our source code study had revealed that the *Interpreter* acted as a *Shield* for the `FileCommandObjects` and that the `Interpreter` now also used this component to execute commands. While the `FileCommandObjects` component utilized the other new components using dynamic loading. This is why we added a *Shield* and an *Indirection* primitive annotation for the `FileCommandObjects` component.

After these changes the *Consistency Checker* reported that the new components were not part of any documented pattern instances and suggested the `FileCommandObjects` as another participant of the documented `Interpreter` pattern. Specifically the component was suggested to be also assigned to the `Expressions` role of the documented `Interpreter`.

We then continued this process until we reached Frag's current version 0.91. In version **0.8** the *Consistency Checker* again reported new classes, which led to another two new components in the architecture abstraction specification that are connected to the `FileCommandObjects` component, as well as a package that had been renamed. This required an update to the architecture abstraction specification. After updating the source code from version 0.8 to version **0.91** the consistency

TABLE 6.2: The number of traceability links created or deleted by the *Traceabiliy Link Generator* during each evolution step.

Version change	Number of created traceability links	Number of deleted traceability links
Frag 0.6 → 0.7	122	113
Frag 0.7 → 0.8	53	22
Frag 0.8 → 0.9	59	91

checker did not report any inconsistencies as all new classes were already covered by the architecture specification and thus no changes to the architecture were necessary.

Traceability

Whenever we changed the architecture abstraction specification, the *Tracebility Link Generator* recalculated all traceability links. As shown in Table 6.2, for each evolution step (version change) the generator created and removed a substantial number of traceability links. This is also true for the evolution step from Frag version 0.8 to Frag version 0.91 where the changes to the source code resulted in 59 new and 91 deleted traceability links although no changes to the architecture abstraction specification had occurred. Keeping these traceability links manually up-to-date requires a substantial effort by the software architect or developer.

Summary

During multiple iterations we identified a total number of 10 architectural components and their connectors. The final architectural component view for Frag is shown in Figure 6.9.

This case illustrates that annotation with primitives can easily be done while documenting a systems architecture using our DSL-based approach. It shows how the consistency checks support the architect during the future evolution of a system once it's architecture has been documented using our approach.

6.5.3 Case Study: Apache CXF

Apache CXF is an open source Web services framework that is developed in Java and supports a wide variety of protocols like e.g. SOAP and RESTful HTTP. We used the architecture overview that is available at the CXF web-site[3] as a basis, and incrementally improved and annotated the architectural component view.

[3] http://cxf.apache.org/docs/cxf-architecture.html

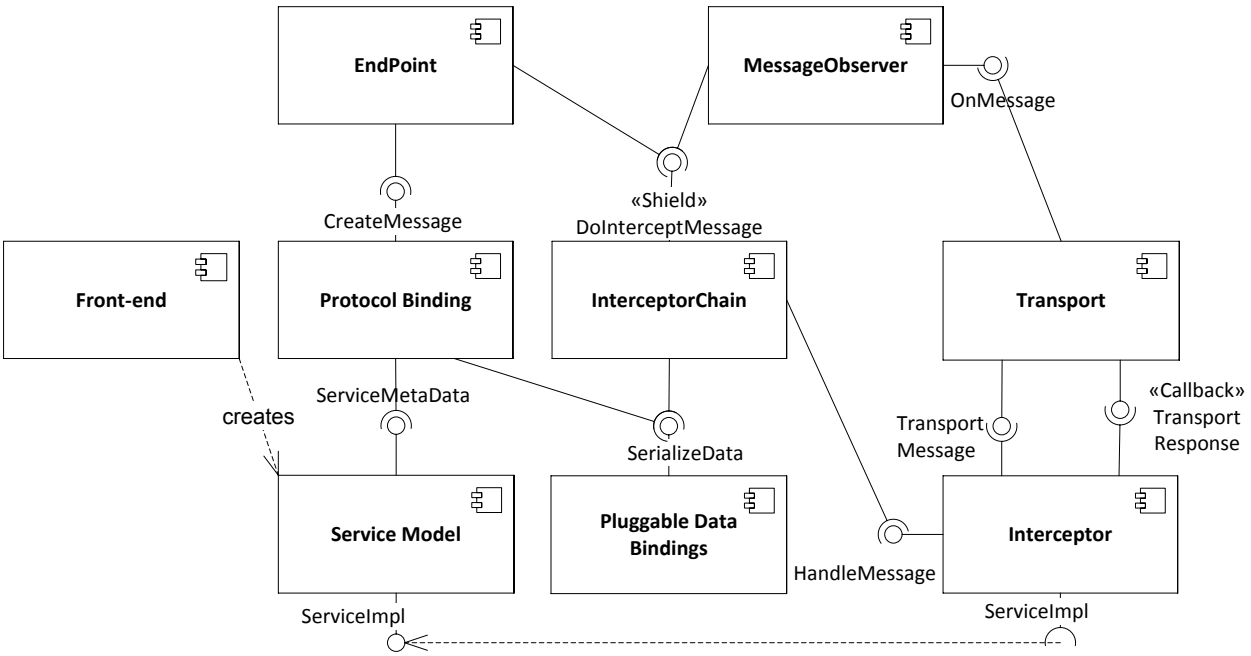

FIGURE 6.11: Apache CXF architecture overview [Apa]

The architecture of CXF 2.4.3 is built around an interceptor chain that is configured to handle all incoming request and outgoing responses on the server-side and on the client-side. As already mentioned it supports different protocols by allowing different protocol bindings and uses different transports to send and receive messages. This means that whenever Apache CXF receives the request to call a specific service the interceptor chain is configured to contain the necessary interceptors for the protocol and so on. The invocation is forwarded through the chain until finally one interceptor in the chain calls the service that was discovered using the service model and then another interceptor uses a conduit on the transport to send the result of the service call to the requesting client as a reply.

During our architecture documentation, we identified a number of cases where components realized characteristics of architectural primitives and annotated the components accordingly.

In particular, the `InterceptorChain` is the only component that accesses the interceptors in the architectural component `Interceptor` and thus was annotated as a *Shield* for the `Interceptor` component. Here, we combined all Interceptors that are implemented in Apache CXF into the `Interceptor` component. Another possibility (i.e., possible variant of the architectural primitive model) would have been to treat each Interceptor as an individual component and use the *Grouping* primitive annotation to combine them. The resulting architectural component view is shown in Figure 6.11.

During the source code study, we also found that the Transport Response connector between the `Interceptor` and the `Transport` components is actually a callback and thus was annotated with the *Callback* primitive.

After we finished the documentation and annotation of the abstracted components with architectural primitive information, our *Pattern Instance Documentation Tool* automatically found possible candidates for the patterns Facade [Gam+95] and Interceptor [AZ05; Cur+04]. The Interceptor pattern's [AZ05; Cur+04] intent is to increase a system's flexibility and extensibility by allowing to transparently updating the services offered by a framework [AZ05]. Applications can register interceptors by adding or removing interceptors to or from a dispatcher at runtime. The dispatcher then notifies the interceptors of events that are sent by the framework's core.

The Facade pattern actually is a design pattern that is used on an architectural level in different variants throughout the literature [AZ05; Fow02; Sch+00]. Selecting the Facade pattern, defined as a Client that uses a Shield to access a specific group of architectural components, is an option in this case, as the `Interceptor` component is actually hidden in a Facade-style. However, in this case, the more sophisticated Interceptor pattern seemed the better choice, as (1) indicated by the component names and (2) obvious after closely inspecting the intent of the respective components and classes. While creating the new pattern instance, we discovered that our description of the Interceptor pattern was too narrow. Our description consisted of a `Caller` component, a `ChainHandler`, one or more `Interceptor` components that are "shielded" by the `ChainHandler` and a `Callee`. When creating the pattern instance documentation, we started to assign the roles as follows: the `Interceptor` component to the `Interceptors` role, the `InterceptorChain` as *ChainHandler* role, and `EndPoint` as `Caller`. However after the assignment of the `ChainHandler` role, the automatic consistency checks detected a constraint violation with respect to the pattern template. The role `Callee` was still unassigned. However when looking at the architectural component view, no suitable candidate for the `Callee` role could be identified and thus additional time was invested in studying the implementation of the pattern in the source code. This manual study, during which we made extensive use of the automatically generated traceability links, came to the result that Apache CXF's implementation of the `Interceptor` pattern does not include a `Callee`, as the Interceptor Chain handles all the logic. So, we modified our template of the pattern by allowing zero to one `Callee` instead of exactly one.

In the implementation of Apache CXF, the client and the server both use an instance of the interceptor chain, although they configure different interceptors. This required us to assign both the `EndPoint` and the `MessageObserver` for the `Caller` role which resulted in another constraint violation as our pattern template that was based on the literature only allowed to assign one `Caller`. In order to account for this, we updated the pattern template to allow multiple callers in the pattern description by adding a multiplicity of `(1..*)` to the *Caller* role. Once the description was updated, we assigned the roles like this: The architectural components `Endpoint` and `MessageObserver` as `Callers`, the `InterceptorChain` was assigned the role of `ChainHandler` and the `Interceptor` component was assigned to the role with the same name. The role `Callee` was not assigned.

We expected that, using our approach, we would also find an instance of the Broker pattern [Zdu+04]. However, we did not identify this pattern – neither using the Pattern Instance Documentation Tool nor by manual identification of participating architectural components. The cause is that the structural aspect of the Broker pattern, as documented so far in our pattern catalog, does not exist in Apache CXF, because it does not use a static setup for handling requests and responses. Apache CXF uses its `InterceptorChain` on the client and on the server and configures the `Interceptors` accordingly. So while it shows the behavior of a Broker pattern this is not reflected in its structure. As a result, we extended our pattern catalog further to also include this variant of the Broker, and finally selected both Broker and Interceptor using the *Pattern Instance Documentation Tool*. We note that the Broker pattern could potentially be identified more precisely, if we would also include behavior information. We will investigate the problem how to integrate behavior information with our approach in future works.

This case illustrates, for an open source system with a substantial number of classes, how the annotation with architectural primitives can be performed during the documentation of a systems architecture using our DSL-based approach. It also illustrates how the pattern catalog is incrementally improved and extended (here one improvement of the Interceptor pattern description and one new variant of the Broker pattern have been explained), to show how our approach can deal with pattern variability. In our future work, we plan to extend our approach with a distributed pattern repository, in which all user updates are stored, so that different users of our approach can benefit from pattern variants documented by others.

6.6 Performance Evaluation of the Pattern Instance Documentation Tool

For the practical applicability of our approach it is crucial that it works smoothly in the working environment of the software designer during software design and development. To test the applicability of our approach in practice, we measured the time it takes our *Pattern Instance Documentation Tool* to find pattern instances for our case studies and in 5 larger (with respect to number of components) synthetic component models. For the synthetic component models we used the basic structure of the Apache CXF case study and created multiples of the number of components from the case with varying component names and some additional random interconnections. Measurements indicate that the time required to search for pattern candidates increases with the size of the component model. However our prototype is able to search for patterns in synthetic component models with more than 350 components in reasonable time while the component models in our case studies do not exceed the number of 12 components.

For all the examples we used the same pattern catalog and primitives (explained above) which contained templates for 15 architectural patterns from the literature [Zdu+04; Fow02; Bus+96;

TABLE 6.3: Results of the performance measurements for the case studies and larger synthetic models (in milliseconds, each executed 1000 times)

Example	Number of Arch. Components	σ	Average	Median
Frag	10	0.84	1.37	1
FreeCol	10	1.52	7.21	7
Apache CXF	12	1.02	8.96	9
Synthetic model 1	24	1.73	11.23	11
Synthetic model 2	48	2.64	11.18	11
Synthetic model 3	96	4.58	19.46	19
Synthetic model 4	192	4.88	34.50	33
Synthetic model 5	384	5.42	70.59	69
Synthetic model 6	768	93.38	331.66	321
Synthetic model 7	1536	148.14	1334.02	1302

AZ05] which includes architecture patterns like MVC, Broker, ApplicationController, PageController, Interpreter, Layers, and WrapperFacade. We measured the time it takes to run the *Pattern Instance Documentation Tool* a thousand times for each of the case studies and synthetic models. To obtain realistic results in a software developer environment, the measurement was performed on a developer notebook (Intel i7 L620, 8 Gb RAM) running Fedora 20 using Eclipse Kepler 4.3, Oracle Java 7. In Table 6.3 we present the number of architectural components, the standard deviation σ, the average, and median values of the execution time for all test cases. We do not report minimal and maximal values as the standard deviation is small compared to means and medians.

Our results indicate that our approach is usable even for larger component models (usually component models have not more than 5-20 components) on an average developer machine. We did not test varying the sizes of the pattern catalog, as our *Pattern Instance Recovery Tool* iterates through the pattern catalog with a loop, calling Algorithm 1 for each loop iteration, meaning that the performance of this complete loop is directly proportional to the size of the pattern catalog.

6.7 Discussion

In this section we briefly discuss the lessons learned from the three case studies and the performance evaluation.

6.7.1 Lessons Learned From the Case Studies

The three case studies show the applicability of our approach for three different types of software and different kinds of architectures. While FreeCol [The11] is a multiplayer game with a graphical user interface and a client server architecture, Apache CXF [Apa] is a web-service framework with

an architecture based on an interceptor chain. The third case study, Frag [Zdu11], is a dynamic scripting language implemented in Java that is built around an Interpreter architecture. The cases also vary in size as Frag has about 10.000 lines of code, while FreeCol is much bigger and has about 100.000 lines of code. The biggest system in the example cases is Apache CXF, which has about 350.000 lines of code. While all three case studies have about the same number of architectural components in the component view, they differ in the abstraction level on which architectural patterns are documented.

Before applying our approach to the three case studies, we manually created our initial pattern catalog based on architecture patterns from the literature. The process of creating a pattern template for a documented pattern consisted of the time necessary to understand the pattern (which is always necessary) and the effort to describe the pattern's structure using our *Pattern Catalog DSL*. In our experience, the description of the pattern with the *Pattern Catalog DSL* required about fifteen minutes per pattern. Ideally the pattern catalog is publicly available and maintained by the community in order to be reused and adapted by individual users.

During our case studies we had to evolve and improve our pattern catalog twice, giving us the opportunity to test the effort required for adapting or creating a new pattern (variant). The necessary effort to create a new pattern variant consists of selecting the original template and then modifying the new variant which in total required not more than a few minutes for the Broker variant. A simple relaxing of constraints as discussed for the Interpreter pattern required only a single change in the pattern catalog without any need for closing and restarting our tool, as changes to the pattern catalog are automatically propagated.

For us the source code studies of the example systems naturally led to the architectural primitive annotations we reported, and no additional effort was required to specifically search for possible options to add primitive information. However, somebody without knowledge about architectural primitives probably requires initial effort to learn about the architectural primitives and their functions before the annotation of architectural primitives during architecture reconstruction and documentation. Compared to the manual identification of architectural patterns, our approach also requires the architect to execute a source code study in order to create an architectural component view and thus could potentially require the same amount of effort. However, the case studies (Section 6.5) showed that the identification of architectural components in source code can be done in an iterative fashion and does not require the architect to study the complete source code at once, while manually identifying architectural patterns in the source code often requires to study a huge amount of classes at once and hence is more challenging than our approach where the architectural patterns are identified in the architectural component view. This is also shown in the case studies where the number of source code artifacts that are related to the implementation of the architectural patterns ranges from 169 to 2340 (see Table 6.4).

TABLE 6.4: Number of source code artifacts (classes and interfaces) compared to architecture artifacts (components and connectors) which need to be considered during architectural pattern identification.

Arch. Pattern	Source code artifacts	Architecture level artifacts
MVC (FreeCol)	459	10 components and 13 connectors
Broker (FreeCol)	169	10 components and 13 connectors
Interceptor (Apache CXF)	2340	12 components and 19 connectors
Interpreter (Frag)	174	10 components and 10 connectors

Once the architectural components and patterns are documented, our approach provides the software architect with automatically generated traceability links and automatic consistency checking. As already discussed in Section 6.5.2 and shown in Table 6.2, even for the Frag case study, the number of traceability links that needed to be updated with each new version was between 75 and 235. Manually creating and updating these traceability links would be a tedious and error prone task.

Table 6.4 shows the discrepancy in the number of elements that have to be considered when identifying architectural patterns on the level of source code and on the level of architectural components for our case studies.

Once the architectural components were documented and annotated with architectural primitive information, the documentation of architectural patterns based on this primitive information required only two manual steps: The selection of suitable pattern candidates from the list of pattern candidates that were automatically provided by the Pattern Instance Documentation Tool and then assignment of all the pattern roles which could not be automatically assigned by the tool.

Regarding our detailed research questions we can draw the following result:

RQ 6.1 Regarding the semi-automatic identification of patterns during architecture reconstruction, we could show the feasibility of our approach through the implementation of our prototype and through our case studies which exemplify the documentation of architectural patterns during architecture reconstruction. However, this approach requires a reusable pattern catalog as its basis. While we created an initial pattern catalog based on the literature, the creation and maintenance of this catalog require some effort which might hinder the adoption of this approach. Thus tools for sharing and maintaining pattern catalogs are required to ease the adoption of this approach.

RQ 6.2 Regarding the maintenance of architectural patterns during the evolution of a reconstructed architecture, we can state that our approach supports further architecture evolution once patterns are documented. This is exemplified in Case Study 6.5.2. As discussed in the limitations below, while our personal experience from the execution of the case studies indicates a reduced effort for documenting and maintaining architectural patterns

during the evolution of a system, we currently cannot not provide scientific evidence that our approach reduces the required effort. We will conduct a controlled experiment to investigate this topic further in the future.

RQ 6.3 In our case studies, we have shown the applicability of the approach for three different types of existing real-life systems with different project sizes. Therefore it is likely that our approach can be generalized to other similar cases. However, the component models in our case studies all have a similar number of components. As discussed in the limitations below, a significantly higher number of components in the component models might lead to a high number of pattern candidates and thus diminished benefit for the users of the approach during the identification of patterns.

RQ 6.4 Regarding the efficiency of the actual pattern instance matching algorithms, the performance evaluation in Section 6.6 shows that our prototype is sufficiently efficient to be used for architectural pattern identification on common developer computers for artificial component models with 300 and more architectural components and thus should be efficient for day-to-day use.

RQ 6.5 With respect to the adequacy of the primivites and the adaptable pattern catalog to handle the variability of architectural patterns, we can state that our case studies show that the concept of primitives can be applied to document architectural patterns and, as already discussed in Section 6.5.3, that only a small effort was necessary for evolving our pattern catalog during the case studies. Furthermore, during the creation of the initial pattern catalog and throughout the case studies, we were able to express all patterns based on the primitives in our Pattern Catalog DSL.

The case studies of FreeCol and Apache CXF showed a limitation of the current approach which is based on structural information only. For some patterns like the Page Controller and MVC patterns, which only differ in one relation, that is required in the MVC pattern and forbidden in the Page Controller pattern, it is hard to distinguish the structural differences during the computation of pattern candidates. This is because the forbidden relations cannot be taken into account during the creation of pattern candidates; however, later in our tool chain, our consistency checks for the documented pattern instances would have detected the constraint violation. The problem of distinguishing structurally similar patterns will be improved in our future work by also considering behavioral models of architectural patterns.

The pattern templates in our initial pattern catalog are based on the pattern descriptions from various sources in the literature (e.g., [Zdu+04; Fow02; Bus+96; AZ05]). Sometimes the templates from the literature are appropriate, and sometimes manual modifications are required. For example, during our case studies we could directly use the templates of the MVC, Broker, and Interpreter patterns we created based on the available literature, while it was necessary to modify the template

for the Interceptor pattern and to create a new variant of the Broker pattern that was suitable to describe its implementation in Apache CXF.

While our approach supports the software architect during the source code study with information on which source artifacts in the architecture abstraction specification are not yet covered and provides traceability links for source artifacts that are already covered, it does work semi-automatically and still requires the software designer to perform a source code study. While automatic approaches try to free the developer of this burden, they usually discover a substantial number of false positives that have to be checked and corrected by the software designer. This leads to the necessity of doing a source code study anyway. Our approach on the other hand focuses on supporting the architect during the documentation of the architecture with tool support for the documentation as well as partial automation of the documentation steps. This includes the automatic generation of connectors between architectural components based on the relations between the components in the source code as well as the automatic suggestion of architectural patterns that match the structure implemented in the documented system. While these suggestions also contain false positives, they do not consist of complete instances (e.g. a suggestion that holds all possible instances of the MVC pattern), but only a list of patterns and if the software architect selects a pattern for documentation, our prototype of the approach supports the software architect during the assignment of roles. This includes providing a list of possible role-assignments based on the already existing role-assignments as well as automatically assigning roles where possible (see Section 6.5.1 for examples).

Later on, during the evolution of a system, automatic approaches usually have to start from scratch, while an architecture that is documented using our approach, is automatically checked against the system's source code without any additional effort. While we cannot provide any quantitative data on the benefits of consistency checks, they have been proposed and used in different contexts for almost 20 years now [Xu+13; Hei+96; MS97]. In addition our approach provides automatically generated traceability links for the documented architecture. In a recent controlled experiment [JZ14], traceability links between architectural component models and the source code have proven to be highly beneficial for architecture understanding.

While the main use case of our approach are systems without existing architecture documentation, it can also be used to formally document other existing (informal) architecture documentation to check if all consistency constraints are fulfilled. A combination with other complimentary forms of architecture documentation like architectural decisions is possible as well.

6.7.2 Threats to Validity

In addition to the limitations already discussed above as lessons learned from our case studies, our case studies and performance evaluations have the following main threats to validity:

- The case study might not be representative to show the general applicability of the approach. As already discussed in the lessons learned, we tried to mitigate this threat by choosing cases from different application areas with varying sizes.

- As our approach requires input about the architectural primitives from the software designers, our results strongly depend on the quality of information provided by the software designers. We strive to improve the input quality by providing tools that support the software designer during the software architecture documentation, but ultimately our approach relies on the assumption that it is substantially easier to model with or detect the architectural primitives than patterns. Our experience so far shows that this assumption is justified.

- At this point we did not perform an evaluation of the applicability of the approach with other users, however we present three extensive case studies that showcase the applicability of the approach for three already existing systems.

- The synthetic models used in the performance evaluation might not be representative. We tried to mitigate this threat by using a real world model as a basis and created multiples of the case with custom component names and additional randomly created interconnections. In addition, we also measured the performance for our three case studies which are existing, realistic systems of varying size and which yield similar results.

- The pattern catalog used in the performance evaluation might not be representative. In order to reduce this risk, we used the pattern catalog that we initial created based on architectural patterns from the literature and that included all changes that were discussed in the example cases. As Algorithm 1 is executed for each pattern template, the execution time has a direct relation to the size of the pattern catalog.

- The effort necessary to create and maintain a useful pattern catalog might be large enough to hamper the usage of our approach. We tried to mitigate this risk by making the manipulation of the pattern catalog easy. The pattern catalog DSL is straightforward to use and the only required knowledge is the same as the one required to use our approach – knowledge about pattern primitives. However, we cannot fully eliminate this risk and other approaches faced this kind of problem before.

- The possiblity remains that the benefits do not outway the costs in the real world. A comparison in a controlled environment would be needed to contrast our appraoch's effort and the effort required to manually perform the same tasks. While we intend to perform this comparison as a controlled experiment in the future, our personal qualitative experience from performing the case studies shows an initial effort that is slightly higher than purely manual documentation for documenting the architectural patterns and a significantly reduced effort in maintaining the documented patterns during architecture evolution through the automatic consistency checking and the automatically maintained traceability links. This initial higher

effort stems from the requirement to gain an understanding of the primitives as well as the need to learn to use our three DSLs. However this additional effort is only required when applying the approach for the first time. For our controlled experiment, we will follow the guidelines proposed by Kitchenham and Wohlin [Woh+12; Kit+02]. The planned experiment will consist of a control group and a treatment group. While the control group will perform at least one architectural recovery and at least one architectural evolution task manually (using only an IDE but not our approach), the treatment group will perform the same tasks using our approach in addition to using an IDE.

- We applied the approach in three case studies that describe systems of different sizes. However, their architectural component models all consist of about ten components. A thread to validity of this approach is that this approach might not scale to systems which have a much higher number of architectural components, contain more architecture patterns, or are much larger in terms of source code size (like large-scale industrial systems). In this chapter, we only studied this scalability aspect in terms of the performance measurements for our Algorithm 1 which indicate acceptable performance for synthetic architectural component models with up to 350 components. However, the threat to validity that a huge number of components, combined with a high number of primitive annotations, might lead to too many possible patterns and thus leads to a diminished benefit for the users of the approach, remains.

6.8 Conclusion

In this chapter we have presented an approach for the semi-automatic documentation of architectural patterns based on architectural primitives. While other approaches automatically detect design patterns in the source code, we require the architect to semi-automatically create an abstraction of a architectural component view that is annotated with architecture primitive information. This raises the abstraction level of the input on which we automatically search for patterns. It also reduces the number of found pattern candidates as well as the search space for our automatic *Pattern Instance Documentation Tool*, as the number of architectural components is significantly smaller then the number of objects in a system. As we use architectural primitives as the basis for our pattern templates in our pattern catalog and in the architectural component view, our search can make use of this additional architectural information and the constraints that are captured by these primitives. Once a pattern instance is documented, our approach subsequently performs automatic consistency checking. We applied our approach in three open source systems case studies to show the applicability of the approach. Our pattern catalog supports the definition of patterns based on primitives, is reusable, supports pattern variability, and can be customized and is extensible. To use our approach, an initial investment in creating a pattern catalog (patterns and pattern

variants) is required. Our performance evaluation results show that the approach is applicable on a typical developer machine during software design and development, even for very large model sizes.

Our case studies (Section 6.5) show that the concept of architectural primitives can be applied to document architectural patterns during architecture reconstruction (RQ6.1) and support the further architecture evolution once they are documented (RQ6.2, as exemplified in the Frag case study in Section 6.5.2).

As we could apply our approach for three systems which all implement different types of applications with different project sizes, it is likely that our results can be generalized to other similar cases (RQ6.3). We could show the feasibility of the approach with the implementation of our prototype which was used to perform the case studies described in this chapter (RQ6.1, RQ6.2) and is described in detail in Section 6.4. In addition to this, the performance evaluation of our prototype (see Section 6.6) shows that algorithms for semi-automatic identification of architectural patterns are sufficiently efficient to be used for architectural pattern identification on common developer computers and thus should be sufficiently efficient for day-to-day use (RQ6.4). With respect to RQ6.5 we found only a small effort was necessary for evolving our pattern catalog during the case studies. This leads us to answer Research Question 3, where we asked whether we could support the software architect in the identification and documentation of architectural patterns during implementation and throughout the evolution of a system: Our approach helps the software architect in the identification and documentation of patterns and aids in keeping the documented pattern instances consistent during the evolution of a software system through automatic consistency checks and through traceability links between the documented architectural pattern instances, the architectural component model, and the source code.

We also plan to further investigate approaches for collaboratively editing and sharing the pattern catalog among users and in the community. At the moment we only support structural primitives that either annotate components or their connectors but no behavioral information. The additional integration of behavior primitives, as for instance introduced by Kamal et al. [KA08], is a topic for future research.

7 Supporting Software Evolution by Integrating DSL-based Architectural Abstraction and Understandability Related Metrics

7.1 Introduction

Software systems must evolve constantly or they will become obsolete [Leh89]. During the evolution of software systems, software architectures tend to erode as requirements change or new features are implemented [Par94] (also known as *architectural erosion*). In addition, the intended, documented architecture and the implemented architecture of a system often drift apart during the system's evolution [Jan+07] (also known as *architectural drift*).

To address the problems of architectural erosion and architectural drift, many approaches have been proposed [Men+02; Mur+95b; Egy04]. In Chapter 5 we propose a semi-automatic approach for keeping the architecture and source code consistent throughout the software evolution. In this approach, we use a Domain Specific Language (DSL) that allows architects to specify architectural abstraction specifications. These architecture abstraction specifications enable the architects to define architectural components based on the source code. Based on the architecture abstraction specifications we then automatically generate an architectural component view of the system and its current state.

While the approach from Chapter 5 addresses the consistency of architecture and source code, it does not offer any solutions to prevent the architectural component models from degrading over time and become less and less understandable. For instance, some architecture design models tend to grow in size over time, as new features are added to the system, until they become at some stage hard to understand.

Clements et al. [Cle+02] stated that it is essential that an architecture is documented well in order to communicate it. Reduced understandability hampers the possibility to communicate the architecture well and thus probably leads to further architectural erosion and drift. This is why we consider the understandability of an architecture as essential to the future evolution of a software system.

In this chapter we propose to integrate our DSL-based architecture evolution approach from Chapter 5 with empirically evaluated understandability metrics. We suggest to use understandability metrics for the architectural component view as a whole as well as understandability metrics that focus on single architectural components. This way, while using our Architecture Abstraction Specification Language to create architectural component views, the architect is automatically informed when the understandability of the architecture in the component models that are created through architecture abstraction specifications is reduced during the evolution of the software system and can take measures to improve the architecture's understandability. The metrics we use are empirically evaluated by Stevanetic et al. [Ste+14a; SZ14] with regard to the understandability of either the whole component view or the individual components. A precondition for the application of the metrics (i.e., for the accurate and successful metrics calculations) is an "up-to-date" component view that reflects the source code of the examined system. The main contributions of this chapter are the conceptual integration of the two approaches and the integration into our DSL-based tool support.

The remainder of this chapter is organized as follows: We give an overview of the proposed integrated approach in Section 7.2. Section 7.3 describes the details of the given integrated approach. We present a case study in which we have studied the applicability of our approach in Section 7.4. We conclude this chapter in Section 7.5.

7.2 Integrated Approach Overview

The approach that we present in this chapter represents an extension of the previously explained approach for supporting semi-automated architectural abstractions of a software system from the source code using a DSL that we call *Architecture Abstraction Specification Language*. The proposed extension of the approach is related to the integration of software metrics that can support the understandability of architectural component views generated using the previously explained approach. The understandability related software metrics are empirically evaluated by Stevanetic et al. [Ste+14a; SZ14] and can further support the maintainability of the continuously evolved architecture.

Namely, they did a series of studies where they tried to empirically evaluate and prove the usefulness of software metrics in assessing the understandability of architectural component views. Their goal was to produce a set of guidelines as best practices for architectural component view design. The metrics that are shown, are collected at the level of individual components [SZ14] as well as at the level of the whole architecture [Ste+14a]. They include three size metrics related to the number of components, the number of connectors and the total number of elements (summing up the number of components and the number of connectors) in the architecture and four metrics related to individual components: the number of classes in a component, the number of incoming

dependencies of a component, the number of outgoing dependencies of a component, and the number of internal dependencies of a component.

Regarding the three architecture level size metrics, Stevanetic et al. showed that middle values of those metrics significantly increase the architectural understandability compared to high or low values [Ste+14a]. The indicated thresholds/guidelines for using the metrics are roughly predicted and need to be investigated further (they are defined below in Section 7.3). More precisely they showed that the component diagrams (visual representations of the component views) with very high numbers of elements usually suffer from mixing of several concerns which might lead to ambiguity and less precision [Ste+14a]. Very low numbers of components, links, and elements are not sufficient to model all relevant concerns of the architecture [Ste+14a]. The four metrics at the level of individual components are shown to be useful in predicting the effort required to understand an individual component, measured through the time that participants spent on studying a component [SZ14]. They have shown either a statistically significant correlation with the effort required to understand a component or can be used in the prediction models obtained using the multivariate regression analysis, to predict the given effort.

The integration of the given metrics in the workflow of the Architecture Abstraction Specification Language is shown in Figure 7.1. In order to more easily distinguish the part related to the integration of the given metrics we marked it red in the figure. Firstly, the metrics calculations are extracted from both the class model and the component view. The obtained metrics values then need to be evaluated with regard to different metrics constraints, i.e., it is checked if the calculated metrics values satisfy required metrics constraints. Metrics constraints represent a set of rules defined on metrics values that need to be satisfied. In our case, they are defined based on the aforementioned empirical evaluations and also take into account some additional reasonable considerations. Namely, for the architectural level metrics the obtained middle values that increase the architectural understandability can be realized as constraints (thresholds/guidelines are shown in Section 7.3). For the metrics at the level of individual components we did not examine any specific values/thresholds that can be specified as constraints but the information related to the obtained prediction models and the statistically significant correlations can be useful in providing the relative values that might be used for identifying critical components which require more effort to be understood (see Section 7.3 for more details). All given constraints and considerations can be further refined with regard to the architects' and developers' specific experience and more specific requirements in the certain domain. In case that some metrics values do not satisfy the corresponding constraints the architectural abstraction specification or the source code have to be improved in order to resolve the inconsistencies that occurred.

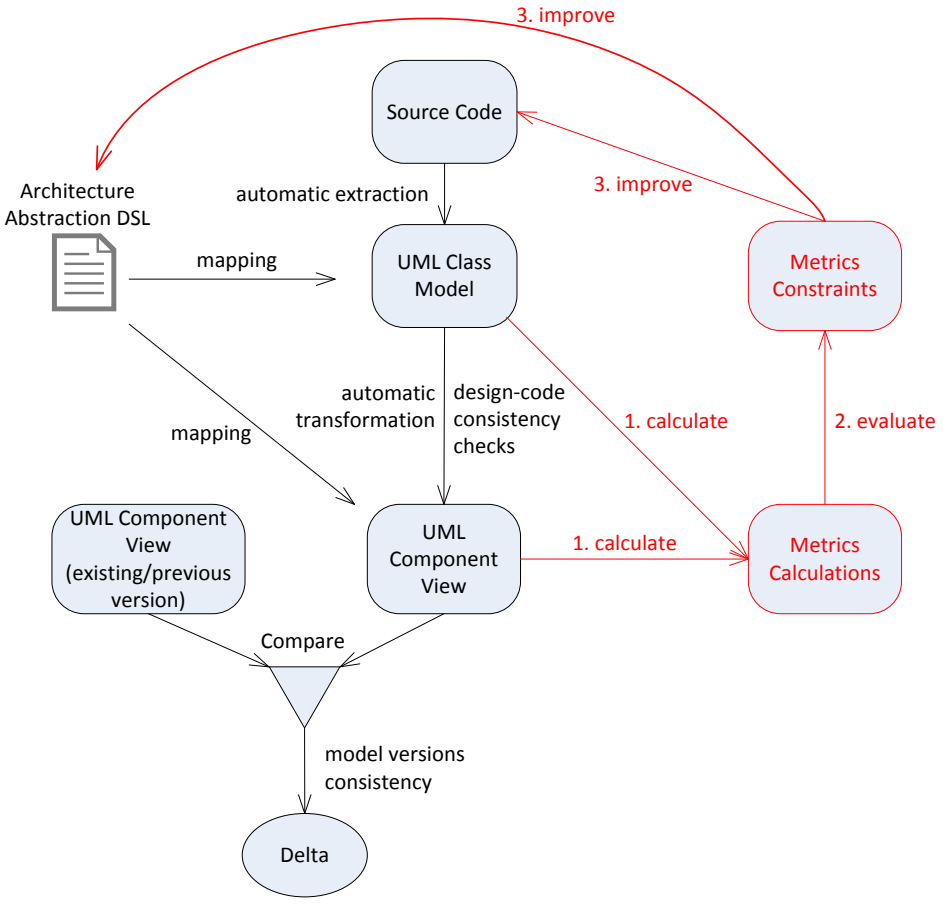

FIGURE 7.1: Integration of the understandability related metrics in the DSL-based architecture abstraction approach

7.3 Integrated Approach Details

In this section, we explain the technical details of our approach. In Section 7.3.1 we discuss the metrics we use in our approach and in Section 7.3.2 we present the details about the DSL-based architecture abstraction approach and its integration with the given metrics.

7.3.1 Understandability Related Metrics

Regarding the empirical studies for supporting the understandability of architectural component views utilize the results from three studies by Stevanetic et al. [Ste+14a]. The first two studies examine to which extent the software architecture could be conveyed through architectural component views (16 different component diagrams were studied) and they are based on the participants' subjective ratings while the third one examines the relationships between the effort required to understand an individual component, measured through the time that participants spent on studying a component, and some component level metrics that describe component's size, complexity and coupling.

As Stevanetic et al. [Ste+14a] state, the statistical evaluation of the results from the first two studies shows that metrics such as the number of components, number of connectors, number of elements, and number of symbols used in the diagrams can significantly decrease architectural understandability when they are above and below a certain, roughly predicted threshold. Also, their results indicate that architectural understandability is linearly correlated with the perceived precision and general understandability of the diagrams (please refer to [Ste+14a] for more details about the terms precision, general understandability, and architectural understandability). The conclusions form these two studies are summarized below [Ste+14a]:

- Any measures that increase the general understandability and precision of architectural component views directly help to improve the architectural understandability.
- Measures to increase the domain knowledge are helpful to increase the understanding of architectural component views in general.
- From a certain size on (in terms of number of elements), architectural component views get hard to understand in general because of the high cognitive load and human perception limits.
- Middle values of the number of components, links, elements, and symbols in the diagram significantly increase the architectural understandability compared to high or low values. The diagrams with very high numbers of elements usually suffer from mixing of several concerns which might lead to ambiguity and less precision. Very low numbers of components, links, and elements are not sufficient to model all relevant concerns of the architecture. These dependencies might also deserve to be investigated further, especially it would be interesting to indicate the thresholds of maximum (minimum) numbers of components, links, elements, and symbols that should be depicted in one diagram more precisely. So far, we consider the thresholds we found as rough indicators.

From these 2 studies we consider three metrics: the number of components, the number of connectors and the total number of elements (summing up the number of components and the number of connectors) in the architecture. As mentioned above, they observed that the middle values of those metrics significantly increase the understandability of the architecture. Therefore the corresponding metrics' constraints can be realized (based on the thresholds that are roughly indicated in our previous study [Ste+14a]). Table 7.1 summarizes the considered architecture level metrics together with the corresponding constraints. The number of symbols is not considered because it is related to the visual representation of the component views that we do not support at the moment. Also we do not consider the first two items in the above mentioned conclusions because we did not examine the appropriate measures for it. Those items are related to the measures of the precision, the general understandability, and the domain knowledge contained in the component views. Some aspects of these measures are automatically taken into account when the Architecture

Abstraction Specification Language is specified like for example the names of the components. Informative and coherent names can increase the precision and convey the domain semantics of the system. However, more studies are necessary to define and examine the appropriate measures and the corresponding constraints for these aspects.

Regarding the third study four metrics related to individual components the number of classes in a component, the number of incoming dependencies of a component, the number of outgoing dependencies of a component, and the number of internal dependencies of a component are considered. Stevanetic et al. [Ste+14a] state that the results of their analysis show a statistically significant correlation between three of the metrics, number of classes, number of incoming dependencies, and number of internal dependencies, on one side, and the effort required to understand a component, on the other side. In a multivariate regression analysis they obtained 3 reasonably well-fitting models that can be used to estimate the effort required to understand a component.

For the metrics at the level of individual components we did not examine any specific values/thresholds that can be specified as constraints. The information related to the obtained correlations and prediction models can be used to provide more relative values (rather than evaluating a design by giving absolute values) that might be used for identifying critical components which require more effort to be understood. Those components can be further simplified and/or reorganized together with other components in the system to satisfy the given understandability requirements. For example, Bouwers et al. found that the components should be balanced in size in order to facilitate the system's analyzability (location of possible failures/bugs in the system) [Bou+11]. In our case the similar reasoning can be applied. Balanced values for the components' understandability effort can facilitate the analyzability of the whole system in terms that all components require the same effort to be understood which can facilitate the location of possible bugs/failures in the system (see Section 7.4 for an illustrative example). Furthermore for the component level metrics the architects/developers can adopt the specific ranges for them based on their concrete experiences and requirements.

The component level metrics together with the prediction models and the identified correlations to the measured understandability effort are shown in Table 7.2. The Spearman's correlation coefficients are shown. They are widely used for measuring the degree of relationship between two variables and take a value between -1 and +1. A positive correlation is one in which the variables increase (or decrease) together. A negative correlation is one in which one variable increases as the other variable decreases. The coefficient for the number of outgoing dependencies metric is not shown because it is not statistically significant.

Metric	Description	Metric's constraint
Number of Components (**NCOM**)	Total number of components in the architecture	5 < **NCOM** < 15
Number of Connectors (**NCONN**)	Total number of connectors in the architecture (regardless whether the connector is one-way or two-ways)	3 < **NCONN** ≤ 17
Number of Elements (**NELEM**)	Total number of elements in the architecture (summing up the number of components and the number of connectors)	11 < **NELEM** ≤ 25

TABLE 7.1: Architecture level metrics [Ste+14a]

Metric	Description	Spearman's correlation coefficient
Number of Classes (**NC**)	Total number of classes inside a component	r=0.74
Number of Incoming Dependencies (**NID**)	Total number of dependencies between the classes outside of a component and the classes inside a component that are used by those outside classes	r=0.26
Number of Outgoing Dependencies (**NOD**)	Total number of dependencies between the classes inside a component and the classes outside of a component that are used by those inside classes	-
Number of Internal Dependencies (**NIntD**)	Total number of dependencies between the classes within a component	r=0.66

Prediction Models
Model 1:~4.85+1.52*__NC__-0.53*__NID__
Model 2:~4.58+1.46*__NC__-0.52*__NID__+0.12*__NOD__
Model 3:~5.32+1.42*__NC__-0.58*__NID__

TABLE 7.2: Component level metrics and the obtained prediction models [Ste+14a]

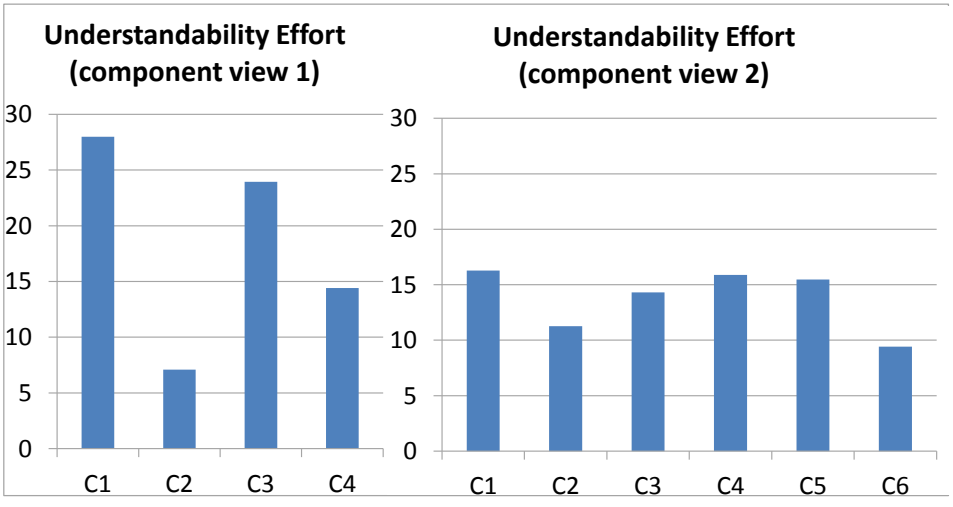

FIGURE 7.2: Understandability effort for both component views

7.3.2 Architecture Abstraction Approach and Metrics Integration

Our consistency checks that were mentioned above and are discussed in detail in Chapter 5 supports the evolution of the software system in such a way that it enables consistency checking between different versions of software and also between different artifacts of the same software version (for example between the component view and the corresponding class view). The integrated empirically evaluated metrics provide an additional consistency checking possibility. Namely, according to the discussion above the integrated metrics can provide a valuable support in assessing the understandability of architectural component views which plays a key role in managing and maintaining the overall system. Different versions of the software can be compared using the given metrics set that can be used to argue about the understandability level of both the architectures and the individual components contained in them. Based on the obtained values critical points can be recognized, for example the components that have significantly increased the effort to be understood can be identified. Also different architecture abstractions can be compared in order to generate the one with the reasonable understandability level. The integrated metrics benefit from the architecture abstraction tool in the way that the later provides an "up-to-date" architectural component view that reflects the source code (i.e. all source code classes are mapped to their respective components) that is necessary for the metrics calculations. This way, the architects/developers can gradually improve the architecture by making the changes in the source code or in the Architecture Abstraction Specification Language and judge the understandability of the architecture created with the DSL. The metrics calculations are integrated using the Xtext validation framework which triggers their execution/recalculation whenever the source code or the architecture component view is changed. The corresponding warnings are reported whenever the metrics values violate the respective set of metrics constraints.

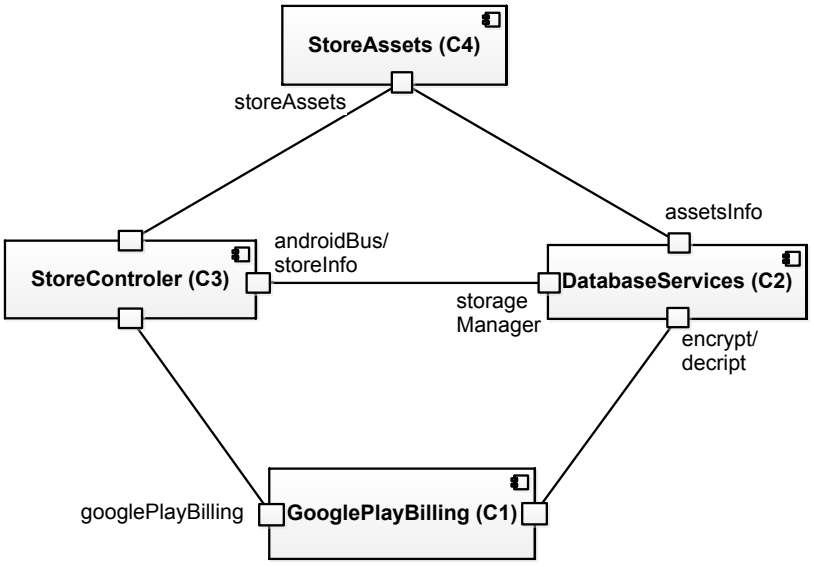

FIGURE 7.3: Soomla Android store component view 1

7.4 Case Study

In this section we present a small case study that illustrates how the previously explained approach can be used to localize possible undesirable effects in the design, in this case the observed fluctuations in the understandability effort of architectural components. The studied system is the Soomla Android store Version 2.0[1], an open source framework for supporting virtual economy in mobile games. Namely, we show two architectural component views of the system that differ in the number of components and the number of classes that the components contain. In both cases we calculate the understandability effort required to understand each component based on the provided prediction models. In the first case, the understandability effort is unevenly distributed over the components, i.e., some components require very low, while some others require very high effort to be understood (see Figure 7.2). After studying the first component view the new component view is generated that better distributes the understandability effort over the components. Thanks to the architecture abstraction tool all source code classes are mapped to their respective components which is a precondition for the accurate and successful metrics calculations. Furthermore the second component view is easily created from an architectural abstraction of the first one by simply relocating the classes in the DSL code from one component to the other. This step, of course, requires human expertize and manual effort. However, please note that the migration to the new view can be done incrementally, by performing small changes in the DSL and observing the change of the metrics with each change in the DSL. In general a large, inherently complex system will have lower understandability because the identified metrics (i.e. NCOM etc.) will be higher, than a small, simple system, regardless of the quality of the architecture abstractions used. In that case the aim of the approach is to adapt the inherently high complexity to the extent that is acceptable using the explained incremental changes.

Figure 7.3 shows the first component view obtained by studying the given software system. The visualization of both component views is separately created in the form of a UML component diagram. Figure 7.4 shows the second component view created to support better distributed understandability effort between the components in order to facilitate their analyzability (see Section 7.3 for more details). The understandability efforts are shown in Figure 7.2.

From Figure 7.2 we see that the Components C1 and C3 of the first component view require a pretty high effort to be understood while the Component C2 requires much less effort. In the second component view the components require more or less balanced effort to be understood, and it is lower than the effort required for the Components C1 and C3 in the first design. This small case illustrates how the given metrics provide a useful feedback in the explained context.

[1]see: http://project.soom.la/

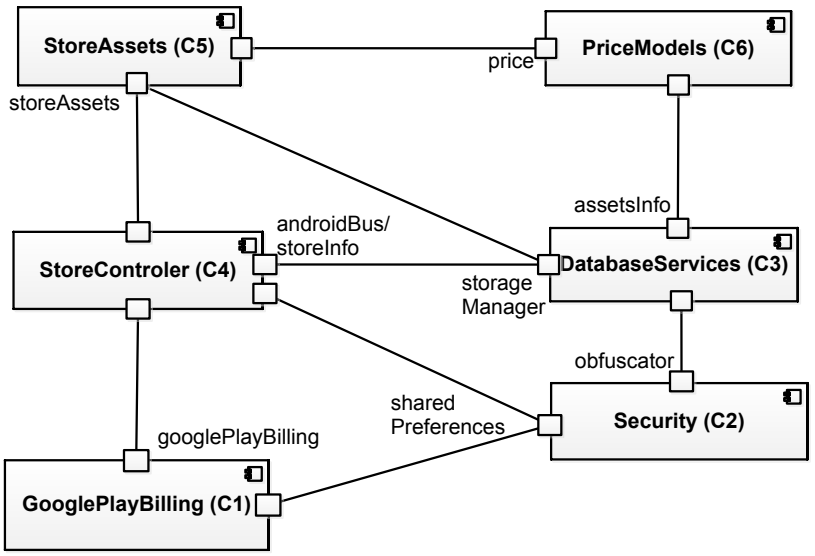

FIGURE 7.4: Soomla Android store component view 2

7.5 Conclusions and Future Work

In this chapter we presented an approach that uses empirically evaluated understandability metrics to support the software architect during architecture documentation and evolution. It is built on architectural component views that are generated from architecture abstraction specifications. We automatically calculate a number of different metrics whenever the architecture abstraction specification or the source code are updated. If the metrics exceed defined thresholds the prototype notifies the software architect of the potential understandability problem who should revise the architecture abstraction specification and the source code to improve the understandability of the architectural component view. The improved understandability then eases the future evolution of the systems as it reduces the risks of misunderstandings and thus the risk of changes that affect the quality of the architecture in a negative way. The main contributions of this chapter lie in the integration of the two approaches and proposing a set of metrics-based guidelines for component model design that are derived from our previous empirical studies.

The approach presented in this chapter aids in answering Research Question 4, as it integrates the means to automatically check the understandability of an architecture document with the approach we introduced in Chapter 5.

A limitation of our approach is that we currently consider only understandability metrics as a measure for quality. In our future work we plan to integrate other quality metrics that can be used to prevent architecture erosion and drift.

Part III

Consistency Managment During Software Evolution

8 Reconciling Software Architecture and Source Code in Support of Software Evolution

8.1 Introduction

In this chapter, we extend on the ideas presented in Chapters 5 and 6, as we integrate our approaches for documenting architectural components and architectural patterns with the concept of documented architectural decisions and evolution styles [Bar+12] to improve the support for evolving software architecture and source code in a consistent manner. As Cuesta el al. state [Cue+13], SA is an artifact for the evolution because it can be used as a shared mental model that guides the planning and restructuring of the software [Hol02], but it is also an artifact of the evolution, because it must be evolved itself [Bar+08]. Several approaches have been proposed that use SA as an artifact of and for the evolution, including different proposals following the Evolution Styles approach [Bar+12; Le +08; NT10; Tam+06]. Authors claim that the evolution from an initial to a target architecture should be carried out by planning and analyzing different Evolution Paths, so that the risks and problems during the evolution can be avoided or at least mitigated.

Despite the undoubted advantages that such an approach has, without tight integration with the software development activities it is unlikely to be put into practice. As Ozkaya et al. [Ozk+07] found out during their interview study, developers in practice do not use architecture-centered practices to manage evolution decisions. They just focus on the coding efforts when evolution needs arise, so that finally architects must cope with architectural erosion or architectural drift [PW92b]. Better support for developers for integrating their work with the SA is needed to avoid architectural erosion and architectural drift. This is also pointed out in a recent study of 705 official releases of nine open-source software projects by Neamtiu et al. [Nea+13]. Amongst other results, Neamtiu et al. reported that developers should not only focus on the implementation but also on carrying out proactive actions, such as reverse engineering, to avoid the development of code that is difficult to evolve.

The fact that developers only pay attention to code is especially common in Open Source (OS) projects where the attention on planning or modeling is often even non-existent. An example in this sense is Moodle, one of the most widely used platforms for e-learning. It has a wide community of

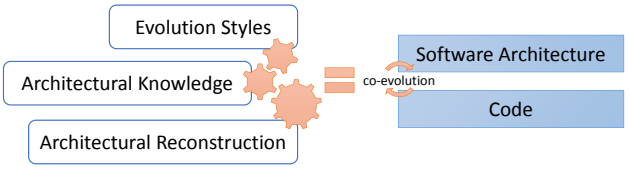

FIGURE 8.1: Main foundations of this chapter

developers, as well as a properly described development process. Moreover, it also has had major changes along its evolution process [MR13] as well as large numbers of issues filed for specific versions [Tra]. However, there is not a clear description of the architecture, the planned evolution, its traces to code, and so on. That is, if we have a look, for example, on Moodle's roadmap for the latest Version 2.7 [Moo], we find that it is described just textually in terms of new components to be developed (e.g. a new events system or a new framework for logging and reporting) or components to be changed (e.g. changes to bootstrap themes). This lack of architectural guidance can lead to a misalignment between the architecture as planned by (some of) its developers and the concrete architecture (implementation) really coded by the developers. But even when the architecture is clearly described, this misalignment between code and software architecture can also happen. For instance, Nakagawa et al. [Nak+08] present a case study that clearly shows that the severe differences between the conceptual architecture of a system and its concrete architecture were accidently or inevitably inserted as the code evolved. Moreover, as these authors claim, these differences had a very negative impact on the system evolution affecting important quality attributes, such as functionality, maintainability, and usability.

All the previous issues, led us to the research question (RQ 5) already mentioned in Chapter 2: How to reconcile the different points of view that software architects and developers have when the software is being evolved and how to enforce the integration of software architecture and source code? This chapter claims that a positive answer to this question can be provided by integrating three different approaches (see Figure 8.1), namely Evolution Styles, Architectural Knowledge and Architectural Reconstruction. In particular, we suggest using evolution styles to guide the stepwise architecture evolution in a number of incremental evolution steps. In each evolution step, we suggest using an architecture reconstruction tool to enforce the evolution of architecture and source code keeping both in sync. The reconstruction tool uses an architecture abstraction specification to generate a component view from the implemented source code. We suggest realizing each evolution step by (a) transforming the architectural abstraction specification for the architectural reconstruction to contain the architectural changes and (b) developing the source code accordingly. After both, the evolution of the architecture and the code have happened, we automatically reconstruct the architecture using our reconstruction tool. The tool now can detect inconsistencies between the architecture specification and the source code. Such inconsistencies are the result of violations in the architecture, the transformation or the source code. These artifacts are incrementally refined

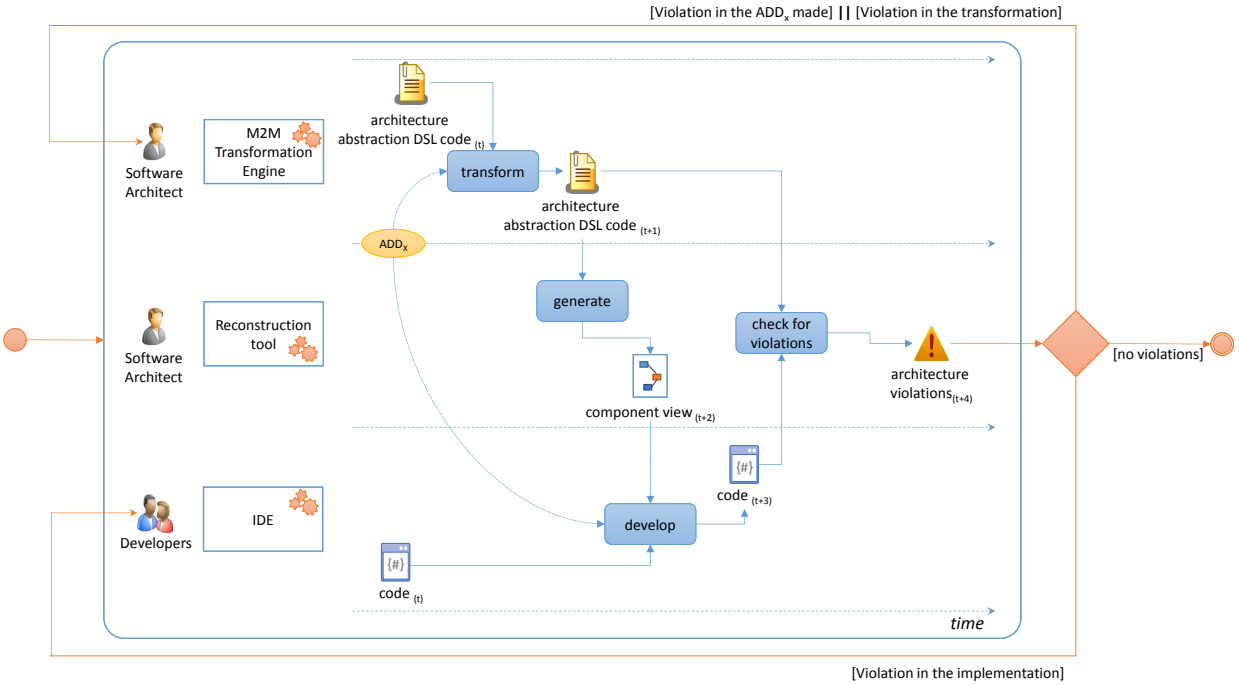

FIGURE 8.2: Carrying out an *evolution step*

until no more violations occur. The result is that the evolution step is carried out and that the architecture and the source code are in sync.

This chapter is structured as follows. After this introduction, Section 3.3 describes the main foundations of this approach, as well as analyses the related work. Section 8.2 provides an overview of our approach and Section 8.3 describes the technical details of the tool support of our approach. Then, four case studies are described in Section 8.4 to illustrate how our work can be put into practice. Finally, the conclusions drawn as well as our future work are presented in Section 8.5.

8.2 Our approach: Code and Software Architecture Evolution

As described in Section 8.1, our approach relies on three previous proposals in the research field. Mainly, the approach uses Evolution Styles [Bar+12] to help the architect to plan and execute the evolution from an initial architecture to a target architecture following one of the different evolution paths available. It also takes into account AKdES [Cue+13] so that each evolution step is traced from a specific ADD that describes how it must be carried out, keeping the history of the evolution process. However, it differs from both of them in how these evolution steps are carried out. Specifically, it extends the definition of both proposals to provide software architects and developers with support for the evolution of both artifacts.

As Figure 8.2 illustrates, in our approach, an evolution step does not focus only on the architectural transformation or the architectural decision made but also on the code as well. In particular, our

approach considers an evolution step as an iterative process that entails several tasks, artifacts and tools. The assumption of our approach is that the component view of the architecture is not manually created, but that the Architecture Reconstruction Tool (see Section 8.3) is used to automatically create the component view based on the architecture specification made in the Architecture Abstraction Specification Language and on the source code.

Provided that we are in an instant t of the evolution, when we have made a description of the architecture using Architecture Abstraction Specification Language, an Architecture Design Decision (ADDX) to be applied in the next evolution step, and a snapshot of the code to be evolved, the following tasks should be carried out:

1. First, the software architect evolves the architecture into a new version in order to apply the ADDX made. Following the ideas presented in [Bar12; Lyt+13b; NC08], this evolution is carried out by applying Model-To-Model (M2M) transformations, that is, every ADD is mapped to an architectural transformation. This means that the evolution graph is really described as a graph of architectural transformations. This helps to improve the maintainability of the evolution graph because it can be re-generated/updated in an automatic way whenever it is necessary.

2. Once the architecture has been evolved, the software architect generates, using the tool that we created for the approach in Chapter 5 (for the remainder of this chapter we refer to this as Reconstruction Tool), a UML component view from the software architecture specification. Developers are provided with this component view because, as discussed in Chapter 4, it helps to improve the design understandability.

3. Then, the developers develop a new version of the code using both the component view as well as the ADDX that also describes how the evolution step should be carried out, from the code point of view. In other words, the specification in the transformed Architecture Abstraction Specification Language can be seen as coding guidelines for the changes that need to be made to the source code in order to realize the ADDX.

4. Once, the code evolution has been finished, the software architect uses the Reconstruction Tool to check whether any misalignment between the code and the planned architecture exists.

The software architect analyzes the results of the Reconstruction Tool and determines whether any one of the following violations has happened:

- A violation in the implementation. Developers failed to apply the required ADDX or the coding guidelines provided in the Architecture Abstraction Specification Language properly.

Thanks to the facilities provided by the Reconstruction Tool, the software architect knows which areas of the code fail in the implementation step, and provide developers with the necessary warnings. Then, a new iteration would be carried out by the developers in order to solve the detected problems in the implementation.

- A violation in the ADDX was made. When the software architect evaluates the violation, he can also conclude that it was a problem in the decision made as it cannot be properly traced to the code. Therefore, a new decision must be made (ADDY) that guides the developers properly and considers the knowledge gained during the implementation of ADDX. The previous decision ADDX should be marked as inhibited, establishing the rationale behind that decision. A new iteration of the evolution step would then be carried out by software architects and developers to apply the new decision made.

- A violation in the architecture abstraction specification. Finally, the software architect can conclude that the architecture abstraction specification created as result of applying the ADDX was not appropriate and a correction of the coding guidelines provided through the Architecture Abstraction Specification Language is needed in order to properly align ADDX and the corresponding source code.

If no violations are detected during the checking task, the evolution step was successful and a new version of both the source code and a corresponding architecture abstraction specification is available for realizing the next evolution step of the evolution path.

As can be observed, both software architects and developers can reconcile the different views they have of the software evolution, software architecture and code, respectively. Thanks to the introduction of the Evolution Styles, software architects can plan and analyze the evolution of the software architecture. Then, by using the AK they can convey the idea behind each evolution step to the developers and also keep the history of all the decisions they made. Finally, thanks to the Architectural Reconstruction approach, both software architects and developers are notified if any kind of problems exists during the on-going evolution step. As a result of the integration of the three approaches, Evolution Styles, AK and Architectural Reconstruction, both software architects and developers are closely cooperating in the process of evolving code and software architecture. In the following section, the approach details and the tool support of the approach are described. Next, in Section 8.4, several case studies are used to present how the approach can be put into practice.

8.3 Approach Details

In this section we describe the technical aspects of our approach and especially the tool support for our approach in more detail. Our approach assumes that ADDs have been made and documented before an evolution step happens. Any ADD documentation tool can be used for this task. In our work, we use the ADvISE[1] tool for this task. We now utilize a variant of the Architecture Abstraction Specifiacation Language discussed in Chapter 5 to describe the architecture specification.

During the evolution of a system, after an architecture has been specified, every ADD that is made, requires us to change the architecture specification. We specify changes to the architecture abstraction specification in terms of QVT-o transformations[2]. QVT-o allows us to define transformations for any form of EMF Models. As our architecture specification is implemented as an Xtext DSL, no setup for QVT-o was required. We implemented QVT-o transformations for the following changes[3] to the architecture specification:

- addComponent: Creation of a new architectural component. The implementation is shown in Section 8.4.1 in Figure 8.3.
- deleteComponent: Removal of an architectural component. Its source code is shown in Appendix D in Figure D.2.
- addConnector: Creation of a new connector between two architectural components. The implementation is shown in Appendix A in Figure D.3.
- deleteConnector: Removal of a connector between two architectural components. Figure D.4 and Figure D.5 in Appendix D show the QVT-o code for this transformation.
- updateAbstractionSpecification: This transformation is used to replace the existing architecture abstraction specification of a component with a new specification. The transformation's implementation is shown in Appendix D in Figure D.1.

As an example for these transformations (see Appendix D for their specification) we show the transformation for adding a new component to the architecture specification in Figure 8.3. The addComponent transformation takes two input models: one is the existing architecture specification and the second model contains the component to be added. This transformation then generates the modified architecture specification as an output model. These transformation enable us to make all the necessary changes to the architecture, so that each evolution step is retraceable.

[1] https://swa.univie.ac.at/Architectural_Design_Decision_Support_Framework_%28ADvISE%29
[2] https://www.eclipse.org/mmt/?project=qvto
[3] As the Architecture Abstraction Specification Language currently does not support the definition of ports, no transformations for creating or deleting ports in the architecture specification have been defined at the moment.

```
modeltype DSL uses 'http://www.univie.ac.at/cs/swa/component/
architectureabstraction/ArchitectureAbstractionDSL';

transformation addComponentTransformation
      (in componentview:DSL, in newComp:DSL , out output:DSL);
main() {
componentview.rootObjects()[DSL::Transformation]-> map
      addComponent(newComp.rootObjects()[DSL::Transformation]
      ->asOrderedSet()->first().map getComponent());
}

mapping DSL::Transformation::addComponent(in inputComp:DSL::ComponentDef):
DSL::Transformation {
      result.name:=self.name;
      result.components:=self.components->including(inputComp);
}
mapping DSL::Transformation::getComponent () : ComponentDef {
init  {
      result := self.components->asOrderedSet()->first();
      }
}
```

FIGURE 8.3: QVT-o transformation for adding a component to the architecture specification

Once the architecture specification has been changed, the Reconstruction Tool, which is built around our Architecture Abstraction Specification Language, transforms our architecture specification into a UML component view (using Xtend[4] for generating the model). In addition the Reconstruction Tool performs a number of consistency checks on the architecture specification and source code and any resulting issues are listed in a consistency report. This report is implemented as warnings and errors in the user interface for our Architecture Abstraction Specification Language. For example, among many other verifications, it checks for connectors between components that exist in the source code but do not exist in the architecture specification and parts of the architecture specification that do not relate to any source code artifacts. Whenever the source code changes, the checking can automatically be executed and a new consistency report is generated. This supports the software developer during and the software architect after the implementation step of our approach when they need to confirm whether the evolution of architecture and code was successful or whether inconsistencies between architecture specification and code still exist. An example for a consistency report is shown below in Section 8.4.1, Figure 8.7.

8.4 Case Studies

In this section we illustrate our approach in four case studies. Two of them are related to the open source project Apache CXF[5] which is a framework for implementing Web services. It is implemented in Java and is built around a central Interceptor Chain which is used to handle incoming and outgoing messages on the client- and on the server-side. In our first case study, described in Section 8.4.1, we used our approach to retrace the changes from Apache CXF Version

[4] http://www.eclipse.org/xtend/
[5] https://cxf.apache.org/

2.6 to Version 2.7. In the second case study, illustrated in Section 8.4.2, we retraced the changes from Apache CXF Version 2.7 to Version 3.0. In the two other case studies we applied our approach in the architectural evolution of Soomla. The Soomla[6] framework is an open-source framework for in-app purchases in Android. In Case Study 3, described in Section 8.4.3 we retraced the architectural changes between version 3.2 and version 3.3 using our approach. Finally, in the last case study, which we describe in Section 8.4.4, we used our approach to add support for a new payment method to the Soomla framework.

8.4.1 Case Study 1: Evolving From Apache CXF 2.6 to Apache CXF 2.7

We used our approach to retrace the changes from Apache CXF Version 2.6 to Version 2.7. In total this version change raised the number of lines of Java source code from about 480.000 to more than 513.000 and consisted of 867 modified classes, 121 new classes, and 7 removed classes. For this case study, we created three architectural component views of the CXF architecture using our Architecture Abstraction Specification Language. These views consist of a high-level overview of the complete architecture as well as two detailed views, one for the architecture of CXF transports and one for the architecture of CXF front-ends. While a CXF transport implements a specific protocol that is used to send messages, CXF front-ends define and implement the different types of supported services like e.g. JAX-WS, JAX-RS.

In Figure 8.4 we show a detailed view for the architecture of Apache CXF transports (showing only a subset of all supported transports). In Apache CXF, transports are used to abstract from the protocols used to send messages from the client to the server and back. Each transport has to provide implementations for the following three concepts: Conduits represent a channel for sending a message, Destinations represent the location of a service (a ServiceEndpoint), and TransportFactories are used to obtain transports for specific URLs.

As we studied the evolution of Apache CXF from Version 2.6 to Version 2.7 we discovered a number of architectural changes to the different views which are summarized in Table 8.1. In order to illustrate our approach in detail, we picked the change with ID 2, that is, the architectural changes necessary to add support for the UDP protocol. The implementation of a new protocol might be necessary when using Apache CXF in a restricted environment or when a new protocol should be supported. In Apache CXF this was the case for Version 2.7 where a new UDP transport was implemented.

In this case, we study the architectural evolution of the given architecture when the ADD to support a new protocol has been taken. As we focus on studying the architectural evolution, we first created an architecture abstraction specification for Apache CXF Version 2.6 which we used as the basis for our approach (the architecture specification at t in Figure 8.2). Our approach

[6] http://soom.la/

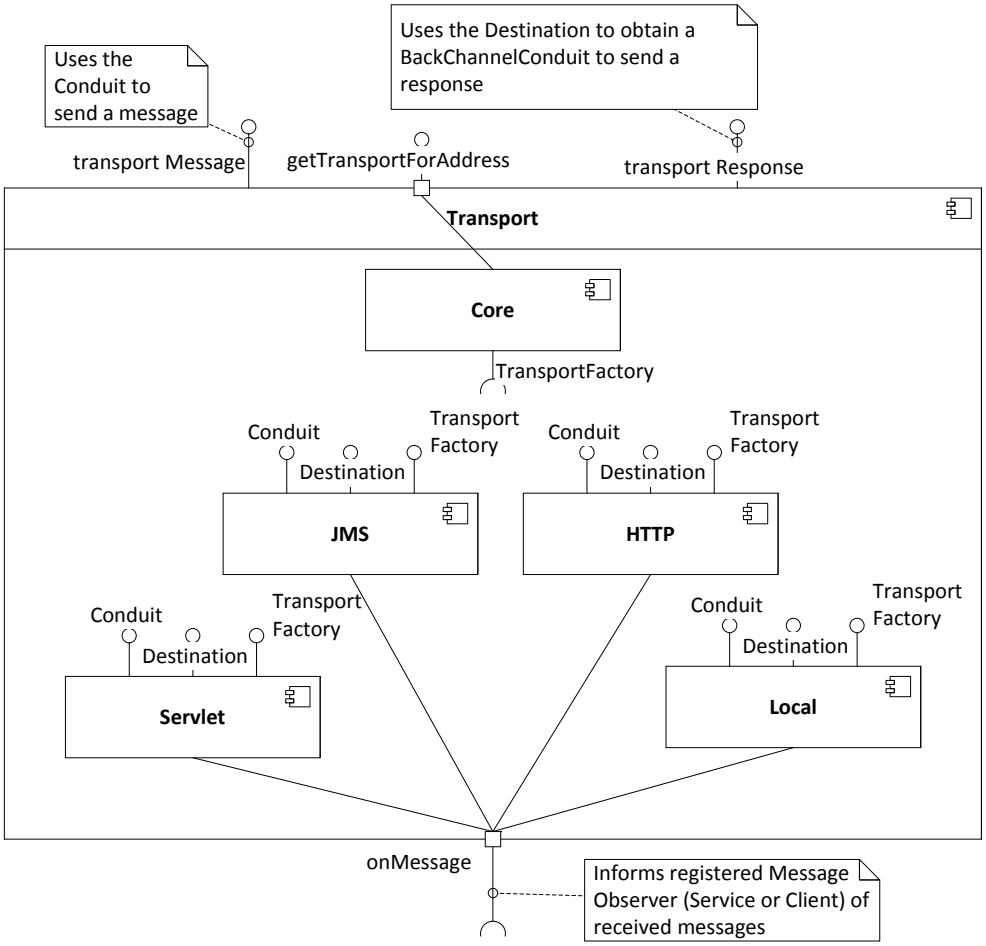

FIGURE 8.4: A simplified view of the architecture of Apache CXF transports

ID	Architectural change	Affected view	Arch. transformations
1	Added support for service discovery	architecture overview	addComponent addConnector
2	Added support for sending messages using UDP protocol	transport view	addComponent addConnector
3	Added support for sending SOAP messages using the UDP protocol	transport view	addComponent addConnector
4	Added support for asynchronous messages over HTTP	transport view	addComponent addConnector (used twice)
5	Partial support for JAX-RS v2.0 (support for JAX RS v1.1 already existed)	frontend view	updateAbstraction Specification

TABLE 8.1: Overview of all architectural changes from Apache CXF Version 2.6 to Apache CXF Version 2.7

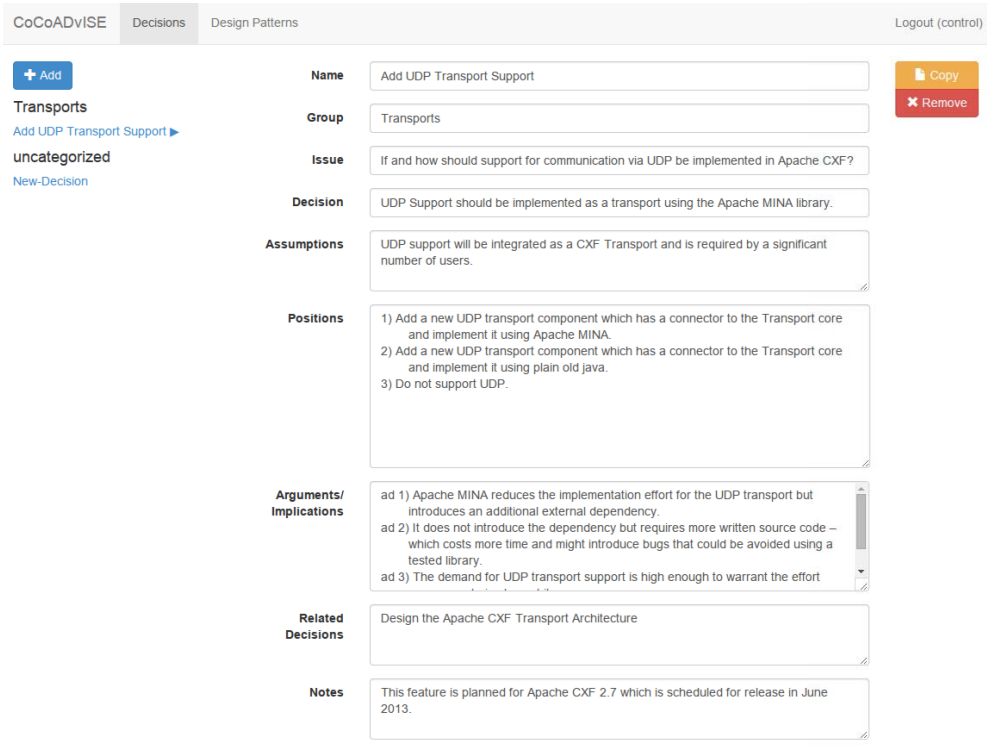

FIGURE 8.5: Documented architectural decision to implement UDP transport support for Apache CXF using the online version CoCoAdvise of the Advise Tool

assumes that this documented architecture already exists. If no initial architecture documentation exists, the semi-automatic architecture reconstruction approach (summarized in Section 8.3 and described in detail in Chapter 5) or other existing architectural reconstruction approaches can be used to obtain an architectural description of a system in its current state. In the following, it is described how an iteration over the evolution step presented in Section 8.2 was carried out:

- Transform step: After the ADD to implement this new protocol was documented (see in Figure 8.5), we executed the transform-step of our approach. In our example, we first used the addComponent transformation to add a component named UDP to the architecture specification for the CXF transports. We exemplarily show the QVT-o transformation for adding a component in Figure 8.3. All other QVT-o transformations used throughout the case studies can be found in Appendix D. The added component with its architecture abstraction specification is shown in Figure 8.6. We use Eclipse Launch configurations in conjunction with the QVT-o transformations to record all the architectural changes carried out during a transformation step (see Appendix D for an example of the launch configuration for adding the new UDP component). We defined this architecture abstraction specification in the following way: using the Architecture Abstraction DSL, we first specified that source code for this component should be part of the Package org.apache.cxf.transport.udp (specified by the UDP component's first Package rule in Figure 8.6) and second, we defined that this package and, thus, the component needs to contain an implementation of the interfaces

```
Component Core
        consists of
        {
                Package (org.apache.cxf.transport, excludeChildren)
                or
                Package (org.apache.cxf.transport.common)
        }

Component UDP
        consists of
        Package (org.apache.cxf.transport.udp)
        or
        {
                Package (org.apache.cxf.transport.udp)
                and
                {
                        InstanceOf (org.apache.cxf.transport.ConduitInitiator)
                        or
                        InstanceOf (org.apache.cxf.transport.Conduit)
                        or
                        InstanceOf (org.apache.cxf.transport.Destination)
                }
        }

//[…]
```

FIGURE 8.6: Excerpt of the architecture specification showing the Core component of the transport view as well as the new architectural component that was added during the first transformation step of our case study with syntax highlighting for reported inconsistencies

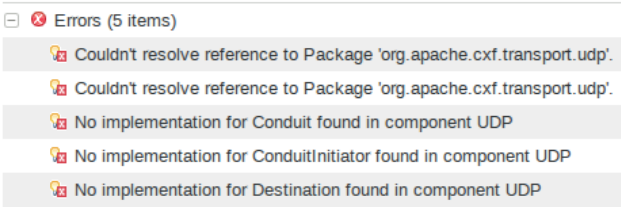

FIGURE 8.7: Consistency report for the new architectural component that was added during the first transformation step in our case study (as shown in Figure 8.6)

ConduitInitiator (which is the interface for the Transport Factory), Conduit, and Destination (specified by the UDP component's second Package rule and the InstanceOf rules in Figure 8.6). Therefore, the consistency checks of our Reconstruction Tool can evaluate whether the transport fulfills all the requirements that Apache CXF defines for its transports, that is, it enables architects to check all the constraints imposed by the style (see Section 3.3.1). At this point, we finished the transform-step and reached the architecture specification shown in Figure 8.2 at time (t+1).

- Generate step: Based on this architecture specification we used our Reconstruction Tool to generate a UML component view similar to the one already shown in Figure 8.4. However, the new component view also contains the new architectural component for the UDP transport. In addition, our Reconstruction Tool provides a consistency report that lists the issues the tool detected after the architecture specification was transformed. Initially, this report, as shown in Figure 8.7, consisted of multiple errors stating that no source code elements could be detected that adhered to the different elements in the architecture abstraction specification of the UDP component. This happens because no package org.apache.cxf.transport.udp existed in the source code yet and no realization of the interfaces Conduit, Destination, and

ConduitInitiator could be found for the UDP component. After the generate step, we have successfully created the UML component view as shown in Figure 8.2 at time (t+2).

- Implementation step: Based on the ADD, the UML component view, and the list of issues, the implementation step (of this iteration) follows. As we are studying an already existing system, instead of implementing the new component, we updated the Apache CXF source code from Version 2.6 to Version 2.7 which concluded our implementation step and brought us to time (t+3) in Figure 8.2.

- Check for violations step: In this step, we used the Reconstruction Tool to check whether the aforementioned issues still remained. While the Reconstruction Tool did not report any issue regarding the UDP component itself, it reported that there was a connection between the Core and the UDP component in the source code that was not covered in the architecture specification. After a short investigation, we decided that the detected violation was a violation in the ADD that occurred because a connector between those two components in the architecture was necessary.

We started a new iteration of our approach which led us back to iterate over the evolution step, applying every one of the identified steps Figure 8.2. During this second iteration, we first updated the ADD and then, in the transformation step, used the addConnector transformation (see Appendix A, Figure D.3) to create a new connector in the architecture specification that connects the Core component with the before added UDP component. After this, we skipped the implementation step, because it was not necessary to change the code, and again used the Reconstruction Tool to check for issues in the check for violations step. As no issues were reported any more, we finished this architecture evolution step.

8.4.2 Case Study 2: Apache CXF 2.7 to Apache CXF 3.0

In this case study, we retraced the version change from Apache CXF 2.7 to 3.0 using our approach. In this version change, the number of lines of source code grew to about 560.000 and consisted of 1224 modified, 296 new, and 111 removed classes. Figure 8.8 shows the high-level architectural component view of Apache CXF 2.7 before any changes were performed. This major update contained a number of changes to CXF that affect the implemented architecture. Among these changes are two new supported transports for which we performed the same architecture transformation steps as for the new UDP transport in the previous case study.

During this version change, two central components of the system, the CORE and the API components, have been merged into a single CORE component and the WSDL-related functionality has been split off into a component of its own called WSDL. Therefore, using our approach, we performed two evolution steps:

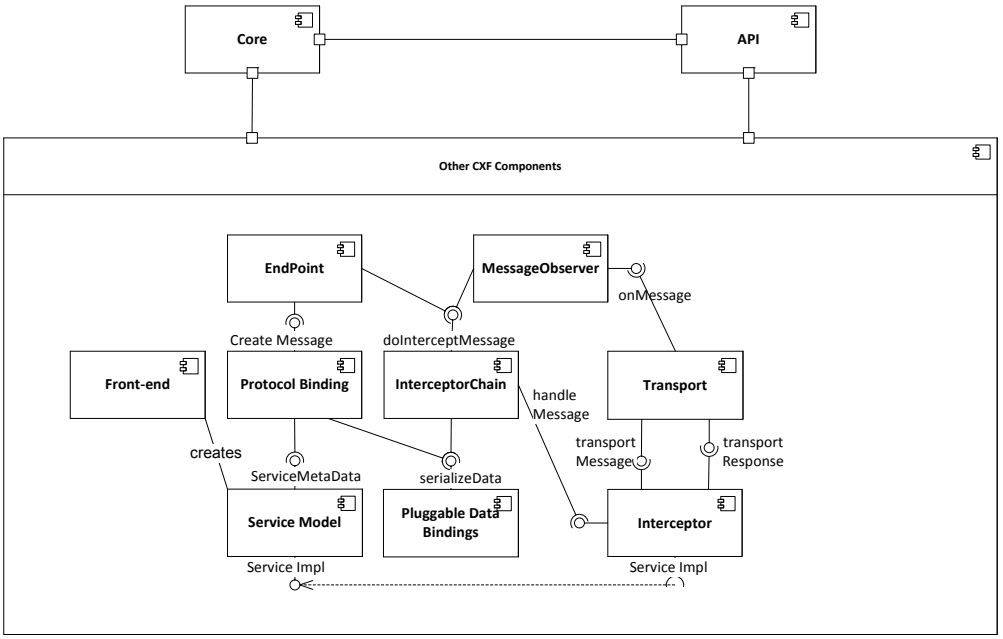

FIGURE 8.8: Architecture overview of Apache CXF 2.7

1. During the first evolution step, we documented the ADD to describe that the two components were merged (see Figure 8.9). In this ADD, we documented the problem, the outcome of the decision, named different possible solutions and arguments pro and contra these solutions, as well as related this decision to other affected decisions. Then we enacted the iterative process described in Section 8.2 as follows:

   - Transform step: We first updated the architecture abstraction specification of the CORE component by applying the updateAbstractionSpecification-transformation that is shown in Appendix D (Figure D.1). This merged the architecture specification of the API component and the CORE component and caused the automated consistency checks of our tool to report an architectural inconsistency as the architecture abstraction specification of the components now overlapped. We then applied the deleteComponent (see Appendix D, Figure D.2) transformation to delete the API component which solved the aforementioned architectural inconsistency but also required us to apply deleteConnector (see Appendix D, Figure D.4 and Figure D.5) for every connector that expressed a dependency to the API component. Then we searched the components whose connectors had been deleted to replace them with new connectors that expressed the dependency on the modified CORE component. However, all of the components already expressed this dependency so that no new connectors were necessary.
   - Generate step: Based on this updated architecture specification we used our Reconstruction Tool to generate a UML component view of Apache CXF and obtained a

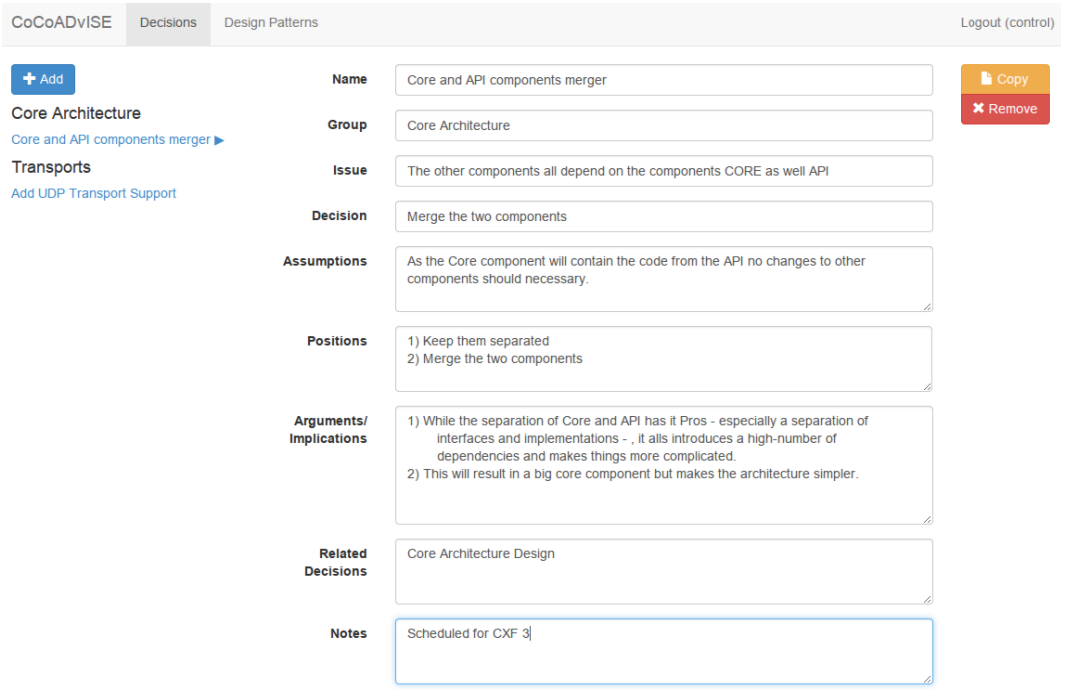

FIGURE 8.9: Documented architectural decision to merge the Components Core and API

consistency report that lists the issues the tool detected after the architecture specification was transformed. This list includes one message for each class that should be moved from the API to the CORE components.

- Implementation step: Similarly to the case study presented in the previous section, the implementation step (of this evolution step) follows and was based on the ADD, the UML component view, and the list of issues. As we are studying an already existing system, instead of implementing the new component, we updated the Apache CXF source code from Version 2.7 to Version 3.0 which concluded our implementation step and brought us to time (t+3) in Figure 8.2.
- Check for violations step: We again used the Reconstruction Tool to identify any remaining inconsistencies. While the list of issues from the Generate step were all fixed, a number of classes were identified as being misplaced. These were the classes that provide the WSDL functionality in Apache CXF which were separated from the new Core component and moved into a separate WSDL component. Until this point, we had not updated our architecture description with respect to this change. In order to deal with this change, we went back and performed the second transformation step described in the following paragraphs.

2. During the second evolution step, we first documented the ADD shown in Figure 8.10 to describe that the WSDL-related functionality was moved to a new component and then carried out the evolution as follows:

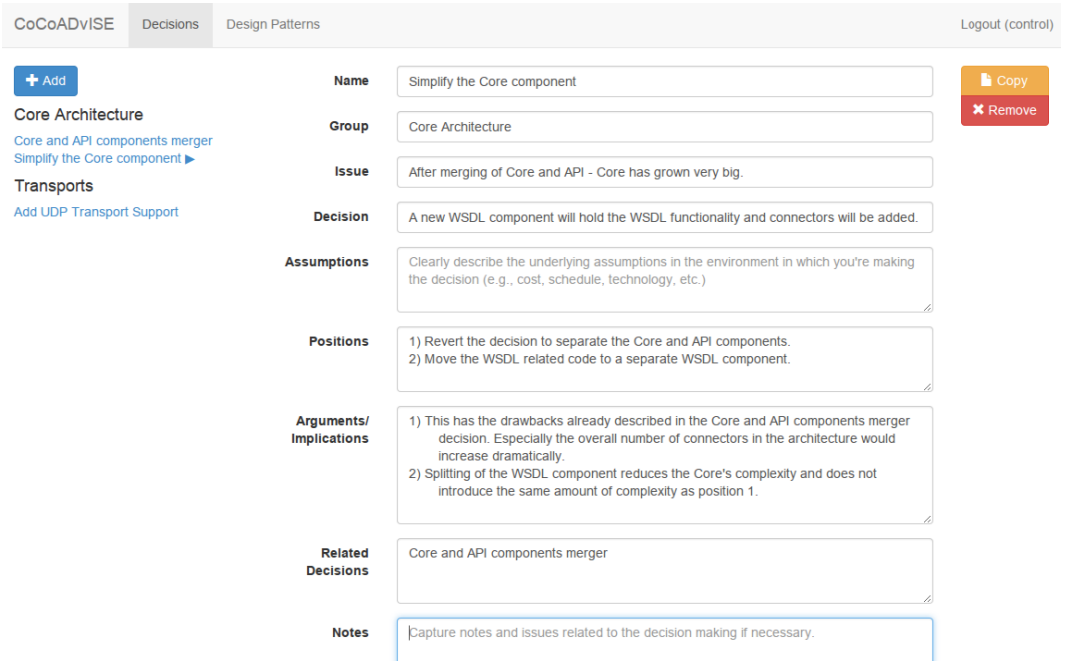

FIGURE 8.10: Docum om

```
Component RTWSDL
        consists of
        {
                Package (root.org.apache.cxf.wsdl)
                or
                Package (root.org.apache.cxf.wsdl11)
        }
        and not
        Package (root.org.apache.cxf.wsdl.http)
```

FIGURE 8.11: The WSDL component that now holds WSDL relevant functionality in Apache CXF 3.0

- Transform step: In this step, we used the addComponent and the updateAbstractionSpecification transformations to create the new WSDL component (shown in Figure 8.11) according to the abstraction specification and to remove those classes implementing the new component from the abstraction specification of the CORE component, respectively.

- Generate step: we regenerated the UML component model and, as we are applying the process to an existing system, we skipped the implementation step and directly checked for violations.
- Check for violations step: Once the above-mentioned changes to the architecture were carried out, no more violations were detected by the Reconstruction Tool and we finished the retrace of the version change from Apache CXF 2.7 to 3.0.

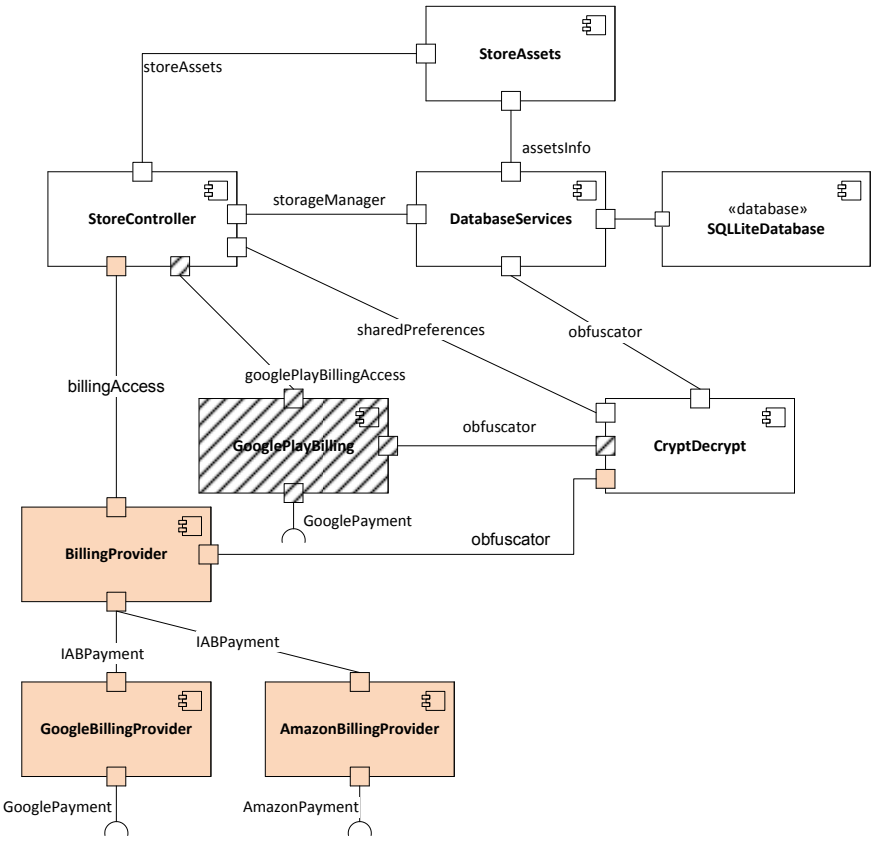

FIGURE 8.12: Soomla Architecture Overview showing the architecture for version 3.2 and the changes for version 3.3. Remove elements have a hatched background and new elements have a colored background.

8.4.3 Case Study 3: Soomla Store Version 3.2 to 3.3

For this case study we retraced the architectural changes for the Soomla Store from version 3.2 to version 3.3. Soomla is a framework that, in newer versions, helps implementing In-App purchases in Android, iOS, and Unity applications but was originally developed purely for Android. For this reason, the Soomla framework in version 3.2 only supported one payment method: Google Play Store. However, the requirement to also support payment via Amazon arose. Since payment via Google was so far "hardcoded" into the framework, this required to change Soomla's architecture to support multiple payment providers and thus introducing an abstract representation of payments and payment providers instead of using the ones provided by Google. Soomla's architecture and the changes applied in this case study are shown in Figure 8.12.

Therefore, using our approach, we again performed multiple evolution steps. During the first evolution step, first, we used our approach for the ADD to introduce a provider independent payment representation. In the second evolution step, we employed our approach for the introduction of the new Google payment provider and in the third evolution step we carried the ADD to add support for payment via Amazon. In the following we provide details for each one of these steps. Figure 8.13 shows the architecture abstraction specification for the new components. We have summarized

```
Component Billing
    consists of
      Package (root.com.soomla.store.billing, excludeChildren)
Component GooglePlayBilling
    consists of
      Package(root.com.soomla.store.billing.google) or
      {
           Package(root.com.soomla.store.billing.google)
           and
           InstanceOf(root.com.soomla.store.billing.IIabService)
      }
Component AmazonBilling
    consists of
      Package(root.com.soomla.store.billing.amazon) or
      {
           Package(root.com.soomla.store.billing.amazon)
           and
           InstanceOf(root.com.soomla.store.billing.IIabService)
      }
```

FIGURE 8.13: Architecture abstraction specification for the new provider independent Soomla Billing component and the new components that implement the payment providers for Google and Amazon

ID	Evolution Step	Architectural change	Arch. transformations
1	1	Added new Google-independent Billing-Provider component	addComponent addConnector (used multiple times - for all components that used have a connector to the old Billing component
2	1	Removing the old – Google-specific Billing component	deleteComponent deleteConnector (used multiple times - to delete all connectors to and from the old Billing component)
3	2	Adding the new Google-PlayBilling component that integrates with the new BillingProvider	addComponent addConnector (to create a connector between BillingProvider and GooglePlay-Billing)
4	3	Adding the new AmazonBilling component that integrates with the new BillingProvider	addComponent addConnector (to create a connector between BillingProvider and AmazonBilling)

TABLE 8.2: List of the architectural changes performed for the architecture evolution from Soomla v3.2 to Soomla v3.3.

all necessary architectural changes for this version change in Table 8.2. In the following, these three steps are described in detail.

1. During the first evolution step, we documented the ADD to replace the Google-specific billing component with a provider-independent billing component (see Appendix C, Figure D.7) and then enacted the iterative process described in Section 3 as follows:

    - Transform step: We first used the addComponent-transformation to create a new component named BillingProvider with a new architecture abstraction specification. Then,

we used the deleteConnector-transformation and the addConnector-transformation to replace all connectors to the old Billing component with connectors to the new BillingProvider component before using the deleteComponent-transformation to remove the old Google-specific Billing component.

- Generate step: Based on this updated architecture specification we used our Reconstruction Tool to generate a UML component view of Soomla's architecture and obtained a consistency report that lists the issues the tool detected after the architecture specification was transformed. In this case, these warnings were that the new component BillingProvider and the updated connectors were not present in the source code and that the implementation of the old Billing component still existed in the source code of the system.
- Implementation step: Similarly to the case studies presented in the previous sections, the implementation step (of this evolution step) follows and was based on the ADD, the UML component view, and the list of issues. As we are studying an already existing system, instead of implementing the new component, we updated the Soomla source code from Version 3.2 to Version 3.3 which concluded our implementation step and brought us to time (t+3) in Figure 8.2.
- Check for violations step: We again used the Reconstruction Tool to identify any remaining inconsistencies. All issues from the Generate step were fixed and the first evolution step was finished.

2. During the second evolution step, we documented the ADD to (re-)add support for payment via Google Play (see Appendix D, Figure D.8) and then enacted the iterative process described in Section 8.2 as follows:

   - Transform step: We used the addComponent-transformation to create a new component called GoogleBillingProvider which is connected to the BillingProvider component. Figure 8.13 shows the architecture abstraction specification for the new BillingProvider and GooglePlayBilling components.
   - Generate step: Based on this updated architecture specification we used our Reconstruction Tool to generate a UML component view of Soomla's architecture and obtained a consistency report that lists the issues the tool detected after the architecture specification was transformed. In this case, these warnings were that the new component and the new connectors were missed in the source code.
   - Implementation step: Again, as we are studying an already existing system, instead of implementing the new component, we updated the Soomla source code by adding the module for payment via Google, which is developed in a separate repository since Soomla supports multiple payment providers. In fact, both currently supported payment

providers are developed as separate modules and the Soomla framework itself does not contain any payment provider-specific code.

- Check for violations step: As before, we used the Reconstruction Tool to identify any remaining inconsistencies. All issues from the Generate step were fixed and the evolution step to (re-)add support for payment via Google Play was finished.

3. During the third evolution step, we documented the architectural decision to add support for payment via Amazon (shown in Appendix C, Figure D.9). In this ADD we noted only one position – the one to implement support for payment via Amazon using the new Billing architecture. We also documented the other related decisions, in this case the decision to utilize the provider-independent billing architecture and the decision to re-implement the payment via Google using this architecture. In the case of needing to revise the question on how to implement this AmazonBillingProvider, the related questions probably also need to be revised. Then we enacted the iterative process described in Section 8.2. As this consisted of exactly the same steps as in evolution step 2, we skip the details at this point. We have already shown the architecture abstraction specification for the Amazon-specific payment component in Figure 8.13 and its integration into the architecture in Figure 8.12. Like the Google-specific code, the code for payment via Amazon is also developed in a separate module that was added in the implementation step. No inconsistencies were found during the check for violations step and once we finished with this evolution step, we had also completed the architecture evolution of Soomla from version 3.2 to version 3.3.

8.4.4 Case Study 4: Soomla v3.3 Implementation of a New Custom Payment Provider for Payment via Carrier

In this case study, instead of analyzing an evolution already carried out, we implemented a new payment option for the Soomla framework that supports payment via a custom local payment provider that offers payment via a Restful API. Again, we first documented the ADD to add this functionality (see Appendix D, Figure D.10) and then used our approach:

- Transform step: We first used the addComponent-transformation to create a new component named RestfulBillingProvider with a new architecture abstraction specification (shown in Figure 8.14) and only one connector to the BillingProvider component.
- Generate step: Based on this updated architecture specification we used our Reconstruction Tool to generate a UML component view of the Soomla's architecture and obtained a consistency report that lists the issues the tool detected after the architecture specification was transformed. In this case, these issues were that the new component (RestfulBillingProvider) and the new connectors were not present in the source code. More specifically that

```
Component Billing
    consists of
      Package (root.com.soomla.store.billing, excludeChildren)
Component GooglePlayBilling
    consists of
      Package(root.com.soomla.store.billing.google) or
      {
            Package(root.com.soomla.store.billing.google)
            and
            InstanceOf(root.com.soomla.store.billing.IIabService)
      }
Component AmazonBilling
    consists of
      Package(root.com.soomla.store.billing.amazon) or
      {
            Package(root.com.soomla.store.billing.amazon)
            and
            InstanceOf(root.com.soomla.store.billing.IIabService)
      }
```

FIGURE 8.14: Architecture abstraction specification for the RestfulBilling component.

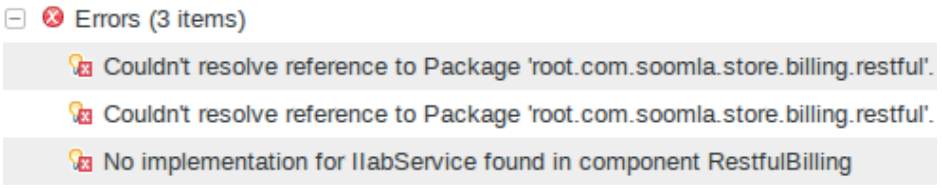

FIGURE 8.15: Consistency report created in the Generate step for the new RestfulBilling Component

no package com.soomla.store.billing.restful existed and that no classes existed in this package that implemented the IIabService interface which is defined by the BillingProvider (shown in Figure 8.15).

- Implementation step: In this step we tackled the issues reported by the tool one by one and first implemented the code to provide the desired functionality by implementing the interfaces defined by the BillingProvider. The most important of these interfaces is the IIabService which is the new provider-independent interface used throughout the Soomla store. As required in the architecture abstraction specification, we placed our code in the package com.soomla.store.billing.restful.
- Check for violations step: We then used the Reconstruction Tool to check for any remaining inconsistencies. However all issues from the Generate step were fixed and the evolution step was complete.

8.4.5 Discussion

These case studies illustrate the application of our iterative approach for the evolution of two existing software systems. Several advantages and strengths of our approach have been confirmed while they were carried out. First, they demonstrate the evolution of all artifacts of the software engineering process ranging from ADDs over the architectural component view down to the level of the source code. Second, we were able to ensure that changes to the architecture that are undertaken based on ADDs are recorded using QVT-o transformations to apply these changes and by providing traceability between different versions of the architecture abstraction specification.

Other strength that the case studies show is how software architect and software developer are supported by automatically providing feedback on the consistency of the architecture and the source code through the Reconstruction Tool which includes reports about missing or misplaced source code as well as the automatic checking of constraints that the software architect defines for the architecture of the system. In this way, our tool enables to check automatically the constraints related to the evolution style. For instance, in Section 8.4.1 it was shown that one of the constraints of the style imposes the implementation of specific interfaces. The use of the approach presented enables both architects to establish these constraints and developers to check whether the code satisfies them. Moreover, the second case study shows how the approach handles changes to the source code that are not previously documented in the architecture description of the software system. In addition, we are able to capture how and why specific evolution steps have happened as every evolution step was described by means of an ADD using ADvISE.

As already stated in the introduction, this is an important information for future evolution that the Reconstruction Tool lacks, if used on its own.

It is also important to mention the limitations of our approach so that it can be evaluated from a practical point of view. One of them, which we initially did not consider to be relevant, is that ADvISE and the Reconstruction Tool are currently two separate tools. An integration of those tools, which we plan in our future work, would improve the user experience for the software architect.

Another limitation of the approach, which it shares with all approaches that invest effort in the architecture documentation, is that it is not really applicable for small software systems as in these cases the source code often is enough for understanding the software architecture. However, this does not extend to middle and large scale systems where architecture documentation is necessary.

Finally, a limitation of the first three case studies is that they retrace the architectural evolution of a system instead of using the approach during the evolution itself. However, this allowed us to study real systems of considerable size instead of showcasing the application on artificial examples. Conversely, in the fourth case study we did not retrace architectural evolution but applied our

approach to perform architecture evolution by integrating an additional payment provider into the Soomla framework, in order to gain the view of a developer. This last case study allowed us to identify how valuable the information of the ADDs and consistency reports were to guide our implementation.

8.5 Conclusions and Future Work

In this chapter we introduce an approach for the joint evolution of software architecture and source code. By integrating our approach for documenting and maintaining architectural component views (Chapter 5) and architectural pattern instances (Chapter 6) with AK and evolution styles, we are able to support the evolution of a system not only on an architectural level, but also let the software architect provide the software developer with guidelines on how to adapt the source code in order to conform to the changed architecture and the constraints imposed by the architectural style. We are then able to perform automated consistency checking between the architectural specification and the source code. It is also noteworthy that all the different types of architectural changes, such as add component, delete component, etc., were automated by using QVT-operational, so that they can be easily traced and undone.

Moreover, we have also presented four case studies based on two real systems: Apache CXF and Soomla. They show the support for incremental changes to the architecture and the underlying source code provided by our approach. With this aim, it has been shown, how the software architect can check whether the architecture specification and the source code are consistent. Otherwise, we provide the software architect with an iterative workflow to reach a state such that the documented architecture and the code have been changed according to the architectural decisions and are consistent with each other. According to this conclusion, we can positively answer the research question that we stated in the introduction, as both software architects and developers can work together throughout the evolution process described keeping software architecture and code in sync.

In our future work we plan to automatically modify existing architecture specification guidelines between architectural component views and source code when the architectural component view is modified. Furthermore, we want to investigate if approaches for estimating the cost of architectural changes can be integrated with our approach.

9 Architecting for Decision Making About Code Evolution

9.1 Introduction

SA evolution is a complex activity, especially for large software projects with multiple development teams that might be located in different countries that work on different parts of the project in parallel, so there is a clear need to manage properly who is in charge of each requested change and how and when it will be carried out. In this chapter we propose an approach that attempts to solve Research Question 6, which we discussed in Section 2.1.

To support stakeholders with methods and tools to help in the evolution process, we have developed a DSL for making decisions about the evolution that provides architects with expressive power to describe which implementation tasks must be performed by the development team and which temporal dependencies exist among these tasks. Once the architect has specified these implementation tasks with the DSL, by using a model-to-text transformation, they are translated to Alloy [Jac11] to evaluate which are the possible decisions for realizing the architecture evolution in terms of the specified implementation tasks. We have integrated this approach with our Architecture Abstraction Specification Language introduced in Chapter 5, so that architects and/or developers can automatically update the specification of the architecture once an implementation task has been completed. This helps keeping two important assets of a software project (the SA and the source code) in sync. More specifically, we use a newer variant of the Architecture Abstraction Specification Language that supports explicit description of connectors like the variant used in Chapter 6 but without the support for pattern primitive annotations as they are not relevant for this chapter. This variant also has a simplified syntax for writing component definitions.

This way, our approach does not only allow to evolve SA and source code in sync, but also requires to define the architectural changes only once (when defining the implementation task) and the architecture description is updated automatically. Furthermore, we can use the Architecture Abstraction Specification Language's features. This proposal provides several advantages:

- First, it provides the software architect with facilities to automatically generate decision alternatives for carrying out the implementation tasks so that they can be easily distributed among the teams or team members.
- Second, it releases the software architect from the burden to manually update the architectural description because the defined implementation tasks are used to *automatically update the architecture specification*. This is very important as it helps to avoid the architectural drift and architectural erosion that usually emerge during the evolution.
- Third, the defined implementation tasks serve for the purpose of *creating a documentation of the evolution*. This is a very important question as several studies [Bra+00; Ozk+10b] carried out with subjects from both industry and academia have concluded that using the architectural documentation the time necessary to carry out the change-tasks could be shortend.

This chapter is structured as follows. In Section 9.2, we first present the DSL we have developed, and then we discuss the support for deciding on implementation steps. A case study that illustrates the feasibility of the approach is shown in Section 9.3. Finally, Section 9.4 concludes this chapter.

9.2 Architecting for Code Evolution

Whenever the code is being developed, the coding tasks are usually carried out in an iterative manner, so that no new component is developed from its very beginning to its end, but usually different components can be developed in parallel. However, the main problem is that there usually are *internal dependencies* among them that must be identified and considered whenever a system is being developed. These internal dependencies impose mainly *temporal constraints,* in terms of when the different features supported by each component should be developed. Let us illustrate this problem with a scenario, which we will use as a running example in the remainder of this chapter. As shown in Figure 9.1(a), initially two components *ComponentA* and *ComponentB* communicate with each other directly through a connector. Now let us assume that, due to new requirements, a distribution of these two components on different servers is necessary. This leads to an Architectural Design Decision (ADD) to implement a version of the broker pattern[1] between these two components. This architectural change is shown in Figure 9.1(b).This ADD leads to a number of *design decisions* and thus a number of different *implementation tasks*, whose timing is constrained by internal dependencies, as shown in Figure 9.2. Specifically, the following tasks need to be implemented in order to complete the implementation of this ADD.

- For using the Broker itself, a suitable middleware framework must be set up and configured.

[1]The broker pattern is a pattern for communication between distributed objects.

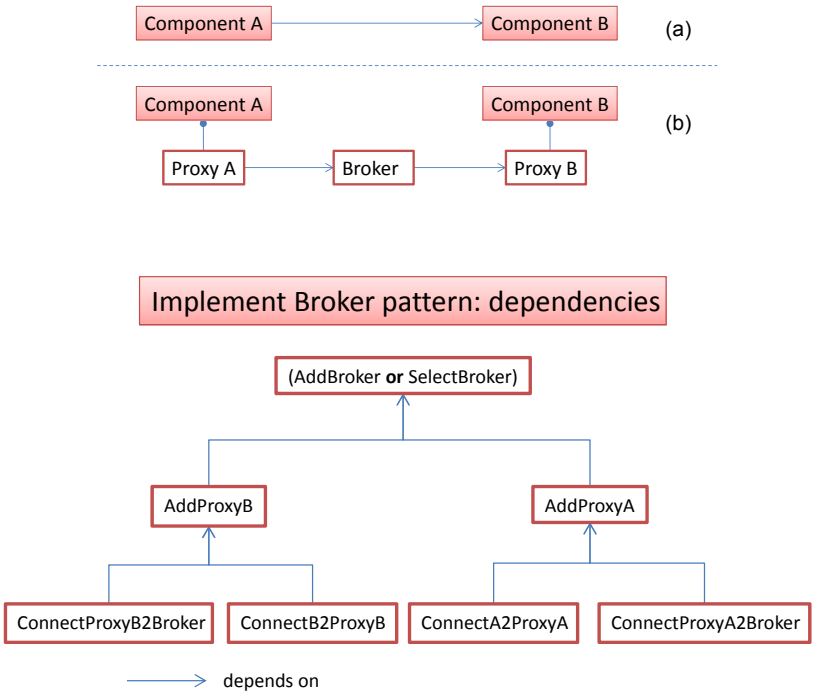

FIGURE 9.2: Planning an Evolution Step

- The above mentioned proxies for the two components need to be created.
- The proxies need to be wired to the broker. Moreover, the components need to be changed in order facilitate the new communication form. The direct connector needs to be removed and the usage of the proxies needs to be implemented. If dependency injection (DI) is used, at least the DI configuration needs to be changed, even if no changes to the components' implementation are necessary.

All of these tasks need to be completed in order to fully comply with the ADD to implement the broker pattern. Even in this small example, a number of temporal dependencies exist between the tasks at hand. The implementation of the proxies requires that the middleware for the Broker is set up and configured, the changes to *Component A* require the existence of the proxy for *Component A*, and the wiring of *Component A* with its proxy requires that the changes to the *Component A* itself are completed. The same or at least similar dependencies exist for *Component B*. These dependencies impose some order in which these tasks need to be completed. In a real world scenario with multiple development teams and more than two components involved in an architectural decision, this problem's complexity grows much further.

In our approach, a software architect defines the tasks and the constraints on the timing of the tasks (e.g. proxy must be implemented before proxy can be connected and used) in a domain specific language especially designed for planning code evolution, called *Evolution DSL*. This DSL allows the architect to specify: (i) a textual description of the implementations task, including any references to relevant ADDs; (ii) the temporal constraints or dependencies of the task; (iii) as

well as the changes to the system's architectural description based on the Architecture Abstraction Specification Language. We describe the technical details and the Evolution DSL in Section 9.2.1.

Based on these task definitions, our approach supports the architect and the developers during the evolution by automating the complex task of creating the possible decision alternatives for executing the given implementation tasks. We utilize the Alloy[2] model finder for automatically providing multiple possible alternatives for the order of the implementation tasks. These models are provided in graphical and textual form by Alloy. While the textual form supports an automatic interpretation, the graphical form shows which tasks do not have any dependencies to other tasks and thus can be implemented in a parallel fashion without running into any dependency issues. It also supports *easy identification of crucial tasks that need to be completed early*, as well as sets of implementation tasks that do not have dependencies outside the given set.

In our running example, two such sets can be identified: The first contains all tasks related to *Component A* and the second contains all tasks related to *Component B*, while the set up and configuration of the middleware of the Broker qualifies as a crucial task that might hinder further work as both identified sets depend on this task. The sets around *Component A* and *Component B* are good candidates for being developed by the same development team, because this team then can work independently from the other team(s) and is not hindered by any dependencies to tasks that are implemented elsewhere once the set up and configuration of the middleware of the Broker component is completed. Furthermore, the automatically generated decision alternatives ensure that no implementation tasks are started, before their dependencies are fulfilled. Finally, the *defined implementation tasks are used to automatically update the architecture description and serve for the purpose of creating a documentation of the evolution.* The technical details of this support are provided in Section 9.2.1.

9.2.1 DSL for Specifying the Code Evolution

In this section we describe the concepts and implementation of our Evolution DSL in detail. An important feature of the Evolution DSL is the tight integration with the architecture description itself, which enables to automatically apply the changes, specified in an implementation task to the architecture description, once it is completed. This *releases the software architect from the burden to manually update the architectural description after an implementation task is completed.*

This is why we have integrated the Evolution DSL, which was implemented in Xtext [Eclb], with the Architecture Abstraction Specification Language that we discussed and presented in Chapter 5.

[2] Alloy [Jac11] is a language to formally describe structures and a solver that takes the constraints of a model and finds structures that satisfy them.

```
ImplementationTask:
'Task' name=ID ':'
('status:' status=STATUS)?
('description:' description=STRING)?

// temporal rules
('precedes' precedes+=[ImplementationTask] (',' precedes+=[ImplementationTask])*)?
('directly precedes' directlyprecedes+=[ImplementationTask] (',' directlyprecedes+=[ImplementationTask])*)?
('henceforth requires' requires+=LogicRule (',' requires+=LogicRule)*)?
('in parallel with' inParallelWith+=[ImplementationTask] (',' inParallelWith+=[ImplementationTask])*)?
('succeeds' succeeds+=[ImplementationTask] (',' succeeds+=[ImplementationTask])*)?
('directly succeeds' directlysucceeds+=[ImplementationTask] (',' directlysucceeds+=[ImplementationTask])*)?
(optional?='is optional')?
('is incompatible with' prevents+=[ImplementationTask] (',' prevents+=[ImplementationTask])*)?
architectureChange=ArchitectureChange;

ArchitectureChange:
AddFeatureTask | AddConnectorTask | RemoveFeatureTask | RemoveConnectorTask | AddComponentTask |
RemoveComponentTask | ModifyComponentTask | ComplexTask;

ComplexTask:
'consists of:'
tasks+=TaskReference (',' tasks+=TaskReference)*;

AddComponentTask:
'architecture changes:'
'add component to' transformation=[archDSL::Transformation|TASKS_QUALIFIED_NAME]
componentToAdd=ComponentDef;
```

FIGURE 9.3: Excerpt of the Xtext grammar for the Evolution DSL shoing the rule for an implementation task, the different tpyes of tasks and two of the rules for specific tasks.

In order to facilitate the understanding of this chapter, Figure 9.3 shows an excerpt of the grammar for the definition of implementation tasks and the temporal rules for implementation tasks as well as the architectural changes supported. In the rule definition *AddComponentTask*, we can see that the Architecture Abstraction Specification Language's rules are reused. This enables the automatic application of the architectural changes from completed implementation tasks to the architectural description. This has been implemented as an Eclipse wizard that enables the architect to select which implementation tasks have been already completed and then, using a model-to-model transformation written in Xtend2 [Ecla], to update the architecture description.

An example of different tasks expressed in the DSL is presented in Figure 9.8. In this example, the complex task of adding a Broker between two Components A and B is divided into multiple subtasks, which consist of implementing the Broker itself (*AddBrokerFeature*), implementing the Proxies for Components A and B(*AddProxyA*, *AddProxyB*) and wiring all the components together. Some of these tasks have (temporal) dependencies. In this example, *ConnectA2Proxy* requires that *ProxyA* has been implemented before Component A can be wired to *ProxyA*. Also, *ConnectA2Proxy* is itself a complex task that consists of two subtasks, which should be carried out in close succession. In Figure 9.4 we skipped the tasks regarding Component B as they are very similar to the tasks regarding Component A.

Other dependencies that stem from organizational requirements (e.g. that the tasks will be split between independent teams of developers) can be modeled in the same way as constraints resulting

```
Task AddBroker:
description: "Tasks necessary for adding
the new broker to the architecture"
consists of:
AddBrokerFeature,
AddProxyA,
AddProxyB,
ConnectA2Proxy,
ConnectProxyA2Broker,
ConnectB2ProxyB,
ConnectProxyB2Broker

Task AddBrokerFeature:
description: "implement the broker functionality"
architecture changes:
add component to Frag
Component Broker consists of
Package("univie.swa.example.broker")

Task UpdateComponentA:
directly precedes ConnectA2Proxy
architecture changes:
replace feature Frag.ComponentA :
Package_univie_swa_example_original_package
with new feature:
Package("univieswa.example.A.usingBroker")
after Frag.ComponentA.Package_univie_swa_example_original_package

Task ConnectA2ProxyA:
description: "implement the conn. between comp. A and proxy A"
precedes ConnectB2ProxyB
succeeds AddProxyA
architecture changes:
add connector to Frag.ComponentA
connector to AddProxyA.ProxyA

Task ConnectProxyA2Broker:
description: "wire the proxy and the broker together"
succeeds AddProxyA,AddBrokerFeature
architecture changes:
add connector to AddProxyA.ProxyA
connector to AddBrokerFeature.Broker

Task AddProxyA:
description:"implement the proxy that hides the broker from comp. a"
architecture changes:
add component to Frag
Component ProxyA consists of
Package("univie.swa.example.proxyA")

Task ConnectA2Proxy:
succeeds AddProxyA
consists of:
UpdateComponentA,
ConnectA2ProxyA
```

FIGURE 9.4: Excerpt from the implementation tasks of the example for adding a broker.

from the implementation itself.

9.2.2 Generating Decision Alternatives for Evolution

Once the tasks are defined, we use the features provided by Xtext to automatically execute a model-to-text transformation that creates an Alloy model, which is used to generate the possible decision alternatives. Alloy [Jac11] is a structural modeling language based on first-order logic for expressing complex structural constraints and behavior. The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking. It allows us to define the concepts of basic and complex implementation tasks, the definition of specific implementation tasks and their constraints based on the abstract concepts, as well as the following (summarized) constraints that need to hold for all implementation task models:

- An implementation task is followed by a set of implementation tasks (*next* relation).
- A complex implementation task is an implementation task that consists of a set of implementation tasks (*consistsOf* relation).
- All defined implementation tasks need to be acyclic with respect to the *next* relation as well as the *consistsOf* relation.
- All defined implementation tasks need to exist in the solution and must be reachable. Either they are part of the initial tasks or they are reachable through an initial task.
- A complex implementation task is immediately followed by one of its subtasks.

```
//...
fact AcyclicImplementationTasks {
   no task: ImplementationTask| task in task.^next
}
fact AcyclicComplexImplementationTasks {
   no task: ComplexImplementationTask | task in task.^consistsOf
}
// ...
one sig AddProxyA extends ImplementationTask {}
one sig ConnectA2ProxyA extends ImplementationTask {}
// ...
pred show {
//..
all s1: AddProxyA, s2: ConnectA2ProxyA | s2 in s1.^next
}
run show for 5
```

FIGURE 9.5: Excerpt of the Alloy code for the introduced Broker example

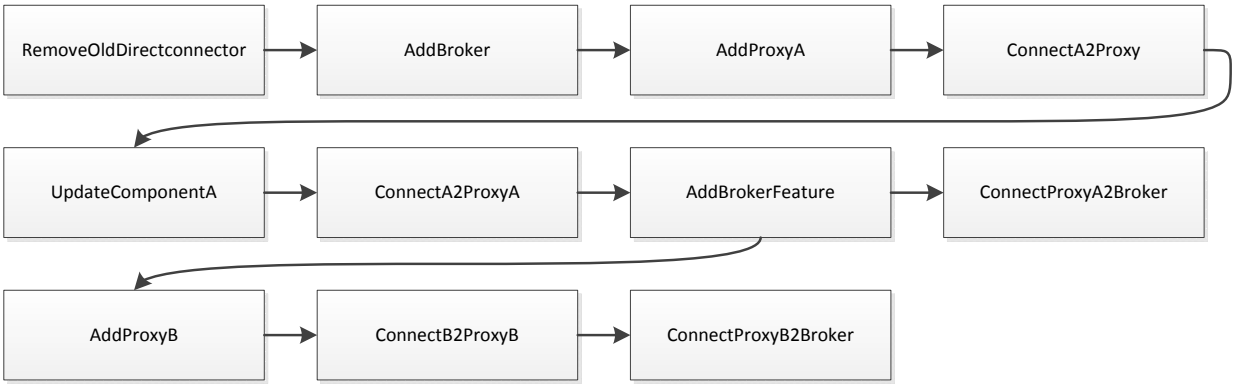

FIGURE 9.6: Decision alternative generated by Alloy for the Broker example.

- A complex implementation tasks precedes all its subtasks.
- Each implementation task can only be part of zero or one complex implementation tasks.

We show an excerpt of the Alloy code that was generated for the Broker example in Figure 9.5. In particular, we show the constraints that ensure that: (i) a task only occurs once, (ii) a task cannot be part of itself, (iii) the definition of the tasks *AddProxyA* and *ConnectA2ProxyA* as implementation tasks, (iv) finally, *AddProxyA* needs to be executed before the task *ConnectA2ProxyA*.

We then use the Alloy tool (version 4.2) to create multiple possible decision models that adhere to the identified constraints. These models are provided in a textual and a graphical representation by the tool. Figure 9.6 shows a possible order of the implementation tasks for the Broker example generated by Alloy.

Please note that a limitation of this approach arises through the use of Alloy, which, as a model finder that uses SAT solving for finding model instances, requires a suitable scope, as within this scope, the search for a model is complete, while the search itself is incomplete. For all our models, we chose a default scope of 5, because it was enough to find multiple solutions for all our generated models. Due to the size of architectural component models and due to the fact that our Alloy

models do not have free variables, our experience shows that for this subset of models a model instance can be found. If no model instance is found, the bound can be raised.

It is worth noting that this approach has been designed for evolving architecture and code in sync. When the code is changed first, the features of the Abstraction DSL can aid in ensuring consistency between architecture and code.

9.3 Case Study

In this section we describe our case study of Soomla, an open source framework for virtual economy operations in a single, cross-platform, SDK mainly used for mobile games [SOO]. In our case study, we describe the changes that were implemented from Version 3.2 to Version 3.3. Figure 9.7 shows an overview of Soomla's architecture and the respective changes to the architecture. In Version 3.2 Soomla's billing system only integrated the Billing API for Android provided by Google which was directly used throughout the system. However, since the need arose to support other billing providers as well, this was no longer suitable and the system needed to be evolved.

We described this evolution as a set of implementation tasks which replace the original provider-dependent *GooglePlayBilling* component with a new provider-independent billing component, and then (re-)implement the provider-specific parts based on the new billing infrastructure. The detailed implementation tasks and their architectural changes are shown in Figure 9.8.

Based on our description of the implementation tasks, an Alloy model was automatically generated by our model-to-text transformation implemented in Xtend. We then used the Alloy model finder to create the decision alternatives for executing the implementation tasks without violating any constraints. This was computed by Alloy in 149 ms and resulted in multiple possible alternatives for executing the implementation tasks at hand. This order ensures that all constraints are satisfied throughout the execution of the different implementation tasks

Once the implementation tasks were completed, we automatically added the architectural changes from the implementation tasks to the architectural description of Soomla using our wizard (see Figure 9.9), which we integrated into the DSL user-interface. This wizard then uses Xtend [Ecla] to apply the changes to the architecture description written in the Architecture Abstraction Specification Language.

This case study, as well as the running broker example, shows the applicability of the approach with respect to feasibility. The time required for finding suitable plans with Alloy was around 150ms for all presented examples on a Lenovo Thinkpad X240 with i5 Processor and 8 Gb RAM and a Samsung Evo 840 SSD. We think that in large projects with multiple developer teams, the effort necessary to use our approach is outweighed by the benefits of having a plan for executing

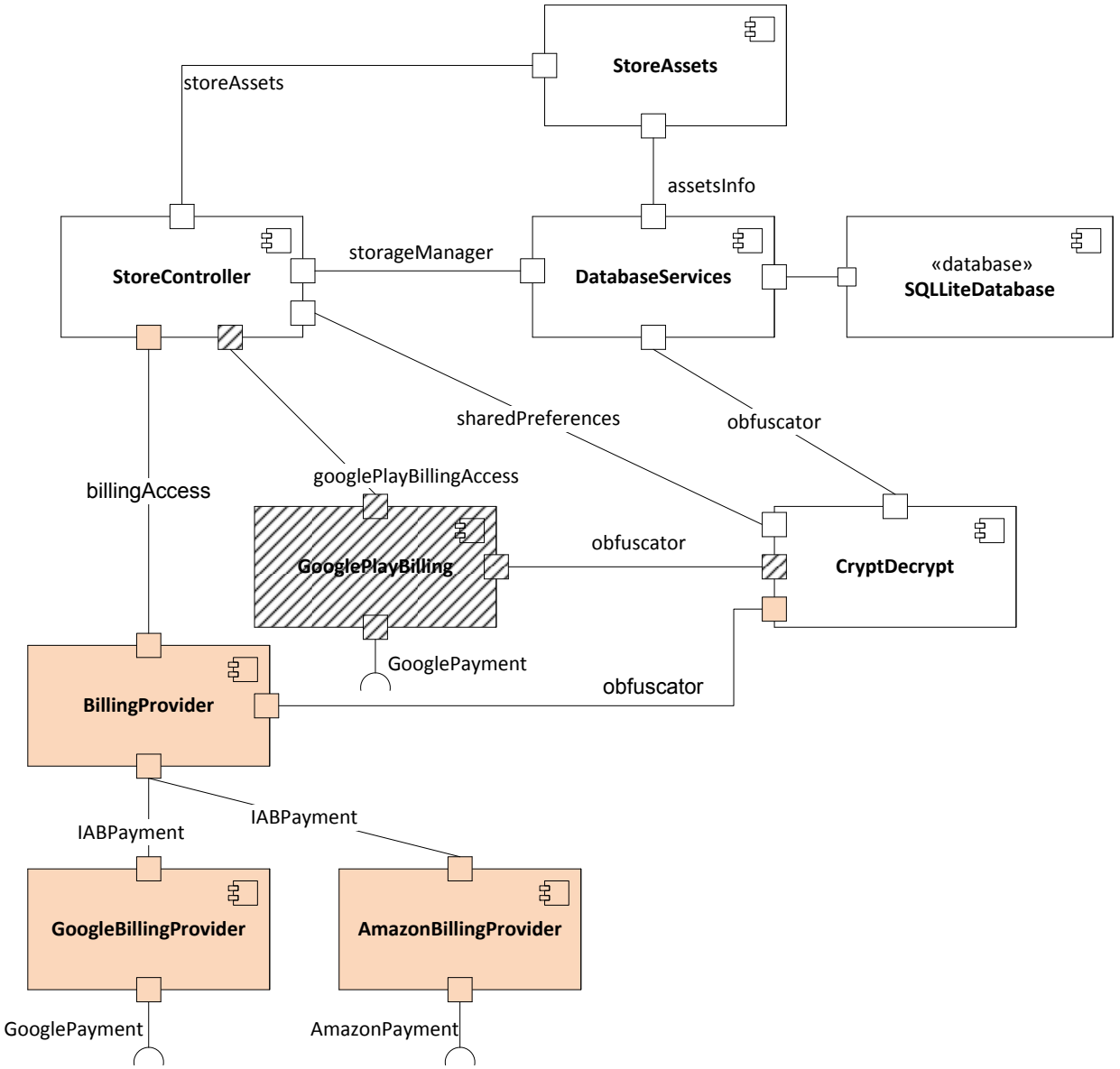

FIGURE 9.7: Architecture overview of Soomla with changes between version 3.2 and version 3.3.

the given tasks that shows which tasks can be executed in parallel, as well as which tasks are prerequisites to other tasks and thus should be prioritized.

9.4 Conclusion

In this chapter we present an approach for ensuring consistency between two important assets of a software project, namely software architecture and source code, during the evolution of a (large) system by describing an evolution as a set of implementation tasks. We provide a DSL that supports the description of implementation tasks based on their effects on a system's architecture, as well as the (temporal) constraints that exist between different implementation tasks. Besides the value of

```
Task ProviderIndependentBilling:
consists of:
ImplementBillingComponent,
WireBillingComponent,
SubstituteBillingInStoreController,
ImplementNewGoogleBillingProvider,
RemoveOldGoogleBillingProvider

Task ImplementBillingComponent:
description: "Implement a new abstract billing provider that is independent from any actual
billing providers"
architecture changes:
add component to Soomla
Component Billing
consists of     Package("root.com.soomla.store.billing",excludeChildren)

Task WireBillingComponent:
succeeds ImplementBillingComponent
architecture changes:
add connector to ImplementBillingComponent.Billing connector to Soomla.CryptDecrypt

Task SubstituteBillingInStoreController:
succeeds ImplementBillingComponent
consists of:
ConnectToAbstractBilling,
RemoveConntectorGooglePlayBilling

Task ConnectToAbstractBilling:
architecture changes:
add connector to Soomla.StoreController connector to ImplementBillingComponent.Billing

Task RemoveConntectorGooglePlayBilling:
precedes RemoveOldGoogleBillingProvider
architecture changes:
remove connector from Soomla.StoreController : connector_GooglePlayBilling

Task ImplementNewGoogleBillingProvider:
succeeds ImplementBillingComponent
architecture changes:
add component to Soomla
Component GoogleBilling
consists of
{
   Package("root.com.soomla.store.billing.google")
   or {
     Package("root.com.soomla.store.billing.google")
     and
     InstanceOf("root.com.soomla.store.billing.IIabService")
   }
}
connector to ImplementBillingComponent.Billing

Task RemoveOldGoogleBillingProvider:
architecture changes:
remove component Soomla.GooglePlayBilling
```

FIGURE 9.8: Implementation tasks for Soomla v3.2 to v3.3.

this DSL for documentation of architecture evolution, our approach supports tool-based guidance throughout the implementation tasks necessary for performing evolution. That is, based on the implementation task descriptions, we use Alloy models to calculate possible decision alternatives for code evolution under the given constraints that ensure the consistency of the evolution or warn the software developer if no viable code evolution decisions can be found. The integration with the architecture description helps keeping software architecture and source code in sync, avoiding drift and erosion. We show the applicability of the approach in a running example based on the implementation of the Broker pattern in an application as well as a real-life scenario for the evolution of the open-source in-app-purchase framework Soomla. This leads us to the conclusion that we have found a positive answer to Research Question 6, as this approach is able to support the architect during the evolution of a system by automatically providing possible plans for performing the different implementation tasks (evolution steps) that have to be completed during the evolution of a system and that are specified by the architect using our Implementation Task DSL.

This chapter contributes to an aspect of software evolution that is not discussed in the chapters

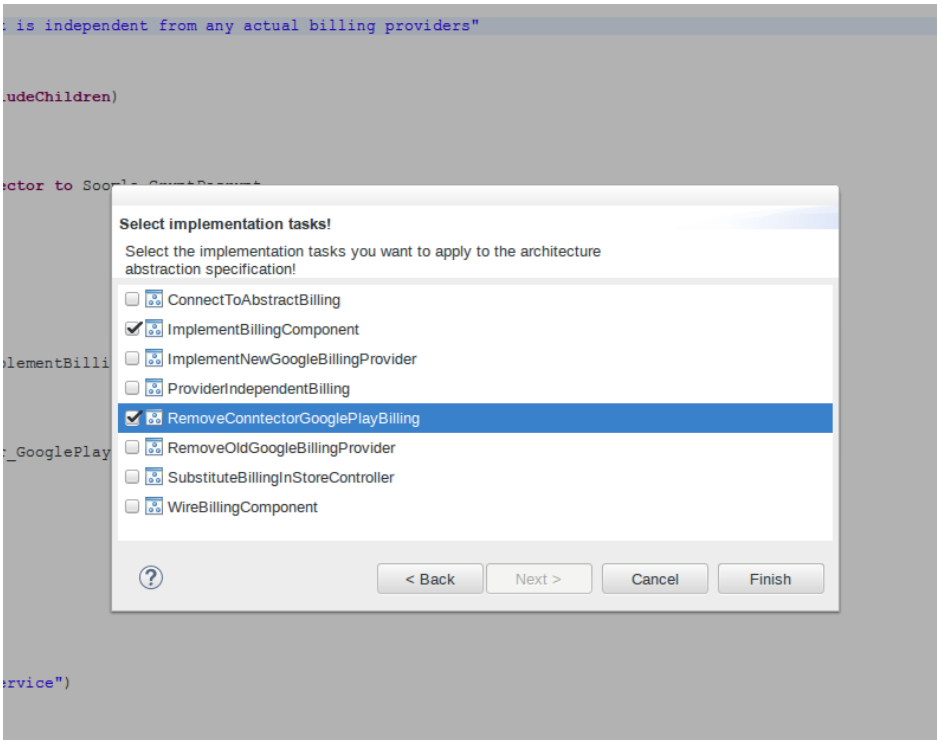

FIGURE 9.9: Wizard integrated into the DSL user-interface to add the architectural changes.

before, but provides the architect with an additional tool to ease the complex task of changing (evolving) a software system.

Part IV

Conclusions

10 Conclusions and Future Work

In this chapter we conclude this thesis by summarizing our contributions and discussing them with respect to the research questions identified in Chapter 2. In this context we also summarize the limitations of the approaches introduced in this thesis and finalize this thesis with a short discussion of open research challenges and possible future work.

10.1 Conclusions & Limitations

This thesis contributes to different topics in the area of software architecture evolution and documentation. It provides empirical evidence that component diagrams are beneficial to the understanding of novice software architects if they provide additional information that is not directly visible from the source code (Chapter 4). As expected, component diagrams did not lead to significant benefits if the same information was visible from the source code or if the component diagrams provided only vague guidance. With respect to Research Question 1, where we asked if architectural component views (in the form of UML component diagrams) have a positive effect during the understanding of a software system, we can state that our controlled experiment indicates a positive answer. However, in order to find a definite answer, more controlled experiments with other audiences like seasoned architects are needed. In an experiment with more experienced architects, it is possible, that component diagrams with vague guidance might also yield significant results.

In Chapter 5 we introduced an approach for documenting architectural component views, based on architectural abstraction specifications that allow relating architectural components with source code and keeping architecture and code consistent during a system's evolution. This approach is geared towards Research Question 2, and we can say that we can support the architect during the documentation of a system's architecture and a system's evolution. Our approach aids in keeping architecture and code consistent and thus reduces the risk of architecture drift and erosion. While we evaluated our approach on 5 different case studies of open source systems of different types, we did not perform a controlled experiment that compares the required effort with the benefits of the approach and thus we cannot completely negate the risk that the approach, although aiding with

consistency and documentation, might require more effort than the benefits it provides. In our experience, however, the positive effects in the long run outweigh the additional effort to create the architectural abstraction specifictions.

With respect to Research Question 3, where we asked whether we could support the software architect during the documentation of architectural patterns based on pattern primitives and keeping the documented pattern instances consistent with the source code during a system's evolution, we can state that our approach, based on pattern primitives, allows the semi-automatic identification and documentation of architectural patterns and, as it is integrated with Architecture Abstraction Specification Language from Chapter 5, it also aids during system evolution by providing consistency checking between the documented architectural pattern instances and the architecture components as well as between the architectural components and the source code. Similar to the Architecture Abstraction Specification Language, we also evaluated this approach on three different case studies using open source systems, but did not perform a controlled experiment or an industrial case study and thus, although our experience has shown that the effort for keeping a system consistent during evolution is signficantly reduced, we cannot completely eliminate the risk that the required effort of the approach are higher then the provided benefits.

In Chapter 7 we try to find answers to Research Question 4, which asked whether an integration and automatic calculation of understandability metrics during the creation and evolution of architectural component views was possible. Our proposed approach integrates the automatic calculation of different understandability metrics related to architectural components as well as source code classes and thus provides means to automatically ensure a minimum level of understandability. However, this approach is currently limited to understandability metrics, while additional, different quality metrics could also be considered.

In Chapter 8 we presented an iterative approach for evolving architecture and source code in-sync that also considers important architectural knowledge in the form of documented architectural design decisions. This approach is geared towards Research Question 5 and integrates our Architecture Abstraction Specification Language from Chapter 5 with Evolution Styles that were introduced by Barnes et al. [Bar+12] and with AKdES from Cuesta et al. [Cue+13]. Again, we evaluated our approach on 4 case studies of open source systems. A limitation of the first three case studies is that they retrace the architectural evolution of a system instead of using the approach during the evolution itself. However, this allowed us to study real systems of considerable size instead of showcasing the application on artificial examples. Conversely, in the fourth case study we did not retrace architectural evolution but applied our approach to perform architecture evolution by integrating an additional payment provider into the Soomla Android framework, in order to gain the view of a developer. This last case study allowed us to identify how valuable the information of the ADDs and consistency reports were to guide our implementation.

Another limitation of the approaches in this thesis, which they share with all approaches that invest effort in architecture documentation, is, that it might not make good economic sense for small software systems as in these cases the source code often is enough for understanding the software architecture. However, this does not extend to middle and large scale systems where architecture documentation is necessary to ease the understanding of a system's architecture.

The last research question, Research Question 6 asked whether it was possible to aid the architect in the description of the different, necessary evolution steps and their dependencies and provide plans for the specific evolution steps during the evolution of a software system. In Chapter 9, we propose an approach that aids the architect in the description of the different, necessary evolution steps (in the Chapter they are referred to as implementation tasks) and their dependencies in a way that integrates with our Architecture Abstraction Specification Language from Chapter 5 (existing software architecture documentation) and utilizes the Alloy model finder [Jac11] to automatically provide plans for the specific evolution steps during the evolution of a software system. As the approach from Chapter 9 is based on a SAT-solver, it only finds plans of a size smaller then the upper bound (but is complete within this bound). An unbounded search might lead to an indefinite long runtime of the SAT solver and would render the approach unusable. While a number of approaches exist that provide planing of interdependent tasks, to the best of our knowledge, these are not tightly integrated into the tools for architecture documentation and thus do not allow to specify the architectural changes resulting from each evolution step, while our approach supports the specification of architectural changes and furthermore allows for an adaptation of the software architecture documentation by automatically applying these architectural changes to the documented architecture.

10.2 Future Work

While this thesis contributes to the aforementioned research questions, additional work is required to completely solve those questions.

While our results show that architectural component diagrams are helpful for understanding certain aspects of a system, a definite answer to Research Question 1 requires additional work in this direction. This includes performing studies with more experienced participants as well as with other forms of architecture documentation in general and also with other representations of architectural component views (e.g. interactive and navigable ones like our Architecture Abstraction Specification Language).

The approach we presented in Chapter 6 currently supports the description of the structure of a system. While Kamal and Avgeriou proposed primitives for specifying behavioral patterns, the description of the behavior of architectural patterns and their semi-automatic identification still

presents open challenges. This includes providing additional architectural views that allow the architect to describe other aspect than the system's structure and structural architecture patterns, but also providing views that for example allow the description of a system's behavior. These views also need to relate to a systems source code and integrate with the architectural component view and architectural patterns provided in this thesis.

Another interesting challenge in the area of (architectural) pattern identification that remains, is the handling of partially implemented patterns. While our approach allows the definition of pattern variants by supporting optional pattern participants as well as multiple similar pattern participants, it currently does not provide ways to identify only partially implemented patterns. Interesting results might be achievable by applying heuristic or fuzzy approaches to this problem.

Additional research work is needed to explain why approaches proposed by academia for documenting architectural patterns are not yet or seldom adopted in industry. This relates to additional work regarding our approaches which we plan to apply in additional case studies performed by practitioners in an industrial environment.

Another topic of our future work will be the integration of automatic clustering approaches with our Architecture Abstraction Specification Language from Chapter 5, which would greatly improve the applicability of our approach in the field of architecture recovery, as the software could suggest one or more initial mappings which the architect can use as a starting point in our DSL and when the architect gains more knowledge about the system, she can refine the documented architecture.

In the future, we plan a better tool-support for our approach proposed in Chapter 8, as in its current form, we use a separate tool for documenting the architecture design decisions. An integration of this tool with our prototypes would improve usability as the architect could navigate from a design decision to the corresponding evolution steps, and from there, via the architectural description, to the source code that is affected by the architectural design decision.

Appendices

A Controlled Experiment on the Supportive Effect of Architectural Component Diagrams for Design Understanding of Novice Architects

In Table A.1 shows the raw data for the controlled experiment presented in Chapter 4. It lists, in anonymized form, the participants group, their score for the different questions, as well as their total score.

TABLE A.1: Raw data that contains all participants group affiliation (C for control group, E for experiment group), score per question, and total score

ID	Group	Q1	Q2	Q3	Q4	Q5	Q6	Q7	SUM
1	C	5	10	4	8	0	10	1	38
2	C	8	10	5	5	10	8	1	47
3	E	9	8	3	5	8	10	3	46
4	E	7	4	1	2	5	5	1	25
5	C	9	10	6	8	10	7	1	51
6	E	7	3	2	4	2	10	0	28
7	E	4	5	3	2	3	7	0	24
8	C	5	10	4	3	4	9	1	36
9	C	3	5	0	4	4	10	1	27
10	C	8	9	7	9	3	7	4	47
11	C	5	6	4	4	6	7	1	33
12	C	8	8	0	3	1	7	1	28
13	E	6	10	5	6	6	10	7	50
14	E	9	10	5	7	7	10	2	50
15	E	6	10	8	7	10	10	2	53
16	E	10	10	7	4	8	9	2	50
17	E	7	10	5	3	7	8	2	42
18	E	9	10	7	3	8	6	1	44
19	C	6	10	5	3	4	9	1	38
20	E	9	7	8	8	5	10	3	50

ID	Group	Q1	Q2	Q3	Q4	Q5	Q6	Q7	SUM
21	E	5	3	3	0	0	5	1	17
22	C	2	1	0	3	1	6	0	13
23	C	8	9	6	5	6	7	0	41
24	C	9	9	7	7	10	8	0	50
25	C	8	9	7	4	1	9	0	38
26	C	7	9	3	4	2	10	2	37
27	C	4	10	7	2	2	10	1	36
28	E	8	4	10	6	10	10	3	51
29	E	8	2	0	3	1	7	1	22
30	E	7	10	4	6	6	10	2	45
31	C	8	8	3	6	1	7	1	34
32	C	6	3	2	3	0	3	1	18
33	E	10	10	8	10	10	9	6	63
34	C	5	4	3	4	1	10	0	27
35	C	7	9	5	6	5	10	2	44
36	E	5	4	0	0	0	3	0	12
37	E	4	1	2	3	0	4	0	14
38	C	3	7	2	4	3	4	1	24
39	C	8	1	2	4	3	9	1	28
40	E	6	9	3	3	0	9	0	30
41	C	4	7	0	3	0	10	1	25
42	C	4	4	2	5	3	8	0	26
43	E	9	8	5	3	7	10	3	45
44	E	9	4	7	5	6	10	4	45
45	C	9	9	3	3	10	10	0	44
46	E	7	8	3	2	10	0	3	33
47	E	9	10	7	5	5	10	1	47
48	C	7	10	2	6	3	8	0	36
49	C	10	10	5	2	7	7	0	41
50	E	7	10	4	6	7	10	5	49
51	E	6	10	5	3	2	10	1	37
52	C	7	10	6	2	3	10	0	38
53	E	10	10	7	8	10	10	10	65
54	E	10	10	0	3	2	8	3	36
55	C	8	2	2	2	1	0	0	15
56	E	3	7	0	1	2	8	0	21
57	E	4	1	0	1	0	3	1	10

ID	Group	Q1	Q2	Q3	Q4	Q5	Q6	Q7	SUM
58	C	5	7	7	3	1	9	0	32
59	C	7	7	5	3	1	9	7	39
60	E	3	1	3	2	3	10	0	22
61	E	10	6	4	7	0	7	1	35
62	C	2	7	3	1	2	7	0	22
63	C	7	10	4	5	3	10	3	42
64	E	4	1	1	2	3	8	0	19
65	E	6	7	1	1	2	8	0	25
66	C	4	9	3	1	5	10	1	33
67	C	10	2	1	7	1	9	0	30
68	C	4	5	4	4	2	8	3	30
69	E	7	10	7	7	10	8	7	56
70	E	8	10	7	5	7	10	6	53

B Xtext Grammar of the Architecture Abstraction DSL

Listing 4 shows the complete Xtext grammar of our architectural abstraction DSL that we present in Chapter 5.

The source code for our proof-of-concept implementation can be found at https://git.swa.univie.ac.at/component-model/component-model-source (only for registered users) and is available under the MIT license.

```
grammar at.ac.univie.cs.swa.component.architectureabstraction.
  ArchitectureAbstractionDSL
with org.eclipse.xtext.common.Terminals

generate architectureAbstractionDSL
"http://www.univie.ac.at/cs/swa/component/architectureabstraction/
  ArchitectureAbstractionDSL"
import "http://www.eclipse.org/uml2/4.0.0/UML" as umlMM
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Transformation:
name=STRING
components+=(ComponentDef)+;

QUALIFIED_NAME returns ecore::EString:
ID ("." ID)*;

ComponentDef returns ComponentDef:
'Component' name=ID
'consists of' (expr=OrComposition)
connectors+=ConnectorAnnotation*;

ConnectorAnnotation:
{ConnectorAnnotation}
'connector to' targets+=[ComponentDef] (',' targets+=[ComponentDef])*
('implemented by' (implementingExpression+=OrComposition)?
('relation: ' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME]
(',' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME])*)?)?;
```

```
OrComposition returns Expression:
ExcludeComposition ({OrComposition.left=current} 'or' right=ExcludeComposition)
  *;

ExcludeComposition returns Expression:
AndComposition ({ExcludeComposition.left=current} 'and not' right=Primary)*;

AndComposition returns Expression:
Primary ({AndComposition.left=current} 'and' right=Primary)*;

Primary returns Expression:
NameFilter | RelationFilter | ExtensionFilter | '{' OrComposition '}';

NameFilter: PackageNameFilter | ClassNameFilter;

RelationFilter: ContainedInPackage | UsesFilter | UsedByFilter |
ChildOfFilter | Supertype | InstanceOf | IsClass | SpecificInterface;

PackageNameFilter: 'Package' '(' regEx=STRING ')';

ClassNameFilter: 'Class' '(' regEx=STRING ')';

UsesFilter:
'Uses' '(' relatedTo=[umlMM::Classifier|QUALIFIED_NAME] ')';

UsedByFilter:
'UsedBy' '(' relatedTo=[umlMM::Classifier|QUALIFIED_NAME] ')';

ChildOfFilter:
'ChildOf' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] ')';

Supertype:
'Supertype' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] ')';

ContainedInPackage:
'Package' '(' relatedTo=[umlMM::Package|QUALIFIED_NAME] (',' excludeChildren?='
  excludeChildren')? ')';

IsClass:
'Class' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] ')';

InstanceOf:
'InstanceOf' '(' relatedTo=[umlMM::Interface|QUALIFIED_NAME] (','
  excludeInterface?='excludeInterface')? ')';

SpecificInterface:
'Interface' '(' relatedTo=[umlMM::Interface|QUALIFIED_NAME] ')';
```

```
ExtensionFilter:
JavaExtensionFilter | XtendExtensionFilter;

JavaExtensionFilter:
'Java' '(' staticMethod=STRING ')';

XtendExtensionFilter:
'Xtend' '(' function=STRING ')';
```

Listing 4: Complete Xtext grammar of our architectural abstraction DSL

C Xtext Grammar of the Architecture Abstraction DSL (Modified Variant for the Identification of Architecture Patterns Based on Primivites)

Listing 5 shows the complete Xtext grammar of our modified architectural abstraction DSL that we present in Chapter 6. This version supports the annotation of architectural components and connectors with pattern primitives.

```
grammar at.ac.univie.cs.swa.component.architectureabstraction.
  ArchitectureAbstractionDSL
with org.eclipse.xtext.common.Terminals

generate architectureAbstractionDSL
"http://www.univie.ac.at/cs/swa/component/architectureabstraction/
  ArchitectureAbstractionDSL"
import "http://www.eclipse.org/uml2/4.0.0/UML" as umlMM
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.ac.at/univie/cs/swa/pattern/catalog/PatternCatalogDSL" as
  patcat

Transformation:
name=STRING
components+=(ComponentDef)+;

QUALIFIED_NAME returns ecore::EString:
ID ("." ID)*;

ComponentDef returns ComponentDef:
'Component' name=ID
'consists of'
(expr=OrComposition)
annotations+=ComponentAnnotation*
connectors+=ComponentConnector*;

OrComposition returns Expression:
ExcludeComposition ({OrComposition.left=current} 'or' right=ExcludeComposition)
  *;
```

```
ExcludeComposition returns Expression:
AndComposition ({ExcludeComposition.left=current} 'and not' right=Primary)*;

AndComposition returns Expression:
Primary ({AndComposition.left=current} 'and' right=Primary)*;

Primary returns Expression:
NameFilter |
RelationFilter |
ExtensionFilter |
'{' OrComposition '}';

NameFilter:
PackageNameFilter | ClassNameFilter;

RelationFilter:
ContainedInPackage | UsesFilter | UsedByFilter | ChildOfFilter | Supertype |
  InstanceOf | IsClass | SpecificInterface;

PackageNameFilter:
'Package' '(' regEx=STRING ')';

ClassNameFilter:
'Class' '(' regEx=STRING ')';

UsesFilter:
'Uses' '(' relatedTo=[umlMM::Classifier|QUALIFIED_NAME] ')';

UsedByFilter:
'UsedBy' '(' relatedTo=[umlMM::Classifier|QUALIFIED_NAME] ')';

ChildOfFilter:
'ChildOf' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] ')';

Supertype:
'Supertype' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] ')';

ContainedInPackage:
'Package' '(' relatedTo=[umlMM::Package|QUALIFIED_NAME] ((',' excludeChildren?='
  excludeChildren')? & (',' excludeNestedElements?='excludeNestedElements')?) ')'
  ;

IsClass:
'Class' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] (',' excludeChildren?='
  excludeChildren')?  ')';

InstanceOf:
```

```
'InstanceOf' '(' relatedTo=[umlMM::Interface|QUALIFIED_NAME] (','
  excludeInterface?='excludeInterface')? ')';

SpecificInterface:
'Interface' '(' relatedTo=[umlMM::Interface|QUALIFIED_NAME] (',' excludeChildren
  ?='excludeChildren')?  ')';

ExtensionFilter:
JavaExtensionFilter | XtendExtensionFilter;

JavaExtensionFilter:
'Java' '(' staticMethod=STRING ')';

XtendExtensionFilter:
'Xtend' '(' function=STRING ')';

/* primitive annotations */
PrimitiveAnnotation:
name=ID;

MyPrimitiveAnnotation returns PrimitiveAnnotation:
ConnectorAnnotation |
ComponentAnnotation;

ComponentAnnotation returns PrimitiveAnnotation:
GroupingAnnotation | LayeringAnnotation;

ConnectorAnnotation:
AggregationCascadeAnnotation | CompositeCascadeAnnotation | CallbackAnnotation |
   IndirectionAnnotation | VirtualConnectorAnnotation | SimpleConnectorAnnotation
   | ShieldAnnotation | TypingAnnotation;

ComponentConnector:
{ComponentConnector}
annotation=ConnectorAnnotation
('connector name: ' connectorName=ID)?
('implemented by'
//  implementationExpression=[umlMM::Relationship|QUALIFIED_NAME]
(implementingExpression+=OrComposition)?
('relation: ' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME]
(',' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME])*)?)?;

GroupingAnnotation returns PrimitiveAnnotation:
{GroupingAnnotation} 'is in Group (' groupId=ID ')';

LayeringAnnotation returns GroupingAnnotation:
{LayeringAnnotation} 'is at Layer (' groupId=ID ')';
```

```
ShieldAnnotation returns ConnectorAnnotation:
{ShieldAnnotation}
"is a Shield for" (targets+=[PatternInstancePrimitiveTarget] (',' targets+=[
  PatternInstancePrimitiveTarget])*)?;

TypingAnnotation returns ConnectorAnnotation:
{TypingAnnotation}
"is Typing for" (targets+=[PatternInstancePrimitiveTarget] (',' targets+=[
  PatternInstancePrimitiveTarget])*)?;

SimpleConnectorAnnotation returns ConnectorAnnotation:
{SimpleConnectorAnnotation}
'connector to' (targets+=[PatternInstancePrimitiveTarget] (',' targets+=[
  PatternInstancePrimitiveTarget])*)?;

VirtualConnectorAnnotation returns ConnectorAnnotation:
{VirtualConnectorAnnotation}
'virtually connected to' (targets+=[PatternInstancePrimitiveTarget] (',' targets
  +=[PatternInstancePrimitiveTarget])*)?;

IndirectionAnnotation returns ConnectorAnnotation:
{IndirectionAnnotation}
'indirection to' (targets+=[PatternInstancePrimitiveTarget] (',' targets+=[
  PatternInstancePrimitiveTarget])*)?;

CallbackAnnotation returns ConnectorAnnotation:
{CallbackAnnotation}
'callback with' target=[PatternInstancePrimitiveTarget] 'trigger interface'
  triggerInterface=[umlMM::Interface|QUALIFIED_NAME] 'callback interface'
  callbackInterface=[umlMM::Interface|QUALIFIED_NAME];

CompositeCascadeAnnotation returns AggregationCascadeAnnotation:
{CompositeCascadeAnnotation}
'composition of:' targets+=[ComponentDef] (','targets+=[ComponentDef])*;

AggregationCascadeAnnotation returns IndirectionAnnotation:
{AggregationCascadeAnnotation}
'aggregation of:'  targets+=[ComponentDef] (','targets+=[ComponentDef])*;

PatternInstancePrimitiveTarget: ComponentDef | GroupingAnnotation;
```

Listing 5: Excerpt of the Xtext grammar of our architectural abstraction DSL

D Reconciling Software Architecture and Source Code in Support of Software Evolution

D.1 Complete Specification of QVT-operational Transformations

In the following figures, the different transformations to apply the architectural changes are described. All of them have been implemented by means of QVT-operational and enable architects to retrace the architectural changes carried out during the evolution. As already described in Chapter 8 Section 8.3, each figure depicts one of the following transformations:

- updateAbstractionSpecification. As Fig. D.1 shows, this transformation modifies the architecture abstraction specification that defines how an architectural component relates to the source code.
- deleteComponent. Fig. D.2 depicts how an architectural component is eliminated from the architecture abstraction specification.
- addConnector. Fig. D.3 describes how a new connector between two architectural components can be created.
- deleteConnector: Fig. D.4 and Fig. D.5 show how a connector between two architectural components is deleted.

```
modeltype DSL uses
'http://www.univie.ac.at/cs/swa/component/architectureabstraction/ArchitectureAbstracti
onDSL';
transformation updateAbstractionSpecification
       (in dsl : DSL, in inputDSL: DSL , out outdsl : DSL);

main() {
       dsl.rootObjects()[DSL::Transformation]->
       map updateAbstractionSpecification(inputDSL.rootObjects()[DSL::Transformation]-
>asOrderedSet()
    ->first().map getComponent());
}

mapping  DSL::Transformation::updateAbstractionSpecification
       (in inputComp:DSL::ComponentDef): DSL::Transformation {
       result.name := self.name;
       result.components := self.components->
              map updateAbstractionSpecificationForComponent(inputComp);
}

mapping DSL::Transformation::getComponent () : ComponentDef {
       init {
       result := self.components->asOrderedSet()->first();
       }
}

mapping DSL::ComponentDef::updateAbstractionSpecificationForComponent
              (in inputComp:DSL::ComponentDef) : DSL::ComponentDef
//when {self.name = inputComp.name}
{
       init {
              result := self;
              if (self.name = inputComp.name) then {
                     result.expr := inputComp.expr;
              } endif;
       }
}
```

FIGURE D.1: QVT-o transformation for updating the architecture specification of a component

```
modeltype DSL uses
'http://www.univie.ac.at/cs/swa/component/architectureabstraction/ArchitectureAbstractionDSL';
transformation deleteComponentTransformation(in dsl : DSL, in inputDSL: DSL , out outdsl : DSL);
main() {
   dsl.rootObjects()[DSL::Transformation]-> map deleteComponent(inputDSL.rootObjects()[DSL::Transformation]
      ->asOrderedSet()->first().map getComponent());
}
mapping  DSL::Transformation::deleteComponent (in inputComp:DSL::ComponentDef): DSL::Transformation {
   result.name := self.name;
   result.components := self.components->excluding(inputComp)
}
mapping DSL::Transformation::getComponent () : ComponentDef {
   init {
      result := self.components->asOrderedSet()->first();
   }
}
```

FIGURE D.2: QVT-o transformation for deleting a component

```
modeltype DSL uses 'http://www.univie.ac.at/cs/swa/component/architectureabstraction/ArchitectureAbstractionDSL';
transformation addComponentTransformation(in dsl : DSL, in inputDSL: DSL , out outdsl : DSL);


main() {
dsl.rootObjects()[DSL::Transformation]-> map
                addConnector(inputDSL.rootObjects()[DSL::Transformation]->
                asOrderedSet()->first().map getComponent());
}


mapping  DSL::Transformation::addConnector (in inputComp:DSL::ComponentDef): DSL::Transformation {
                result.name := self.name;
                result.components := self.components->map addConnectorToComponent(inputComp);
}


mapping DSL::Transformation::getComponent () : ComponentDef {
                init {
                result := self.components->asOrderedSet()->first();
                }
}


mapping DSL::ComponentDef::addConnectorToComponent(in inputComp:DSL::ComponentDef) : DSL::ComponentDef
{
                init {
                result := self;
                if (self.name = inputComp.name) then {
                                var connector := inputComp.connectors->asOrderedSet()->first();
                                var addedConnector :=  self.connectors->including(connector);
                                result.connectors := addedConnector;
                } endif;
                }
}
```

FIGURE D.3: QVT-o transformation for adding a new connector to the architecture specification of a component

```
modeltype DSL uses
'http://www.univie.ac.at/cs/swa/component/architectureabstraction/ArchitectureAbstraction
DSL';
transformation deleteConnectorTransformation(in dsl : DSL, in inputDSL: DSL , out outdsl
: DSL);

main() {
       dsl.rootObjects()[DSL::Transformation]-> map
              deleteConnector(inputDSL.rootObjects()[DSL::Transformation]
              ->asOrderedSet()->first().map getComponent());
}

mapping  DSL::Transformation::deleteConnector (in inputComp:DSL::ComponentDef):
DSL::Transformation {
       result.name := self.name;
       result.components := self.components->map
              deleteConnectorFromComponent(inputComp);
}

mapping DSL::Transformation::getComponent () : ComponentDef {
init {
       result := self.components->asOrderedSet()->first();
       }
}

mapping DSL::ComponentDef::deleteConnectorFromComponent(in inputComp:DSL::ComponentDef) :
DSL::ComponentDef
{
       init {
       result := self;
       if (self.name = inputComp.name) then {
       var connector := inputComp.connectors->asOrderedSet()->first();
       var annotation := connector.annotation;
       // name based comparison necessary as these are different objects that
       // cannot be compared with equals
       var targetNames := annotation.oclAsType(ConnectorAnnotation)
                          ->targets()->collect(t| t.name);
       var connectorsToRemove := self.connectors ->
              select(c | c.annotation.targets()->collect(t|t.name)=(targetNames));
       var withoutRemovedConnector :=  self.connectors-(connectorsToRemove);
       result.connectors := withoutRemovedConnector;
       } endif;
       }
}
```

FIGURE D.4: QVT-o transformation for deleting a connector (part 1 of 2)

```
helper ConnectorAnnotation::targets() :
Sequence(PatternInstancePrimitiveTarget) {
	if(self.oclIsTypeOf(SimpleConnectorAnnotation)) then	{
		return self.oclAsType(SimpleConnectorAnnotation).targets;
	}endif;
	if(self.oclIsTypeOf(ShieldAnnotation)) then	{
		return self.oclAsType(ShieldAnnotation).targets;
	}endif;
		if(self.oclIsTypeOf(VirtualConnectorAnnotation)) then	{
		return self.oclAsType(VirtualConnectorAnnotation).targets;
	}endif;
		if(self.oclIsTypeOf(IndirectionAnnotation)) then	{
		return self.oclAsType(IndirectionAnnotation).targets;
	}endif;
		if(self.oclIsTypeOf(CallbackAnnotation)) then {
		var list : Sequence(PatternInstancePrimitiveTarget);
		list->append(self.oclAsType(CallbackAnnotation).target);

		return list;
	}endif;
		if(self.oclIsTypeOf(CompositeCascadeAnnotation)) then	{
		return self.oclAsType(CompositeCascadeAnnotation).targets;
	}endif;
		if(self.oclIsTypeOf(AggregationCascadeAnnotation)) then	{
		return self.oclAsType(AggregationCascadeAnnotation).targets;
	}endif;
	return null;
}
```

FIGURE D.5: QVT-o transformation for deleting a connector (part 2 of 2)

D.2 Exemplary Launch Configuration for Executing QVT-operational Transformations

```
<launchConfiguration type="org.eclipse.m2m.qvt.oml.QvtTransformation">
<booleanAttribute key="org.eclipse.m2m.qvt.oml.interpreter.clearContents1" value="true"/>
<booleanAttribute key="org.eclipse.m2m.qvt.oml.interpreter.clearContents2" value="true"/>
<booleanAttribute key="org.eclipse.m2m.qvt.oml.interpreter.clearContents3" value="true"/>
<mapAttribute key="org.eclipse.m2m.qvt.oml.interpreter.configurationProperties"/>
<intAttribute key="org.eclipse.m2m.qvt.oml.interpreter.elemCount" value="3"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.featureName1" value=""/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.featureName2" value=""/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.featureName3" value=""/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.module"
 value="platform:/resource/QVTTransformations/add_component.qvto"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetModel1"
 value="platform:/resource/QVTTransformations/cxf_transport_2.6.archabst"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetModel2"
 value="platform:/resource/QVTTransformations/addUDPComponentToCXF.archabst"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetModel3"
 value="platform:/resource/QVTTransformations/transformed/cxf_transport_2.6.withUDPComponent.archabst"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetType1" value="NEW_MODEL"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetType2" value="NEW_MODEL"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.targetType3" value="NEW_MODEL"/>
<stringAttribute key="org.eclipse.m2m.qvt.oml.interpreter.traceFile"
 value="platform:/resource/QVTTransformations/out2.archabst.qvtotrace"/>
<booleanAttribute key="org.eclipse.m2m.qvt.oml.interpreter.useTraceFile" value="true"/>
</launchConfiguration>
```

FIGURE D.6: Exemplary launch configuration for adding the new UDP component to CXF architecture abstraction specification.

D.3 Documented Architectural Decisions for the Soomla Case Studies

D.3.1 Case Study 3

In the following figures, the ADDs related to the Case Study 3, described in Section 8.4.3, are illustrated.

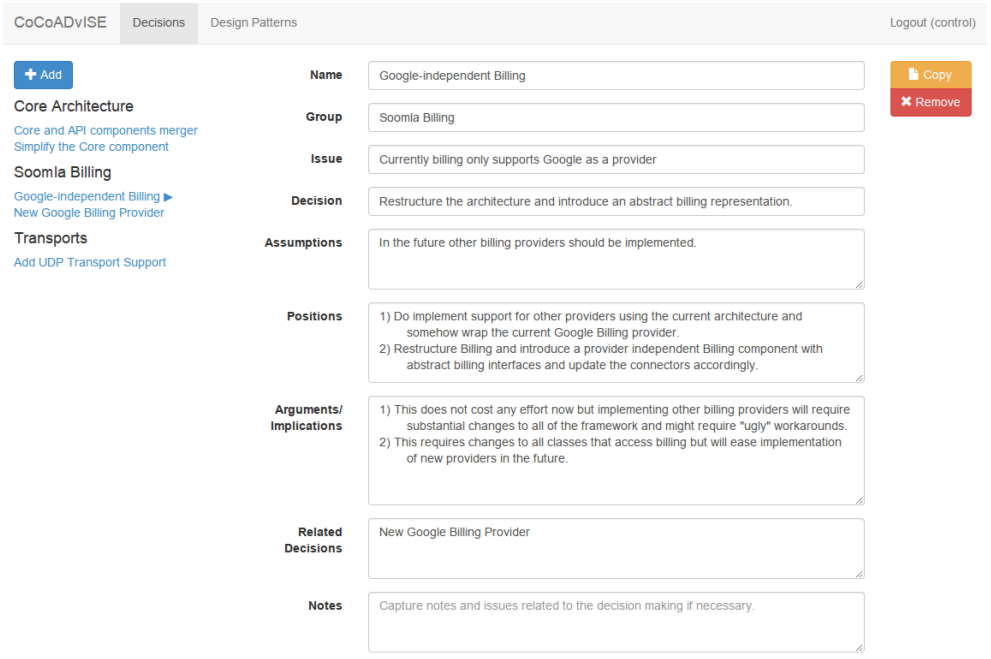

FIGURE D.7: Architectural decision to implement provider independent billing in the Soomla framework

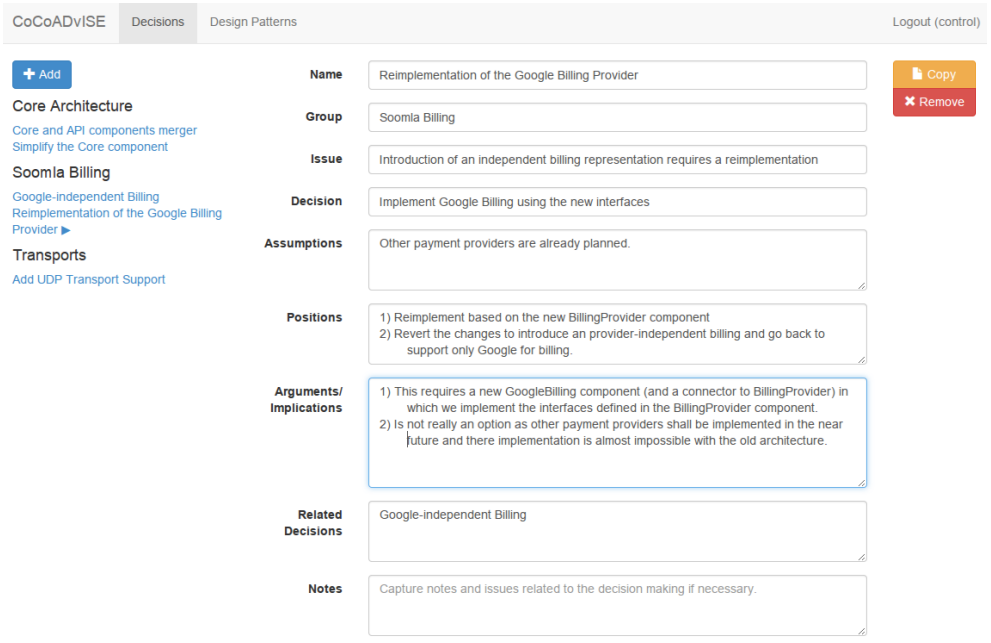

FIGURE D.8: Documented architectural decision to re-add support for Google Play Billing

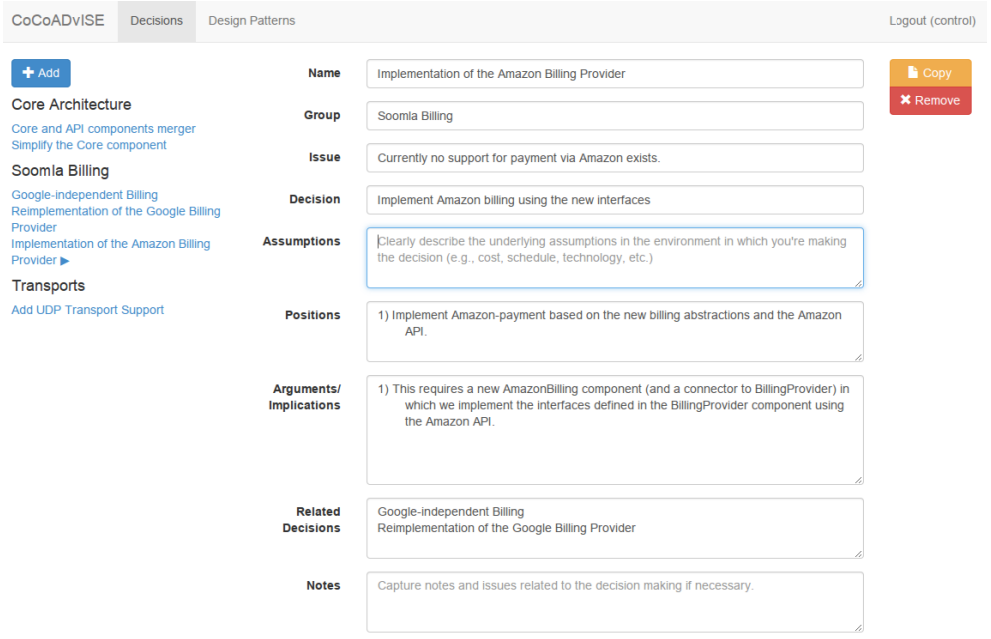

FIGURE D.9: Documend architectural decison to add support for payment via Amazon.

D.3.2 Case Study 4

The following figure shows the ADDs related to the Case Study 4, described in Section 8.4.4, are illustrated

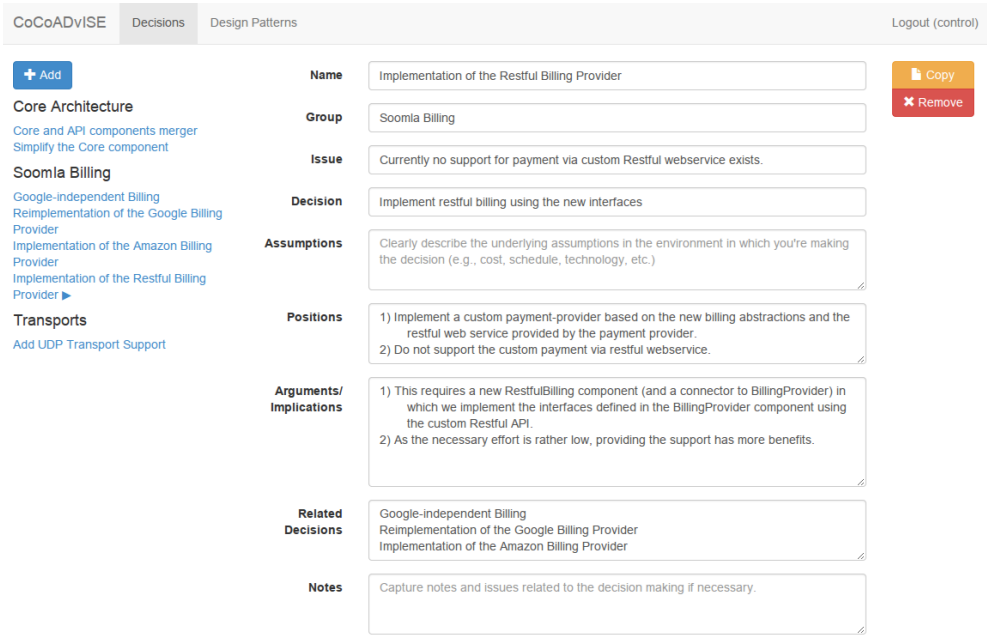

FIGURE D.10: Architectural decision to add support for our custom payment provider using a Restful service

Bibliography

[Abr+00] Fernando Brito e Abreu, Gonçalo Pereira, and Pedro Sousa. "A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems." In: *Proceedings of the Conference on Software Maintenance and Reengineering*. CSMR '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 13–. ISBN: 0-7695-0546-5. URL: http://dl.acm.org/citation.cfm?id=518900.795263.

[AG97] Robert Allen and David Garlan. "A formal basis for architectural connection." In: *ACM Trans. Softw. Eng. Methodol.* 6.3 [July 1997], pp. 213–249. ISSN: 1049-331X. DOI: 10.1145/258077.258078. URL: http://doi.acm.org/10.1145/258077.258078.

[Ahm+14] Aakash Ahmad, Pooyan Jamshidi, and Claus Pahl. "Classification and comparison of architecture evolution reuse knowledge-a systematic review." In: *J. Softw. Evol. Process* 26.7 [2014], pp. 654–691. ISSN: 20477473. DOI: 10.1002/smr.1643. URL: http://doi.wiley.com/10.1002/smr.1643.

[Aln+13] Awny Alnusair, Tian Zhao, and Gongjun Yan. "Automatic Recognition of Design Motifs Using Semantic Conditions." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: ACM, 2013, pp. 1062–1067. ISBN: 978-1-4503-1656-9. DOI: 10.1145/2480362.2480564.

[Apa] *Apache CXF*. http://cxf.apache.org. 2011.

[Are+04] G. Arevalo, F. Buchli, and O. Nierstrasz. "Detecting Implicit Collaboration Patterns." In: *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. 2004, pp. 122–131. DOI: 10.1109/WCRE.2004.18.

[AZ05] Paris Avgeriou and Uwe Zdun. "Architectural Patterns Revisited - A Pattern Language." In: *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*. Irsee, Germany, July 2005, pp. 1–39.

[Bab+07] Muhammad Ali Babar, Len Bass, and Ian Gorton. "Factors influencing industrial practices of software architecture evaluation: an empirical investigation." In: *Proceedings of the International Conference on the Quality of Software Architectures*. QoSA'07. Medford, MA: Springer-Verlag, 2007, pp. 90–107. ISBN: 3-540-77617-6, 978-3-540-77617-8.

[Bar+08] Olivier Barais, Anne Françoise Le Meur, Laurence Duchien, and Julia Lawall. "Software Architecture Evolution." In: *Softw. Evol.* Ed. by Tom Mens and Serge Demeyer. Springer Berlin Heidelberg, 2008, pp. 233–262. ISBN: 978-3-540-76439-7.

[Bar+12] Jeffrey M. Barnes, David Garlan, and Bradley R. Schmerl. "Evolution styles: foundations and models for software architecture evolution." In: *Softw. Syst. Model.* 13.2 [Nov. 2012], pp. 649–678. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0301-9. URL: http://link.springer.com/10.1007/s10270-012-0301-9.

[Bar+13] Jeffrey M Barnes, Ashutosh Pandey, and David Garlan. "Automated planning for software architecture evolution." In: *28th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE 2013)*. IEEE, Nov. 2013, pp. 213–223. ISBN: 978-1-4799-0215-6. DOI: 10.1109/ASE.2013.6693081. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6693081.

[Bar12] Jeffrey M Barnes. "NASA's advanced multimission operations system." In: *8th Int. ACM SIGSOFT Conf. Qual. Softw. Arch. (QoSA '12)*. New York, New York, USA: ACM Press, 2012, pp. 3–12. ISBN: 9781450313469. DOI: 10.1145/2304696.2304700. URL: http://dl.acm.org/citation.cfm?doid=2304696.2304700.

[Bas+03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2003.

[Ber+80] Edward H. Bersoff, Vilas D. Henderson, and Stanley G. Siegel. *Software configuration management. An investment in product integrity*. New York, New York, USA: Addison-Wesley, 1980.

[BF03] Z. Balanyi and R. Ferenc. "Mining Design Patterns from C++ Source Code." In: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on.* 2003, pp. 305–314. DOI: 10.1109/ICSM.2003.1235436.

[BG05] Sami Beydeda and Volker Gruhn. *Model-Driven Software Development*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN: 354025613X.

[BG09] Roberto Almeida Bittencourt and Dalton Dario Serey Guerrero. "Comparison of Graph Clustering Algorithms for Recovering Software Architecture Module Views." In: *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 251–254. ISBN: 978-0-7695-3589-0. DOI: 10.1109/CSMR.2009.28. URL: http://dl.acm.org/citation.cfm?id=1545011.1545446.

[Bha+12] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. "Graph-based analysis and prediction for software evolution." In: *ICSE'12*. 2012, pp. 419–429.

[Bif+08] Stefan Biffl, Muhammad Ali Babar, and Dietmar Winkler. "Impact of experience and team size on the quality of scenarios for architecture evaluation." In: *Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering*. EASE'08. Bari, Italy: British Computer Society, 2008, pp. 1–10.

[BJ94] Kent Beck and Ralph E. Johnson. "Patterns Generate Architectures." In: *Proceedings of the 8th European Conference on Object-Oriented Programming*. ECOOP '94. London, UK, UK: Springer-Verlag, 1994, pp. 139–149. ISBN: 3-540-58202-9.

[Boe+09] Remco C. de Boer, Patricia Lago, Alexandru Telea, and Hans van Vliet. "Ontology-driven visualization of architectural design decisions." In: *Jt. Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit. (WICSA/ECSA 2009)*. IEEE, Sept. 2009, pp. 51–60. ISBN: 978-1-4244-4984-2. DOI: 10.1109/WICSA.2009.5290791. URL: http://dblp.uni-trier.de/db/conf/wicsa/wicsa2009.html\#BoerLTV09.

[Bos04] Jan Bosch. "Software Architecture: The Next Step." In: *1st Eur. Work. Softw. Archit.* Heidelberg: Springer, 2004, pp. 194–199.

[Bou+10] Nelis Boucké, Danny Weyns, and Tom Holvoet. "Composition of architectural models: Empirical analysis and language support." In: *J. Syst. Softw.* 83.11 [Nov. 2010], pp. 2108–2127. ISSN: 0164-1212.

[Bou+11] Eric Bouwers, Jose P. Correia, Arie Deursen, and Joost Visser. "Quantifying the Analyzability of Software Architectures." In: *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*. Boulder, CO, USA: IEEE, June 2011, pp. 83–92. ISBN: 978-1-61284-399-5. DOI: 10.1109/wicsa.2011.20. URL: http://dx.doi.org/10.1109/wicsa.2011.20.

[BP00] Federico Bergenti and Agostino Poggi. "Improving UML Designs Using Automatic Design Pattern Detection." In: *In Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*. 2000, pp. 336–343.

[Bra+00] Lars Bratthall, Enrico Johansson, and Björn Regnell. "Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software." In: *2nd Int. Conf. Prod. Focus. Softw. Process Improv. (PROFES 2000)*. Springer, 2000, pp. 126–139.

[Bre+12] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. "A systematic review of software architecture evolution research." In: *Inf. Softw. Technol.* 54.1 [2012], pp. 16–40. ISSN: 09505849. DOI: 10.1016/j.infsof.2011.06.002. URL: http://linkinghub.elsevier.com/retrieve/pii/S0950584911001376.

[Bro+09] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. "Automated extraction of palladio component models from running enterprise Java applications." In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS '09. Pisa, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, 10:1–10. ISBN: 978-963-9799-70-7. DOI: http://dx.doi.org/10.4108/ICST.VALUETOOLS2009.7981. URL: http://dx.doi.org/10.4108/ICST.VALUETOOLS2009.7981.

[Bro13] Simon Brown. *Software Architecture for Developers*. Vancouver, BC, Canada: Leanpub, 2013.

[Bru+06] Bernd Bruegge, Andrea De Lucia, Fausto Fasano, and Genoveffa Tortora. "Supporting Distributed Software Development with fine-grained Artefact Management." In: *ICGSE*. IEEE, 2006, pp. 213–222. ISBN: 0-7695-2663-2.

[Bus+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. New York, NY, USA: John Wiley & Sons, Inc., 1996, p. 497. ISBN: 0-471-95869-7.

[CC79] Thomas D Cook and Donald T Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979.

[CL+09] José A. Cruz-Lemus, Marcela Genero, M. Esperanza Manso, Sandro Morasca, and Mario Piattini. "Assessing the understandability of UML statechart diagrams with composite states–A family of empirical studies." In: *Empirical Softw. Engg.* 14.6 [Dec. 2009], pp. 685–719. ISSN: 1382-3256. DOI: 10.1007/s10664-009-9106-z.

[Cle+01] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2001. Chap. 2. Evaluat, p. 368. URL: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/020170482X.

[Cle+02] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002. ISBN: 0201703726.

[Cle96] Paul C. Clements. "A Survey of Architecture Description Languages." In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IWSSD '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 16–. ISBN: 0-8186-7361-3. URL: http://dl.acm.org/citation.cfm?id=857204.858261.

[Cor+02] R. Correia, C. Matos, M. El-Ramly, R. Heckel, G. Koutsoukus, and L. Andrade. *Software Engineering at the Architectural Level: Transformation of Legacy Systems*. Tech. rep. University of Leicester, 2002.

[Cor+10] Anna Corazza, Sergio Di Martino, and Giuseppe Scanniello. "A Probabilistic Based Approach towards Software System Clustering." In: *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*. CSMR '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 88–96. ISBN: 978-0-7695-4321-5. DOI: 10.1109/CSMR.2010.36.

[Cue+13] Carlos E. Cuesta, Elena Navarro, Dewayne E. Perry, and Cristina Roda. "Evolution styles: using architectural knowledge as an evolution driver." In: *J. Softw. Evol. Process* 25.9 [Sept. 2013], pp. 957–980. ISSN: 20477473. DOI: 10.1002/smr.1575. URL: http://doi.wiley.com/10.1002/smr.1575.

[Cur+04] E. Curry, D. Chambers, and G. Lyons. "Extending Message-Oriented Middleware Using Interception." In: *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*. Edinburgh, Scotland, UK, May 2004, pp. 32–37.

[DB11] Markus von Detten and Steffen Becker. "Combining clustering and pattern detection for the reengineering of component-based software systems." In: *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*. QoSA-ISARCS '11. Boulder, Colorado, USA: ACM, 2011, pp. 23–32. ISBN: 978-1-4503-0724-6. DOI: http://doi.acm.org/10.1145/2000259.2000265. URL: http://doi.acm.org/10.1145/2000259.2000265.

[Dee+90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. "Indexing by Latent Semantic Analysis." In: *Journal of the American Society of Information Science* 41 [1990], pp. 391–407.

[Det+10] Markus Von Detten, Matthias Meyer, and Dietrich Travkin. "Reclipse-A Reverse Engineering Tool Suite." In: *Analysis* [2010]. URL: http://www.fujaba.de/uploads/tx_sibibtex/2010_TechReport_tr-ri-10-312_vDMT.pdf.

[Det11] Markus von Detten. "Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection Through Symbolic Execution." In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. PASTE '11. Szeged, Hungary: ACM, 2011, pp. 17–20. ISBN: 978-1-4503-0849-6. DOI: 10.1145/2024569.2024573.

[Die+08] Jens Dietrich, 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 '08. Ammersee, Germany: ACM, 2008, pp. 91–94. ISBN: 978-1-60558-112-5. DOI: http://doi.acm.org/10.1145/1409720.1409735. URL: http://doi.acm.org/10.1145/1409720.1409735.

[Din+14] Wei Ding, Peng Liang, Antony Tang, and Hans van Vliet. "Knowledge-based approaches in software documentation: A systematic literature review." In: *Inf. Softw. Technol.* 56.6 [June 2014], pp. 545–567. ISSN: 09505849. DOI: 10.1016/j.infsof.2014.01.008. URL: http://linkinghub.elsevier.com/retrieve/pii/S0950584914000196.

[DL+06] A. De Lucia, R. Oliveto, F. Zurolo, and M. Di Penta. "Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment." In: *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on.* 2006, pp. 317–326. DOI: 10.1109/ICPC.2006.28.

[DL+10] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. "Improving Behavioral Design Pattern Detection through Model Checking." In: *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on.* 2010, pp. 176–185. DOI: 10.1109/CSMR.2010.16.

[DL+11] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. "Improving Source Code Lexicon via Traceability and Information Retrieval." In: *IEEE Trans. Softw. Eng.* 37.2 [Mar. 2011], pp. 205–227. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.89. URL: http://dx.doi.org/10.1109/TSE.2010.89.

[DP+07] Massimiliano Di Penta, Antonella Santone, and Maria Luisa Villani. "Discovery of SOA Patterns via Model Checking." In: *2nd International Workshop on Service Oriented Software Engineering: In Conjunction with the 6th ESEC/FSE Joint Meeting.* IW-SOSWE '07. Dubrovnik, Croatia: ACM, 2007, pp. 8–14. ISBN: 978-1-59593-723-0.

[DP09] S Ducasse and D Pollet. "Software Architecture Reconstruction: A Process-Oriented Taxonomy." In: *IEEE Trans. Softw. Eng.* 35.4 [2009], pp. 573–591. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.19. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4815276.

[Eas+08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. "Selecting Empirical Methods for Software Engineering Research." In: *Guide to Advanced Empirical Software Engineering.* Springer London, 2008, pp. 285–311.

[Ecla] Eclipse. *Xtend.* URL: https://www.eclipse.org/xtend.

[Eclb] Eclipse. *Xtext.* URL: https://eclipse.org/Xtext.

[Eff+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. "Xbase: implementing domain-specific languages for Java." In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering.* GPCE '12. Dresden, Germany: ACM, 2012, pp. 112–121. ISBN: 978-1-4503-1129-8. DOI: 10.1145/2371401.2371419. URL: http://doi.acm.org/10.1145/2371401.2371419.

[Egy04] Alexander Egyed. "Consistent Adaptation and Evolution of Class Diagrams during Refinement." In: *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, ETAPS 2004 Barcelona, Spain*. Vol. 2984. Lecture Notes in Computer Science. Springer, 2004, pp. 37–53. ISBN: 3-540-21305-8.

[EH99] A. H. Eden and Y. Hirshfeld. "LePUS – Symbolic Logic Modeling of Object Oriented Architectures: A Case Study." In: *Second Nordic Workshop on Software Architecture - NOSA'99*. Ronneby, Sweden, Apr. 1999, pp. 1–14.

[Eig+03] Markus Eiglsperger, Michael Kaufmann, and Martin Siebenhaller. "A topology-shape-metrics approach for the automatic layout of UML class diagrams." In: *Proceedings of the 2003 ACM symposium on Software visualization*. SoftVis '03. San Diego, California: ACM, 2003, 189–ff. ISBN: 1-58113-642-0.

[Eli10] M.O. Elish. "Exploring the Relationships between Design Metrics and Package Understandability: A Case Study." In: *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. June 2010, pp. 144–147. DOI: 10.1109/ICPC.2010.43.

[Ell+03] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. *Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools*. Tech. rep. Florham Park NJ 07932, USA, 2003, p. 23.

[Fal+10] Davide Falessi, Muhammad Ali Babar, Giovanni Cantone, and Philippe Kruchten. "Applying empirical software engineering to software architecture: challenges and lessons learned." In: *Empirical Softw. Engg.* 15.3 [June 2010], pp. 250–276.

[Fei+09] Martin Feilkas, Daniel Ratiu, and Elmar Jurgens. "The loss of architectural knowledge during system evolution: An industrial case study." In: *17th IEEE Int. Conf. Progr. Compr.* IEEE Computer Society Press, May 2009, pp. 188–197. ISBN: 978-1-4244-3998-0. DOI: 10.1109/ICPC.2009.5090042. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5090042.

[Fer+10] Remo Ferrari, James A. Miller, and Nazim H. Madhavji. "A controlled experiment to assess the impact of system architectures on new system requirements." In: *Requir. Eng.* 15.2 [June 2010], pp. 215–233. ISSN: 0947-3602.

[FM06a] Tie Feng and Ji Maletic. "Applying dynamic change impact analysis in component-based architecture design." In: *Seventh ACIS Int. Conf. Softw. Eng. Artif. Intell. Networking, Parallel/Distributed Comput. SNPD 2006*. [2006]. DOI: 10.1109/SNPD-SAWN.2006.21. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1640665.

[FM06b] Tie Feng and Jonathan I. Maletic. "Applying Dynamic Change Impact Analysis in Component-based Architecture Design." In: *Proceedings of the Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*. SNPD-SAWN '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 43–48. ISBN: 0-7695-2611-X. DOI: 10.1109/SNPD-SAWN.2006.21. URL: http://dx.doi.org/10.1109/SNPD-SAWN.2006.21.

[FO00] Norman E. Fenton and Niclas Ohlsson. "Quantitative Analysis of Faults and Failures in a Complex Software System." In: *IEEE Trans. Softw. Eng.* 26.8 [Aug. 2000], pp. 797–814. ISSN: 0098-5589.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321127420.

[Fow10] Martin Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. 1st ed. Addison-Wesley Professional, 2010. ISBN: 0321712943.

[Fre+04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004. ISBN: 0596007124.

[Fri06] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006. ISBN: 0596528124.

[FS+12] Ana M. Fernández-Sáez, Marcela Genero, and Michel R. V. Chaudron. "Does the level of detail of UML models affect the maintainability of source code?" In: *Proceedings of the 2011th international conference on Models in Software Engineering*. MODELS'11. Wellington, New Zealand, 2012, pp. 134–148. ISBN: 978-3-642-29644-4. DOI: 10.1007/978-3-642-29645-1_15.

[Gal11] Matthias Galster. "Dependencies, traceability and consistency in software architecture: towards a view-based perspective." In: *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. ECSA '11. Essen, Germany: ACM, 2011, 1:1–1:4. ISBN: 978-1-4503-0618-8. DOI: 10.1145/2031759.2031761. URL: http://doi.acm.org/10.1145/2031759.2031761.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[Gar+09] David Garlan, Jeffrey M. Barnes, Bradley R. Schmerl, and Orieta Celiku. "Evolution styles: Foundations and Tool support for Software Architecture Evolution." In: *Jt. Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit. (WICSA/ECSA 2009)*. IEEE, 2009, pp. 131–140. ISBN: 978-1-4244-4984-2. DOI: 10.1109/WICSA.2009.5290799. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5290799.

[GC09] Varun Gupta and Jitender Kumar Chhabra. "Package coupling measurement in object-oriented software." In: *J. Comput. Sci. Technol.* 24.2 [Mar. 2009], pp. 273–283. ISSN: 1000-9000. DOI: 10.1007/s11390-009-9223-6. URL: http://dx.doi.org/10.1007/s11390-009-9223-6.

[GC12] Varun Gupta and Jitender Kumar Chhabra. "Package level cohesion measurement in object-oriented software." In: *J. Braz. Comp. Soc.* 18.3 [2012], pp. 251–266. URL: http://dblp.uni-trier.de/db/journals/jbcs/jbcs18.html#GuptaC12.

[Gen+08] Marcela Genero, José A. Cruz-Lemus, Danilo Caivano, Silvia Abrahão, Emilio Insfran, and José A. Carsí. "Assessing the Influence of Stereotypes on the Comprehension of UML Sequence Diagrams: A Controlled Experiment." In: *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. MoDELS '08. Toulouse, France: Springer-Verlag, 2008, pp. 280–294. ISBN: 978-3-540-87874-2. DOI: 10.1007/978-3-540-87875-9_20.

[GJ01] Yann-Gaël Guéhéneuc and Narendra Jussien. "Using explanations for design-patterns identification." In: *proceedings of the 1 st IJCAI workshop on Modeling and Solving Problems with Constraints*. AAAI Press, 2001, pp. 57–64.

[Gra+00] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. "Predicting Fault Incidence Using Software Change History." In: *IEEE Trans. Softw. Eng.* 26.7 [July 2000], pp. 653–661. ISSN: 0098-5589.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1st ed. Addison-Wesley Professional, 2009. ISBN: 0321534077, 9780321534071.

[Gru05] Lars Grunske. "Formalizing architectural refactorings as graph transformation systems." In: *Proc. - Sixth Int. Conf. Softw. Eng., Artif. Intell. Netw. Parallel/Distributed Comput. First ACIS Int. Work. Self-Assembling Wirel. Netw., SNPD/SAWN 2005*. Vol. 2005. 2005, pp. 324–329. ISBN: 0769522947. DOI: 10.1109/SNPD-SAWN.2005.37.

[GS09] David Garlan and Bradley Schmerl. "??vol: A tool for defining and planning architecture evolution." In: *Proc. - Int. Conf. Softw. Eng.* 2009, pp. 591–594. ISBN: 9781424434527. DOI: 10.1109/ICSE.2009.5070563.

[Guo+99] George Yanbing Guo, Joanne M. Atlee, and Rick Kazman. "A Software Architecture Reconstruction Method." In: *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 1999, pp. 15–34.

[HA10] Uwe van Heesch and Paris Avgeriou. "Naive architecting - understanding the reasoning process of students: a descriptive survey." In: *Proceedings of the 4th European conference on Software architecture*. ECSA'10. Copenhagen, Denmark: Springer-Verlag, 2010, pp. 24–37. ISBN: 3-642-15113-2, 978-3-642-15113-2.

[HA11] Uwe van Heesch and Paris Avgeriou. "Mature Architecting - A Survey about the Reasoning Process of Professional Architects." In: *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture*. WICSA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 260–269. ISBN: 978-0-7695-4351-2.

[Hai+15a] Thomas Haitzer, Elena Navarro, and Uwe Zdun. "Architecting for Decision Making About Code Evolution." In: *Proceedings of the 2015 European Conference on Software Architecture Workshops*. ECSAW '15. Dubrovnik, Cavtat, Croatia: ACM, 2015, 52:1–52:7. ISBN: 978-1-4503-3393-1. DOI: 10.1145/2797433.2797487. URL: http://doi.acm.org/10.1145/2797433.2797487.

[Hai+15b] Thomas Haitzer, Elena Navarro, and Uwe Zdun. "Reconciling software architecture and source code in support of software evolution." submitted. Oct. 2015.

[Haj+06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. "CodeQuest: scalable source code queries with datalog." In: *Proceedings of the 20th European conference on Object-Oriented Programming*. ECOOP'06. Nantes, France: Springer-Verlag, 2006, pp. 2–27. ISBN: 3-540-35726-2, 978-3-540-35726-1. DOI: 10.1007/11785477_2. URL: http://dx.doi.org/10.1007/11785477_2.

[Han+11] Klaus Marius Hansen, Kristjan Jonasson, and Helmut Neukirchen. "Controversy Corner: An empirical study of software architectures' effect on product quality." In: *J. Syst. Softw.* 84.7 [July 2011], pp. 1233–1243. ISSN: 0164-1212.

[Har+07] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. "Using Patterns to Capture Architectural Decisions." In: *IEEE Softw.* 24.4 [2007], pp. 38–45.

[Har+95] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. "Reverse engineering to the architectural level." In: *Proceedings of the 17th international conference on Software engineering*. ICSE '95. Seattle, Washington, United States: ACM, 1995, pp. 186–195.

[Hee+12] Uwe van Heesch, Paris Avgeriou, Uwe Zdun, and Neil Harrison. "The supportive effect of patterns in architecture decision recovery - A controlled experiment." In: *Sci. Comput. Program.* 77.5 [May 2012], pp. 551–576. ISSN: 0167-6423.

[Hei+11] W. Heijstek, T. Kuhne, and M. R V Chaudron. "Experimental Analysis of Textual and Graphical Representations for Software Architecture Design." In: *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. 2011, pp. 167–176. DOI: 10.1109/ESEM.2011.25.

[Hei+96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. "Automated Consistency Checking of Requirements Specifications." In: *ACM Trans. Softw. Eng. Methodol.* 5.3 [July 1996], pp. 231–261. ISSN: 1049-331X. DOI: 10.1145/234426.234431.

[Her+13] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M Gonzalez-Barahona. "The evolution of the laws of software evolution." In: *ACM Comput. Surv.* 46.2 [Nov. 2013], pp. 1–28. ISSN: 03600300. DOI: 10.1145/2543581.2543595. URL: http://dl.acm.org/citation.cfm?doid=2543581.2543595.

[Heu+03] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. "Automatic Design Pattern Detection." In: *Proceedings of the 11th IEEE International Workshop on Program Comprehension*. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 94–104. ISBN: 0-7695-1883-4.

[Hev+04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. "Design science in information systems research." In: *MIS Q.* 28.1 [Mar. 2004], pp. 75–105. ISSN: 0276-7783.

[HH04] Ahmed E. Hassan and Richard C. Holt. "Using Development History Sticky Notes to Understand Software Architecture." In: *Proceedings of the 12th IEEE International Workshop on Program Comprehension*. IWPC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 183–.

[HM76] J. W. Hunt and M. D. McIlroy. *An Algorithm for Differential File Comparison*. Tech. rep. CSTR 41. Murray Hill, NJ: Bell Laboratories, 1976.

[Hof+00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.

[Hol02] Ric Holt. "Sofware Architecture as a Shared Mental Model." In: *Proc. ASERC Work. Softw. Archit.* Ed. by University of Alberta. 2002.

[Hol05] Steve Holzner. *Ant: The Definitive Guide, Second Edition*. 2nd ed. O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00609-9.

[Hol98] Richard C. Holt. "Structural Manipulations of Software Architecture Using Tarski Relational Algebra." In: *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*. WCRE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 210–. ISBN: 0-8186-8967-6.

[Hun+04] Bruce C. Hungerford, Alan R. Hevner, and Rosann W. Collins. "Reviewing Software Diagrams: A Cognitive Study." In: *IEEE Trans. Softw. Eng.* 30.2 [Feb. 2004], pp. 82–96. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.1265814. URL: http://dx.doi.org/10.1109/TSE.2004.1265814.

[Hun+08] Sascha Hunold, Matthias Korch, Björn Krellner, Thomas Rauber, Thomas Reichel, and Gudula Rünger. "Transformation of Legacy Software into Client/Server Applications through Pattern-Based Rearchitecturing." In: *32nd Annu. IEEE Int. Comput. Softw. Appl. Conf.* IEEE, 2008, pp. 303–310. ISBN: 978-0-7695-3262-2. DOI: 10.1109/COMPSAC.2008.158. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4591573.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683.

[HZ12] Thomas Haitzer and Uwe Zdun. "DSL-based Support for Semi-automated Architectural Component Model Abstraction Throughout the Software Lifecycle." In: *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '12. Bertinoro, Italy: ACM, 2012, pp. 61–70. ISBN: 978-1-4503-1346-9. DOI: 10.1145/2304696.2304709.

[HZ13] Thomas Haitzer and Uwe Zdun. "Controlled Experiment on the Supportive Effect of Architectural Component Diagrams for Design Understanding of Novice Architects." English. In: *Software Architecture*. Ed. by Khalil Drira. Vol. 7957. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 54–71. ISBN: 978-3-642-39030-2. DOI: 10.1007/978-3-642-39031-9_6.

[HZ14] Thomas Haitzer and Uwe Zdun. "Semi-automated architectural abstraction specifications for supporting software evolution." In: *Sci. Comput. Program.* 90 [Sept. 2014], pp. 135–160. ISSN: 01676423. DOI: 10.1016/j.scico.2013.10.004.

[HZ15] Thomas Haitzer and Uwe Zdun. "Semi-automatic Architectural Pattern Identification and Documentation Using Architectural Primitives." In: *J. Syst. Softw.* 102.C [Apr. 2015], pp. 35–57. ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.12.042.

[Iee] *IEEE Standard Glossary of Software Engineering Terminology*. Tech. rep. 1990, pp. 1+. DOI: 10.1109/ieeestd.1990.101064. URL: http://dx.doi.org/10.1109/ieeestd.1990.101064.

[IK04] Igor Ivkovic and Kostas Kontogiannis. "Tracing Evolution Changes of Software Artifacts through Model Synchronization." In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 252–261. ISBN: 0-7695-2213-0. URL: http://dl.acm.org/citation.cfm?id=1018431.1021433.

[ISO11] ISO/IEC/IEEE. "Systems and software engineering – Architecture description." In: *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* [Jan. 2011], pp. 1 –46. DOI: 10.1109/IEEESTD.2011.6129467.

[Ive+04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Rodrigo Oviedo Silva. *Documenting Component and Connector Views with UML 2.0*. Tech. rep. CMU/SEI-2004-TR-008. Software Engineering Institute (Carnegie Mellon University), 2004.

[Jac11] Daniel Jackson. *Software Abstractions. Logic, Language and Abstractions*. MIT Press, 2011.

[Jam+13] Pooyan Jamshidi, Mohammad Ghafari, Aakash Ahmad, and Claus Pahl. "A Framework for Classifying and Comparing Architecture-centric Software Evolution Research." In: *2013 17th Eur. Conf. Softw. Maint. Reengineering*. IEEE, Mar. 2013, pp. 305–314. ISBN: 978-0-7695-4948-4. DOI: 10.1109/CSMR.2013.39. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6498478.

[Jan+07] Anton Jansen, Jan van der Ven, Paris Avgeriou, and Dieter K. Hammer. "Tool Support for Architectural Decisions." In: *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*. WICSA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 4–. ISBN: 0-7695-2744-2. DOI: http://dx.doi.org/10.1109/WICSA.2007.47. URL: http://dx.doi.org/10.1109/WICSA.2007.47.

[JB05] Anton Jansen and Jan Bosch. "Software Architecture as a Set of Architectural Design Decisions." In: *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. WICSA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 109–120. ISBN: 0-7695-2548-2.

[JZ14] Muhammad Atif Javed and Uwe Zdun. "The Supportive Effect of Traceability Links in Architecture-Level Software Understanding: Two Controlled Experiments." In: *2014 IEEE/IFIP Conference on Software Architecture, WICSA 2014, Sydney, Australia, April 7-11, 2014*. 2014, pp. 215–224. DOI: 10.1109/WICSA.2014.43.

[KA08] Ahmad Waqas Kamal and Paris Avgeriou. "Modeling Architectural Patterns' Behavior Using Architectural Primitives." In: *Proceedings of the 2nd European conference on Software Architecture*. ECSA '08. Paphos, Cyprus: Springer-Verlag, 2008, pp. 164–179. ISBN: 978-3-540-88029-5. DOI: 10.1007/978-3-540-88030-1_13.

[Kac+06a] O. Kaczor, Y. Guéhéneuc, and S. Hamel. "Efficient Identification of Design Patterns with Bit-vector Algorithm." In: *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. 2006, 10 pp.–184. DOI: 10.1109/CSMR.2006.25.

[Kac+06b] Olivier Kaczor, Yann Gaël Guéhéneuc, and Sylvie Hamel. "Efficient identification of design patterns with bit-vector algorithm." In: *Proc. Eur. Conf. Softw. Maint. Reengineering, CSMR* [2006], pp. 175–184. ISSN: 15345351. DOI: 10.1109/CSMR.2006.25.

[Kan+08] Ananya Kanjilal, S. Sengupta, and S. Bhattacharya. "CAG: A Component Architecture Graph." In: *TENCON, IEEE Region 10 International Conference*. 2008. DOI: 10.1109/TENCON.2008.4766419.

[Kel+99] Rudolf K. Keller, 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 '99. Los Angeles, California, USA: ACM, 1999, pp. 226–235. ISBN: 1-58113-074-0. DOI: 10.1145/302405.302622.

[Kit+02] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. "Preliminary Guidelines for Empirical Researchin Software Engineering." In: *IEEE Trans. Softw. Eng.* 28.8 [Aug. 2002], pp. 721–734. DOI: 10.1109/TSE.2002.1027796.

[Kle99] Jon M. Kleinberg. "Authoritative sources in a hyperlinked environment." In: *J. ACM* 46 [5 1999], pp. 604–632. ISSN: 0004-5411. DOI: http://doi.acm.org/10.1145/324133.324140. URL: http://doi.acm.org/10.1145/324133.324140.

[Kno+06] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. "Static Evaluation of Software Architectures." In: *Software Maintenance and Reengineering, European Conference on* [2006], pp. 279–294.

[Kon+13] Marco Konersmann, Zoya Durdik, Michael Goedicke, and Ralf H Reussner. "Towards Architecture-centric Evolution of Long-living Systems (the ADVERT Approach)." In: *Proc. 9th Int. ACM Sigsoft Conf. Qual. Softw. Archit.* 2013, pp. 163–168. ISBN: 978-1-4503-2126-6. DOI: 10.1145/2465478.2465496. URL: http://doi.acm.org/10.1145/2465478.2465496.

[Kos03] Rainer Koschke. "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey." In: *Journal of Software Maintenance* 15.2 [Mar. 2003], pp. 87–109. ISSN: 1040-550X. DOI: 10.1002/smr.270. URL: http://dx.doi.org/10.1002/smr.270.

[KP] Mark Kofman and Erik Perjons. *MetaDiff - a Model Comparison Framework*. metadiff.sourceforge.net/docs/metadiff.pdf.

[KP96] Christian Krämer and Lutz Prechelt. "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software." In: *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*. WCRE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 208–216. ISBN: 0-8186-7674-4.

[Kru+06] Philippe Kruchten, Patricia Lago, and Hans van Vliet. "Building Up and Reasoning About Architectural Knowledge." In: *Quality of Software Architectures*. Ed. by Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner. Vol. 4214. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 43–58.

[Kru95] Philippe Kruchten. "The 4+1 View Model of Architecture." In: *IEEE Softw.* 12.6 [Nov. 1995], pp. 42–50. ISSN: 0740-7459. DOI: 10.1109/52.469759. URL: http://dx.doi.org/10.1109/52.469759.

[KZ10a] Patrick Könemann and Olaf Zimmermann. "Linking design decisions to design models in model-based software development." In: *Proceedings of the 4th European conference on Software architecture*. ECSA'10. Copenhagen, Denmark: Springer-Verlag, 2010, pp. 246–262. ISBN: 3-642-15113-2, 978-3-642-15113-2. URL: http://dl.acm.org/citation.cfm?id=1887899.1887920.

[KZ10b] Patrick Könemann and Olaf Zimmermann. "Linking design decisions to design models in model-based software development." In: *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 6285 LNCS [2010], pp. 246–262. ISSN: 03029743. DOI: 10.1007/978-3-642-15114-9_19.

[Leh80] M.M. Lehman. "Programs, life cycles, and laws of software evolution." In: *Proc. IEEE* 68.9 [1980], pp. 1060–1076. ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1456074.

[Leh89] M. M. Lehman. "Uncertainty in Computer Application and Its Control Through the Engineering of Software." In: *Journal of Software Maintenance* 1.1 [Sept. 1989], pp. 3–27.

[Leh96] Meir M Lehman. "Laws of software evolution revisited." In: *5th Eur. Work. Softw. Process Technol.* Nancy: Springer Berlin Heidelberg, 1996, pp. 108–124. DOI: 10.1007/BFb0017737.

[Li+13] Zengyang Li, Peng Liang, and Paris Avgeriou. "Application of knowledge-based approaches in software architecture: A systematic mapping study." In: *Inf. Softw. Technol.* 55.5 [May 2013], pp. 777–794. ISSN: 09505849. DOI: 10.1016/j.infsof.2012.11.005. URL: http://linkinghub.elsevier.com/retrieve/pii/S0950584912002315.

[LM10] Jeff Linwood and Dave Minter. *Beginning Hibernate, Second Edition*. 2nd. Berkely, CA, USA: Apress, 2010. ISBN: 1430228504, 9781430228509.

[LN95] Danny B. Lange and Yuichi Nakamura. "Interactive Visualization of Design Patterns Can Help in Framework Understanding." In: *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*. OOPSLA '95. Austin, Texas, USA: ACM, 1995, pp. 342–357. ISBN: 0-89791-703-0. DOI: 10.1145/217838.217874.

[Luc+07] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. "Recovering Traceability Links in Software Artifact Management Systems using Information Retrieval Methods." In: vol. 16. New York, NY, USA: ACM, 2007. DOI: 10.1145/1276933.1276934.

[Lun+06] Mircea Lungu, Michele Lanza, and Tudor Girba. "Package Patterns for Visual Architecture Recovery." In: *Proceedings of the Conference on Software Maintenance and Reengineering*. CSMR '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 185–196.

[LW07] Kung-Kiu Lau and Zheng Wang. "Software Component Models." In: *IEEE Trans. Softw. Eng.* 33.10 [Oct. 2007], pp. 709–724. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70726. URL: http://dx.doi.org/10.1109/TSE.2007.70726.

[Lyt+13a] Ioanna Lytra, Holger Eichelberger, Huy Tran, Georg Leyh, Klaus Schmid, and Uwe Zdun. "On the Interdependence and Integration of Variability and Architectural Decisions." In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: ACM, 2013, 19:1–19:8. ISBN: 978-1-4503-2556-1. DOI: 10.1145/2556624.2556634. URL: http://doi.acm.org/10.1145/2556624.2556634.

[Lyt+13b] Ioanna Lytra, Huy Tran, and Uwe Zdun. "Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations." In: *7th Eur. Conf. Softw. Arch. (ECSA 2013)*. Springer Berlin Heidelberg, 2013, pp. 224–239. DOI: 10.1007/978-3-642-39031-9_20.

[Ma+06] Yutao Ma, Keqing He, Dehui Du, Jing Liu, and Yulan Yan. "A Complexity Metrics Set for Large-Scale Object-Oriented Software Systems." In: *Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*. CIT '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 189–. ISBN: 0-7695-2687-X. DOI: 10.1109/CIT.2006.3. URL: http://dx.doi.org/10.1109/CIT.2006.3.

[Mac77] Alan K. Mackworth. "Consistency in Networks of Relations." In: *Artificial Intelligence* 8.1 [1977], pp. 99–118. ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/0004-3702(77)90007-8.

[Mar03] Robert C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003. URL: http://dl.acm.org/citation.cfm?id=515230.

[MB07] Onaiza Maqbool and Haroon Babri. "Hierarchical Clustering for Software Architecture Recovery." In: *IEEE Trans. Softw. Eng.* 33 [11 2007], pp. 759–780. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70732. URL: http://dl.acm.org/citation.cfm?id=1314036.1314083.

[MC08] Parastoo Mohagheghi and Reidar Conradi. "An empirical investigation of software reuse benefits in a large telecom product." In: *ACM Trans. Softw. Eng. Methodol.* 17.3 [June 2008], 13:1–13:31. ISSN: 1049-331X. DOI: 10.1145/1363102.1363104. URL: http://doi.acm.org/10.1145/1363102.1363104.

[McV+11] Andrew McVeigh, Jeff Kramer, and Jeff Magee. "Evolve: tool support for architecture evolution." In: *2011 33rd Int. Conf. Softw. Eng.* [2011], pp. 1040–1042. ISSN: 0270-5257. DOI: 10.1145/1985793.1985990.

[MD00] Yashwant K. Malaiya and Jason Denton. "Module Size Distribution and Defect Density." In: *Proceedings of the 11th International Symposium on Software Reliability Engineering*. ISSRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 62–. ISBN: 0-7695-0807-3.

[ME12] Patrick Mäder and Alexander Egyed. "Assessing the effect of requirements traceability for software maintenance." In: *ICSM*. IEEE Computer Society, 2012, pp. 171–180. ISBN: 978-1-4673-2313-0.

[Mec04] Robert Mecklenburg. *Managing Projects with GNU Make (Nutshell Handbooks)*. O'Reilly Media, Inc., 2004. ISBN: 0596006101.

[Med+02] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. "Modeling software architectures in the Unified Modeling Language." In: *ACM Trans. Softw. Eng. Methodol.* 11.1 [2002], pp. 2–57.

[Med+03] Nenad Medvidovic, Alexander Egyed, and Paul Grünbacher. "Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery." In: *Proc. of the 2nd International Software Requirements to Architectures Workshop(STRAW)*. Portland, Oregon, 2003, pp. 61–68.

[Men+02] Kim Mens, Tom Mens, and Michel Wermelinger. "Maintaining software through intentional source-code views." In: *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. SEKE '02. Ischia, Italy: ACM, 2002, pp. 289–296. ISBN: 1-58113-556-4. DOI: 10.1145/568760.568812. URL: http://doi.acm.org/10.1145/568760.568812.

[Mey06] Eric A Meyer. *CSS: The Definitive Guide*. O'Reilly Media, Inc., 2006. ISBN: 0596527330.

[Mik98] T. Mikkonen. "Formalizing Design Patterns." In: *Proceedings of the 20th international conference on Software engineering*. Kyoto, Japan: IEEE Computer Society, 1998, pp. 115–124.

[Mil+10] James A. Miller, Remo Ferrari, and Nazim H. Madhavji. "An exploratory study of architectural effects on requirements decisions." In: *J. Syst. Softw.* 83.12 [Dec. 2010], pp. 2441–2455. ISSN: 0164-1212.

[MM01] Jonathan I. Maletic and Andrian Marcus. "Supporting Program Comprehension Using Semantic and Structural Information." In: *Proceedings of the 23rd International Conference on Software Engineering*. ICSE '01. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 103–112. ISBN: 0-7695-1050-7.

[MM03] N. R. Mehta and N. Medvidovic. "Composing Architectural Styles From Architectural Primitives." In: *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*. Helsinki, Finland: ACM Press, 2003, pp. 347–350.

[Moh+04] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. "An Empirical Study of Software Reuse vs. Defect-Density and Stability." In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 282–292. ISBN: 0-7695-2163-0.

[Moo] Moodle. *Moodle*. URL: http://docs.moodle.org/dev/Roadmap [visited on 04/15/2014].

[Moo+08] Oege Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. "Generative and Transformational Techniques in Software Engineering II." In: ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. .QL: Object-Oriented Queries Made Easy, pp. 78–133. ISBN: 978-3-540-88642-6. DOI: 10.1007/978-3-540-88643-3_3. URL: http://dx.doi.org/10.1007/978-3-540-88643-3_3.

[MR13] Héctor J. Macho and Gregorio Robles. "Preliminary lessons from a software evolution analysis of Moodle." In: *First Int. Conf. Technol. Ecosyst. Enhancing Multicult. - TEEM '13*. New York, New York, USA: ACM Press, 2013, pp. 157–161. ISBN: 9781450323451. DOI: 10.1145/2536536.2536560. URL: http://dl.acm.org/citation.cfm?doid=2536536.2536560.

[MR47] H. B. Mann and Whitney D. R. "On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other." In: *Annals of Mathematical Statistics* 18.1 [1947], pp. 50–60.

[MS95] Salvatore T. March and Gerald F. Smith. "Design and Natural Science Research on Information Technology." In: *Decis. Support Syst.* 15.4 [Dec. 1995], pp. 251–266. ISSN: 0167-9236. DOI: 10.1016/0167-9236(94)00041-2. URL: http://dx.doi.org/10.1016/0167-9236(94)00041-2.

[MS97] Rohit Mahajan and Ben Shneiderman. "Visual and Textual Consistency Checking Tools for Graphical User Interfaces." In: *IEEE Trans. Softw. Eng.* 23.11 [Nov. 1997], pp. 722–735. ISSN: 0098-5589. DOI: 10.1109/32.637386.

[MT00] Nenad Medvidovic and Richard N. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages." In: *IEEE Trans. Softw. Eng.* 26.1 [Jan. 2000], pp. 70–93. ISSN: 0098-5589. DOI: 10.1109/32.825767. URL: http://dx.doi.org/10.1109/32.825767.

[Mur+95a] Gail C. Murphy, David Notkin, and Kevin Sullivan. "Software reflexion models." In: *ACM SIGSOFT Softw. Eng. Notes* 20.4 [1995], pp. 18–28. ISSN: 01635948. DOI: 10.1145/222132.222136.

[Mur+95b] Gail C. Murphy, David Notkin, and Kevin Sullivan. "Software reflexion models: bridging the gap between source and high-level models." In: *SIGSOFT Softw. Eng. Notes* 20 [4 1995], pp. 18–28. ISSN: 0163-5948. DOI: http://doi.acm.org/10.1145/222132.222136. URL: http://doi.acm.org/10.1145/222132.222136.

[Nak+08] Elisa Yumi Nakagawa, Elaine Parros Machado de Sousa, Kiyoshi de Brito Murata, Gabriel de Faria Andery, Leonardo Bitencourt Morelli, and José Carlos Maldonado. "Software Architecture Relevance in Open Source Software Evolution: A Case Study." In: *32nd Annu. IEEE Int. Comput. Softw. Appl. Conf. (COMPSAC 2008)* [2008], pp. 1234–1239. DOI: 10.1109/COMPSAC.2008.171. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4591757.

[Nav+13] Elena Navarro, Carlos E. Cuesta, Dewayne E. Perry, and Pascual González. "Antipatterns for Architectural Knowledge Management." In: *Int. J. Inf. Technol. Decis. Mak.* 12.3 [2013], pp. 547–589. DOI: http://dx.doi.org/10.1142/S0219622013500211. URL: http://www.dsi.uclm.es/personal/ElenaNavarro/research_publications_International_Journals.html.

[NC08] Elena Navarro and Carlos E Cuesta. "Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach." In: *Second Eur. Conf. Softw. Arch. (ECSA 2008)*. Springer, 2008, pp. 114–130. DOI: 10.1007/978-3-540-88030-1_10. URL: http://www.springerlink.com/content/v116347218g6816g/http://dl.acm.org/citation.cfm?id=1434560.

[NC09] Ariadi Nugroho and Michel R. Chaudron. "Evaluating the Impact of UML Modeling on Software Quality: An Industrial Case Study." In: *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*. MODELS '09. Denver, CO: Springer-Verlag, 2009, pp. 181–195. ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_14. URL: http://dx.doi.org/10.1007/978-3-642-04425-0_14.

[Nea+13] Iulian Neamtiu, Guowu Xie, and Jianbo Chen. "Towards a better understanding of software evolution: an empirical study on open-source software." In: *J. Softw. Evol. Process* 25.3 [2013], pp. 193–218. ISSN: 20477473. DOI: 10.1002/smr.564. URL: http://doi.wiley.com/10.1002/smr.564.

[NT10] Joost Noppen and Dalila Tamzalit. "ETAK: Tailoring Architectural Evolution by (re-)using Architectural Knowledge." In: *ICSE Work. Shar. Reusing Archit. Knowl. (SHARK '10)*. New York, New York, USA: ACM Press, 2010, pp. 21–28. ISBN: 9781605589671. DOI: 10.1145/1833335.1833339. URL: http://portal.acm.org/citation.cfm?doid=1833335.1833339.

[OD04] Mari Carmen Otero and JosÃ© Javier Dolado. "Evaluation of the comprehension of the dynamic modeling in UML." In: *Information and Software Technology* 46.1 [2004], pp. 35 –53. ISSN: 0950-5849. DOI: 10.1016/S0950-5849(03)00108-3.

[Ozk+07] Ipek Ozkaya, Rick Kazman, and Mark Klein. "Quality-attribute based economic valuation of architectural patterns." In: *First Int. Work. Econ. Softw. Comput. ESC'07*. 2007. ISBN: 0769529550. DOI: 10.1109/ESC.2007.8.

[Ozk+10a] Ipek Ozkaya, Peter Wallin, and Jakob Axelsson. "Architecture knowledge management during system evolution." In: *2010 ICSE Work. Shar. Reusing Arch. Knowl. (SHARK '10)*. Helsinki: ACM Press, 2010, pp. 52–59. ISBN: 9781605589671. DOI: 10.1145/1833335.1833343. URL: http://portal.acm.org/citation.cfm?id=1833335.1833343.

[Ozk+10b] Ipek Ozkaya, Peter Wallin, and Jakob Axelsson. "Architecture Knowledge Management during System Evolution – Observations from Practitioners." In: *SHARK'10* [2010], pp. 52–59. ISSN: 02705257. DOI: 10.1145/1833335.1833343.

[Paa+00] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, and A. Inkeri Verkamo. "Software Metrics by Architectural Pattern Mining." In: *in Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress*. 2000, pp. 325–332.

[Pah+09] Claus Pahl, Simon Giesecke, and Wilhelm Hasselbring. "Ontology-based modelling of architectural styles." In: *Inf. Softw. Technol.* 51.12 [Dec. 2009], pp. 1739–1749. ISSN: 09505849. DOI: 10.1016/j.infsof.2009.06.001. URL: http://linkinghub.elsevier.com/retrieve/pii/S0950584909000846.

[Pal+12] F. Palma, H. Farzin, Y. Gueheneuc, and N. Moha. "Recommendation System for Design Patterns in Software Development: An DPR Overview." In: *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*. June 2012, pp. 1–5. DOI: 10.1109/RSSE.2012.6233399.

[Par94] David Lorge Parnas. "Software Aging." In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 279–287. ISBN: 0-8186-5855-X. URL: http://dl.acm.org/citation.cfm?id=257734.257788.

[Pas+10] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor das Chagas Mendonca. "Static Architecture-Conformance Checking: An Illustrative Overview." In: *IEEE Softw.* 27 [5 2010], pp. 82–89. ISSN: 0740-7459. DOI: 10.1109/MS.2009.117.

[PG02] Martin Pinzger and Harald Gall. "Pattern-Supported Architecture Recovery." In: *Proceedings of the 10th International Workshop on Program Comprehension*. IWPC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 53–62.

[Phi+03] Ilka Philippow, Detlef Streitferdt, and Matthias Riebisch. "Design Pattern Recovery in Architectures for Supporting Product Line Development and Application." In: *Modelling Variability for Object-Oriented Product Lines*. BookOnDemand Publ. Co, 2003, pp. 42–57.

[Pin+05] Martin Pinzger, Harald Gall, and Michael Fischer. "Towards an Integrated View on Architecture and its Evolution." In: *Electronic Notes in Theoretical Computer Science* 127.3 [2005], pp. 183 –196.

[Pur+01] Helen C. Purchase, Linda Colpoys, Matthew McGill, David Carrington, and Carol Britton. "UML class diagram syntax: an empirical study of comprehension." In: *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation - Volume 9*. APVis '01. Sydney, Australia: Australian Computer Society, Inc., 2001, pp. 113–120. ISBN: 0-909925-87-9.

[PW92a] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the study of software architecture." In: *SIGSOFT Softw. Eng. Notes* 17.4 [Oct. 1992], pp. 40–52. ISSN: 0163-5948.

[PW92b] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture." In: *ACM Softw. Eng. Notes* 17.4 [1992], pp. 40–52. DOI: 10.1145/141874.141884.

[Qin+05] Li Qingshan, Chu Hua, Hu Shengming, Chen Ping, and Zhao Yun. "Architecture Recovery and Abstraction from the Perspective of Processes." In: *Proceedings of the 12th Working Conference on Reverse Engineering*. WCRE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 57–66.

[Qus+11] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley. "SCOTCH: Test-to-code traceability using slicing and conceptual coupling." In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. 2011, pp. 63–72. DOI: 10.1109/ICSM.2011.6080773.

[Qus+12] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. "Evaluating test-to-code traceability recovery methods through controlled experiments." In: *Journal of Software: Evolution and Process* [2012], n/a–n/a. ISSN: 2047-7481. DOI: 10.1002/smr.1573. URL: http://dx.doi.org/10.1002/smr.1573.

[RD99] Tamar Richner and Stéphane Ducasse. "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information." In: *Proceedings of the IEEE International Conference on Software Maintenance*. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 13–.

[Ric+07] F. Ricca, M. Di Penta, Marco Torchiano, P. Tonella, and M. Ceccato. "The Role of Experience and Ability in Comprehension Tasks Supported by UML Stereotypes." In: *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. 2007, pp. 375–384. DOI: 10.1109/ICSE.2007.86.

[RM11] Ghulam Rasool and Patrick Mader. "Flexible Design Pattern Detection Based on Feature Types." In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 243–252. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100060.

[Ros+11] Jacek Rosik, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar, and Dave Connolly. "Assessing architectural drift in commercial software development: a case study." In: *Softw. Pract. Exp.* 41.1 [Jan. 2011], pp. 63–86. ISSN: 00380644. DOI: 10.1002/spe.999. URL: http://doi.wiley.com/10.1002/spe.999.

[Ros+13] Dominik Rost, Matthias Naab, Crescencio Lima, and Christina von Flach Garcia Chavez. "Software Architecture Documentation for Developers: A Survey." English. In: *Software Architecture*. Ed. by Khalil Drira. Vol. 7957. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 72–88. ISBN: 978-3-642-39030-2. DOI: 10.1007/978-3-642-39031-9_7. URL: http://dx.doi.org/10.1007/978-3-642-39031-9_7.

[RR02] Claudio Riva and Jordi Vidal Rodriguez. "Combining Static and Dynamic Views for Architecture Reconstruction." In: *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. CSMR '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 47–.

[RW05] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005. ISBN: 0321112296.

[San+05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. "Using dependency models to manage complex software architecture." In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 167–176. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094824. URL: http://doi.acm.org/10.1145/1094811.1094824.

[Sar01] K. Sartipi. "A software evaluation model using component association views." In: *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*. 2001, pp. 259–268. DOI: 10.1109/WPC.2001.921736.

[Sar03] Kamran Sartipi. "Software Architecture Recovery based on Pattern Matching." In: *Proceedings of the International Conference on Software Maintenance*. ICSM '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 293–. ISBN: 0-7695-1905-9.

[SB03] Douglas C. Schmidt and Frank Buschmann. "Patterns, Frameworks, and Middleware: Their Synergistic Relationships." In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Portland, Oregon: IEEE Computer Society, 2003, pp. 694–704. ISBN: 0-7695-1877-X. URL: http://dl.acm.org/citation.cfm?id=776816.776917.

[Sca+10] Giuseppe Scanniello, Anna D'Amico, Carmela D'Amico, and Teodora D'Amico. "An approach for architectural layer recovery." In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: ACM, 2010, pp. 2198–2202. ISBN: 978-1-60558-639-7. DOI: http://doi.acm.org/10.1145/1774088.1774551. URL: http://doi.acm.org/10.1145/1774088.1774551.

[Sch+00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. 2nd. New York, NY, USA: John Wiley & Sons, Inc., 2000. ISBN: 0471606952, 9780471606956.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Addison-Wesley, 1996.

[SG98] Jochen Seemann and Jürgen Wolff von Gudenberg. "Pattern-based Design Recovery of Java Software." In: *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '98/FSE-6. Lake Buena Vista, Florida, USA: ACM, Nov. 1998, pp. 10–16. ISBN: 1-58113-108-9.

[Sha+09] Arun Sharma, P. S. Grover, and Rajesh Kumar. "Dependency analysis for component-based software systems." In: *SIGSOFT Softw. Eng. Notes* 34.4 [July 2009], pp. 1–6. ISSN: 0163-5948. DOI: 10.1145/1543405.1543424. URL: http://doi.acm.org/10.1145/1543405.1543424.

[Shu+96] F.J. Shull, W. Melo, and V.R. Basili. *An Inductive Method for Discovering Design Patterns from Object-oriented Software Systems*. Computer science technical report series. University of Maryland, 1996.

[SOO] SOOMLA. *Open source framework version 3.1*. URL: http://soom.la/ [visited on 04/24/2015].

[Soy11] Soyatec. *eUML2*. http://www.soyatec.com/euml2/. 2011.

[Spi11] Diomidis Spinellis. *UMLGraph*. http://www.umlgraph.org. 2011.

[Sta+06] Miroslaw Staron, Ludwik Kuzniarz, and Claes Wohlin. "Empirical assessment of using stereotypes to improve comprehension of UML models: A set of experiments." In: *J. Syst. Softw.* 79.5 [May 2006], pp. 727–742. ISSN: 0164-1212. DOI: 10.1016/j.jss.2005.09.014. URL: http://dx.doi.org/10.1016/j.jss.2005.09.014.

[Ste+14a] Srdjan Stevanetic, Muhammad Atif Javed, and Uwe Zdun. "Empirical Evaluation of the Understandability of Architectural Component Diagrams." In: *Proceedings of the WICSA 2014 Companion Volume*. WICSA '14 Companion. Sydney, Australia: ACM, 2014, 4:1–4:8. ISBN: 978-1-4503-2523-3. DOI: 10.1145/2578128.2578230.

[Ste+14b] Srdjan Stevanetic, Thomas Haitzer, and Uwe Zdun. "Supporting Software Evolution by Integrating DSL-based Architectural Abstraction and Understandability Related Metrics." In: *Proceedings of the 2014 European Conference on Software Architecture Workshops*. ECSAW '14. Vienna, Austria: ACM, 2014, 19:1–19:8. ISBN: 978-1-4503-2778-7. DOI: 10.1145/2642803.2642822.

[Ste+15] Srdjan Stevanetic, Muhammad Atif Javed, and Uwe Zdun. "The Impact of Hierarchies on the Architecture-level Software Understandability - A Controlled Experiment." In: *24th Australasian Software Engineering Conference*. Sept. 2015. URL: http://eprints.cs.univie.ac.at/4423/.

[Ste46] S. Stevens. "On the theory of scales of measurement." In: *Science* 103.2684 [1946], 677—680.

[SW05] D. Sun and K. Wong. "On evaluating the layout of UML class diagrams for program comprehension." In: *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. 2005, pp. 317–326. DOI: 10.1109/WPC.2005.26.

[SW08] K Stencel and P Wegrzynowicz. "Detection of Diverse Design Pattern Variants." In: *Softw. Eng. Conf. 2008. APSEC '08. 15th Asia-Pacific*. 2008, pp. 25–32. DOI: 10.1109/APSEC.2008.67.

[SW65] S. S. Shapiro and M. B. Wilk. "An analysis of variance test for normality (complete samples)." In: *Biometrika* 3.52 [1965].

[SZ05] George Spanoudakis and Andrea Zisman. "Software Traceability: A Roadmap." In: *Handbook of Software Engineering and Knowledge Engineering*. Ed. by S. K. Chang. Vol. 3. World Scientific Publishing Co., 2005.

[SZ14] Srdjan Stevanetic and Uwe Zdun. "Exploring the Relationships Between the Understandability of Components in Architectural Component Models and Component Level Metrics." In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. London, England, United Kingdom: ACM, 2014, 32:1–32:10. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601264. URL: http://doi.acm.org/10.1145/2601248.2601264.

[TA01] Paolo Tonella and Giuliano Antoniol. "Inference of object-oriented design patterns." In: *Journal of Software Maintenance* 13.5 [2001], pp. 309–330.

[TA05] Jeff Tyree and Art Akerman. "Architecture Decisions: Demystifying Architecture." In: *IEEE Software* 22 [2005], pp. 19–27.

[TA99] P. Tonella and G. Antoniol. "Object Oriented Design Pattern Inference." In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*. 1999, pp. 230–238. DOI: 10.1109/ICSM.1999.792619.

[Tam+06] Dalila Tamzalit, Mourad C. Oussalah, Olivier Le Goaer, and Abdelhak-Djamel Seriai. "Updating software architectures : A style-based approach." In: *Int. Conf. Softw. Eng. Res. Pract. (SERP 2006)*. Las Vegas: CSREA Press, 2006, pp. 313–318.

[Tan+06] Antony Tang, Ali Babar Muhammad, Ian Gorton, and Jun Han. "A survey of architecture design rationale." In: *J. Syst. Softw.* 79.12 [2006], pp. 1792–1804. ISSN: 01641212. DOI: 10.1016/j.jss.2006.04.029. URL: http://dblp.uni-trier.de/db/journals/jss/jss79.html\#TangBGH06.

[Tan+07] Antony Tang, Ann E Nicholson, Yan Jin, and Jun Han. "Using Bayesian belief networks for change impact analysis in architecture design." In: *J. Syst. Softw.* 80.1 [2007], pp. 127–148. ISSN: 01641212. DOI: 10.1016/j.jss.2006.04.004. URL: http://dblp.uni-trier.de/db/journals/jss/jss80.html\#TangNJH07.

[Tar07] Pentti Tarvainen. "Adaptability Evaluation of Software Architectures: A Case Study." In: *31st Annu. Int. Comput. Softw. Appl. Conf. - Vol. 2 - (COMPSAC 2007)*. IEEE, July 2007, pp. 579–586. ISBN: 0-7695-2870-8. DOI: 10.1109/COMPSAC.2007.240. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4291181.

[Tay+10] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture - Foundations, Theory, and Practice*. Wiley, 2010, pp. I–XXIV, 1–712. ISBN: 978-0-470-16774-8.

[Tek+07] Bedir Tekinerdogan, Christian Hofmann, and Mehmet Aksit. "Modeling Traceability of Concerns for Synchronizing Architectural Views." In: *Journal of Object Technology* 6.7 [2007], pp. 7–25. URL: http://doc.utwente.nl/60276/.

[Ter+13] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. "A recommendation system for repairing violations detected by static architecture conformance checking." In: *Softw. Pract. Exp.* [Sept. 2013], n/a–n/a. ISSN: 00380644. DOI: 10.1002/spe.2228. URL: http://doi.wiley.com/10.1002/spe.2228.

[TH03] Scott Tilley and Shihong Huang. "A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding." In: *Proceedings of the 21st annual international conference on Documentation*. SIGDOC '03. San Francisco, CA, USA: ACM, 2003, pp. 184–191. ISBN: 1-58113-696-X. DOI: 10.1145/944868.944908. URL: http://doi.acm.org/10.1145/944868.944908.

[Tib+10] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. "A family of languages for architecture constraint specification." In: *J. Syst. Softw.* 83.5 [May 2010], pp. 815–831. ISSN: 01641212. DOI: 10.1016/j.jss.2009.11.736. URL: http://linkinghub.elsevier.com/retrieve/pii/S016412120900315X.

[Tic+00] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. "FAMIX and XMI." In: *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. WCRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 296–.

[Tof+14] Dan Tofan, Matthias Galster, Paris Avgeriou, and Wes Schuitema. "Past and Future of Software Architectural Decisions – a Systematic Mapping Study." In: *Inf. Softw. Technol.* [Mar. 2014]. ISSN: 09505849. DOI: 10.1016/j.infsof.2014.03.009. URL: http://linkinghub.elsevier.com/retrieve/pii/S0950584914000706.

[Tor04] Marco Torchiano. "Empirical assessment of UML static object diagrams." In: *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*. 2004, pp. 226–230. DOI: 10.1109/WPC.2004.1311064.

[Tsa+06] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis. "Design Pattern Detection Using Similarity Scoring." In: *Software Engineering, IEEE Transactions on* 32.11 [2006], pp. 896–909. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.112.

[UM12] Raoul-Gabriel Urma and Alan Mycroft. "Programming language evolution via source code query languages." In: *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*. PLATEAU '12. Tucson, Arizona, USA: ACM, 2012, pp. 35–38. ISBN: 978-1-4503-1631-6. DOI: 10.1145/2414721.2414728. URL: http://doi.acm.org/10.1145/2414721.2414728.

[Ump+06] David A. Umphress, T. Dean Hendrix, James H. Cross II, and Saeed Maghsoodloo. "Software visualizations for improving and measuring the comprehensibility of source code." In: *Science of Computer Programming* 60.2 [2006], pp. 121 –133. ISSN: 0167-6423. DOI: 10.1016/j.scico.2005.10.001.

[VK07] Vijay K Vaishnavi and William Kuechler. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Auerbach, 2007.

[Was+09] Hironori Washizaki, Kazuhiro Fukaya, Atsuto Kubo, and Yoshiaki Fukazawa. "Detecting Design Patterns Using Source Code of Before Applying Design Patterns." In: *Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science*. ICIS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 933–938. ISBN: 978-0-7695-3641-5. DOI: 10.1109/ICIS.2009.209.

[Wen+01] Lothar Wendehals, Jörg Niere, and Jörg P. Wadsack. *Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic*. Tech. rep. University of Paderborn, 2001.

[Wen03] L. Wendehals. "Improving Design Pattern Instance Recognition by Dynamic Analysis." In: *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*. 2003, pp. 29–32.

[Whe09] David Wheeler. *SLOCcount*. 2009. URL: http://www.dwheeler.com/sloccount/.

[Woh+03] Claes Wohlin, Martin Höst, and Kennet Henningsson. "Empirical Research Methods in Software Engineering." In: *Empirical Methods and Studies in Software Engineering*. Vol. 2765. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 7–23. DOI: 10.1007/978-3-540-45143-3_2.

[Woh+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. 1st ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-29043-5. URL: http://link.springer.com/10.1007/978-3-642-29044-2.

[WP10] Stefan Winkler and Jens Pilgrim. "A survey of traceability in requirements engineering and model-driven development." In: *Softw. Syst. Model.* 9.4 [Sept. 2010], pp. 529–565. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0145-0. URL: http://dx.doi.org/10.1007/s10270-009-0145-0.

[Wuy98] R. Wuyts. "Declarative Reasoning about the Structure of Object-Oriented Systems." In: *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*. 1998, pp. 112–124. DOI: 10.1109/TOOLS.1998.711007.

[Xu+13] Chang Xu, YePang Liu, S.C. Cheung, Chun Cao, and Jian Lv. "Towards context consistency by concurrent checking for Internetware applications." English. In: *Science China Information Sciences* 56.8 [2013], pp. 1–20. ISSN: 1674-733X. DOI: 10.1007/s11432-013-4907-5.

[Yan+04] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. "DiscoTect: A System for Discovering Architectures from Running Systems." In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 470–479.

[ZA05] Uwe Zdun and Paris Avgeriou. "Modeling architectural patterns using architectural primitives." In: *ACM SIGPLAN Not.* 40.10 [Oct. 2005], p. 133. ISSN: 03621340. DOI: 10.1145/1103845.1094822. URL: http://portal.acm.org/citation.cfm?doid=1103845.1094822.

[ZA08] Uwe Zdun and Paris Avgeriou. "A catalog of architectural primitives for modeling architectural patterns." In: *Inf. Softw. Technol.* 50.9-10 [Aug. 2008], pp. 1003–1034. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2007.09.003. URL: http://dx.doi.org/10.1016/j.infsof.2007.09.003.

[Zac87] John A. Zachman. "A Framework for Information Systems Architecture." In: *IBM Syst. J.* 26.3 [Sept. 1987], pp. 276–292. ISSN: 0018-8670. DOI: 10.1147/sj.263.0276. URL: http://dx.doi.org/10.1147/sj.263.0276.

[Zdu+04] Uwe Zdun, Michael Kircher, and Markus Völter. "Remoting Patterns." In: *IEEE Internet Computing* 8.6 [2004], pp. 60–68.

[Zdu11] Uwe Zdun. *The Frag Language*. http://frag.sourceforge.net/. 2011.

[Zha+02] Jianjun Zhao, Hongji Yang, Liming Xiang, and Baowen Xu. "Change impact analysis to support architectural evolution." In: *Journal of Software Maintenance* 14.5 [Sept. 2002], pp. 317–333. ISSN: 1040-550X. DOI: 10.1002/smr.258. URL: http://dx.doi.org/10.1002/smr.258.

[Zim+07] Olaf Zimmermann, Thomas Gschwind, Jochen Küster, Frank Leymann, and Nelly Schuster. "Reusable architectural decision models for enterprise application development." In: *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 4880 LNCS [2007], pp. 15–32. ISSN: 03029743. DOI: 10.1007/978-3-540-77619-2_2.

[Zim+08] Olaf Zimmermann, Uwe Zdun, Thomas Gschwind, and Frank Leymann. "Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method." In: *7th IEEE/IFIP Work. Conf. Softw. Archit. WICSA 2008* [2008], pp. 157–166. DOI: 10.1109/WICSA.2008.19.

[Zim+09] Olaf Zimmermann, Jana Koehler, Frank Leymann, Ronny Polley, and Nelly Schuster. "Managing architectural decision models with dependency relations, integrity constraints, and production rules." In: *Journal of Systems and Software* 82.8 [2009], pp. 1249–1267.

[De +09] Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. "Assessing IR-based traceability recovery tools through controlled experiments." In: *Empir. Softw. Eng.* 14.1 [2009], pp. 57–92. ISSN: 13823256.

[Ecl11a] Eclipse Foundation. *EMF Compare*. http://www.eclipse.org/emf/compare/. 2011.

[Ecl11b] Eclipse Foundation. *Xtext*. http://www.eclipse.org/Xtext/. 2011.

[Le +08] Olivier Le Goaer, Dalila Tamzalit, Mourad Chabane Oussalah, and Abdelhak-Djamel Seriai. "Evolution styles to the rescue of architectural evolution knowledge." In: *3rd Int. Work. Shar. Reusing Archit. Knowl.* New York, New York, USA: ACM Press, 2008, pp. 31–36. ISBN: 9781605580388. DOI: 10.1145/1370062.1370071. URL: http://portal.acm.org/citation.cfm?doid=1370062.1370071.

[M. 11] M. Doliner, J. Erdfelt, J. Lewis, G. Lukasik, J. Mareš, and J. Thomerson. *Cobertura*. http://cobertura.sourceforge.net. 2011.

[Obj10] Object Management Group. *UML 2.3 Superstructure*. 2010. URL: http://www.omg.org/spec/UML/2.3.

[The11] The Freecol Team. *FreeCol*. http://freecol.org. 2011.

[Tra] Tracker Moodle. *Tracker Moodle*. URL: https://tracker.moodle.org/ [visited on 04/15/2014].