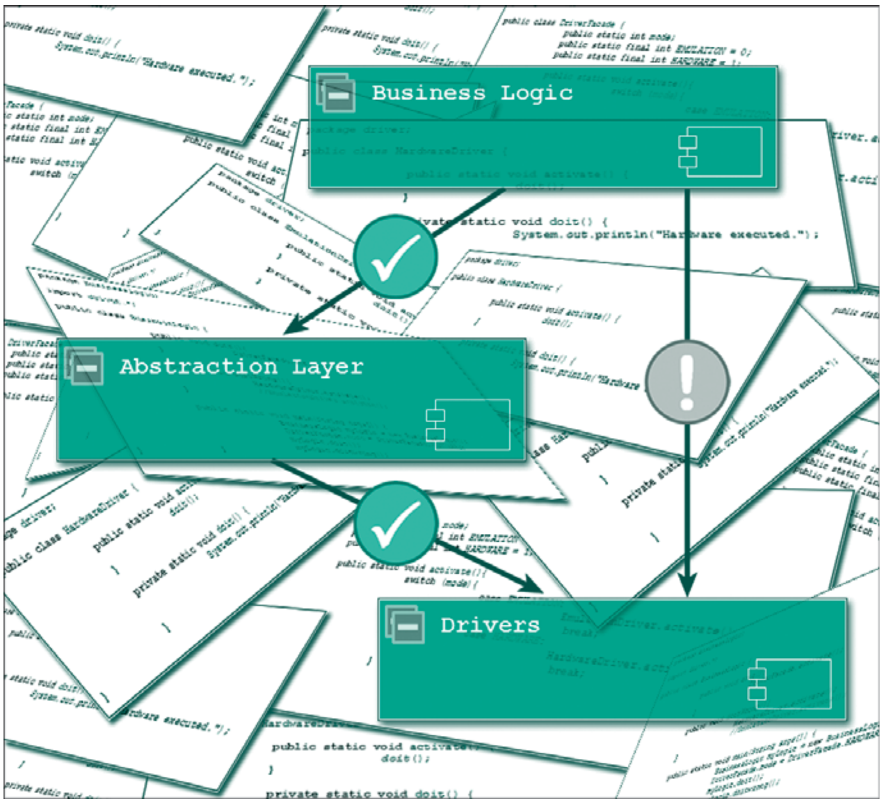
PhD Theses in Experimental Software Engineering

Vol. 35

Jens Knodel

Sustainable Structures in Software Implementations by Live Compliance Checking



FRAUNHOFER VERLAG

Zugl.: Kaiserslautern, Univ., Diss., 2010

Printing:
Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Printed on acid-free and chlorine-free bleached paper.

© by **Fraunhofer Verlag**, 2011
ISBN 978-3-8396-0233-1
Fraunhofer-Informationszentrum Raum und Bau IRB
Postfach 800469, 70504 Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon +49 711 970 - 25 00
Telefax +49 711 970 - 25 08
E-Mail verlag@fraunhofer.de
URL http://verlag.fraunhofer.de

Sustainable Structures in Software Implementations by Live Compliance Checking

Beim Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr. Ing.)

genehmigte Dissertation
von

Dipl.-Inf. Jens Knodel

Fraunhofer-Institut für Experimentelles Software Engineering
(Fraunhofer IESE)
Kaiserslautern

Berichterstatter:	Prof. Dr. Dr. h. c. Dieter Rombach Prof. Dr. Jürgen Ebert
Dekan:	Dr. habil. Bernd Schürmann
Tag der Wissenschaftlichen Aussprache:	12.05.2010

D 386

"*All the forces in the world are not so powerful as an idea whose time has come.*"
Victor Hugo (*1802 – †1885)

Abstract

Software architecture – besides other aspects – outlines the structure of software systems prescribing the intended decomposition into components and dependencies among them. Developers then translate the abstract building blocks of the system into source code. Architecture compliance captures the degree to which required or requested demands on the structure in the implementation of a software system have been met. Analyzing the actual decomposition reveals a dilemma of today’s software development organizations: Almost all implementations exhibit significant structural violations. We could observe this practical problem across various application domains such as embedded systems and information systems, as well as for academic systems.

We first investigated the impact of this lack of compliance on the development effort in three replications of a controlled experiment. We compared the effort for a sample system evolution task on two functionally equivalent implementation variants of the same system – one with significant structural violations and the other one realized in compliance with the architecture. The results showed that the effort required was, on average, more than double (204%) for the implementation with structural violations. The empirical results provide evidence that the evolution of a system becomes effort-intensive when lacking compliance, which imposes maintenance risks for the development organization. The architecture as a management vehicle for stakeholders turns out to be unreliable, delusive, and almost useless. To counteract these threats, the development organization would have to invest significant effort to reshape the system towards the decomposition as specified by the architecture.

The primary contribution of this thesis is live architecture compliance checking, which sustains the structure in software implementations. This new compliance checking technique verifies any source code modifications made. Any delta – source code locally modified by a developer – is first mined for relevant information and then checked for compliance. Developers receive live feedback on the compliance checking results and are immediately made aware of drifts between architecture and implementation. Live compliance checking supports distributed teams of developers – starting from day one of the development. The entire source code is constantly monitored, analyzed, and checked continuously. This live detection allows developers correcting violations promptly: It further acts as pro-active training for developers regarding the intended structural decomposition.

Live compliance checking detects architecture violations at the earliest point in time possible – right after their insertion. The fast response time for feedback enables developers to repair the structure with minimal effort and, thus, to sustain the intended decomposition over time.

The idea of live compliance checking has been realized as an extension of the Fraunhofer SAVE tool (Software Architecture Visualization and Evaluation). This variant – called SAVE LiFe (Live Feedback) – is a client-server-client system, which enables live feedback on compliance for distributed development teams. One central server performs compliance checking for the modification made by multiple distributed developers while the architects' client tracks the overall compliance status. The application of SAVE LiFe over a period of 35 days in an experiment provided empirical evidence for its usefulness. The developers supported by the live compliance checking caused about 60% less structural violations throughout than the developers of the control group. Based on this result, we can conclude that live compliance checking potentially leads to less rework and to effort savings due to the reduced number of structural violations.

In short, this thesis presents live architecture compliance checking – an empirically validated, tool-supported approach for sustaining structure in software implementations. It enables development organizations to successfully rely on their software architecture as the instrument for guiding the evolution of the software system.

Acknowledgements

I would like to express my sincere gratitude to the people who supported me, accompanied me, and empowered me in the journey of my PhD.

First of all, I would like to thank Dieter Rombach for supervising my thesis and his guidance along the way. I appreciate working at Fraunhofer IESE, especially the opportunities I had and the lessons I learned here. Secondly, I would like to thank Jürgen Ebert for agreeing to review this thesis. My thanks go also to Jens Schmitt for steering the dissertation committee.

I would like to thank my friends and colleagues at Fraunhofer IESE for many fruitful discussions about my work, software architectures, and other topics. I appreciated your comments on my ideas, on early drafts of this thesis, or at the rehearsal of my PhD defense. Here I want to specifically mention Michalis Anastasopoulos, Joachim Bayer, Martin Becker, Ralf Carbon, Christian Denger, Slawomir Duszynski, Jörg Dörr, Thomas Forster, Mikael Lindvall, Dirk Muthig, Thorsten Keuler, Matthias Naab, Daniel Pech, Dominik Rost, Marcus Trapp, Christian Webel, Balthasar Weitzel. I am grateful that I had the opportunity to get to know you and to work with you. I also thank Sonnhild Namingha for proofreading this thesis.

Furthermore, I would like to express my sincere thanks to all members of the SAVE team. What started as a proof-of-concept turns out to become market-ready product these days. I would like to take this opportunity to thank everyone who was and still is involved in evolution of SAVE. I enjoyed working with you and appreciate all your contributions made to SAVE and SAVE LiFe.

I would like to thank the people who helped me realizing the experiments of my thesis as well as all participants of the experiments. Moreover, I would like to thank the many people from our industrial partners who allowed me applying my ideas.

Last but far from least, I want to thank my wife Catalina and my parents Martina and Hubert who have supported me in every possible way. I am eternally grateful for their constant love.

Kaiserslautern, May 2010
Jens Knodel

Table of Contents

List of Figures

List of Tables

List of Definitions

1 Introduction

The key assets for every software development organization are the implementations of its respective software systems. The implementation is the result of a planned application of *software engineering* principles, techniques, methods, and tools following a predefined development process (see [Naur 1968]). The implementation determines the overall success of the organization – delivering a software system with the requested functionality while meeting effort, quality, and time constraints is crucial. Because of these essential demands on every commercial software product, development organizations need to efficiently manage the implementations they produce. This need is an even more pressing issue in the evolution of the implementation, due to the laws of continuing change and of increasing complexity (these laws were stated by [Lehman 1985] and repeated by [Sommerville 2001]).

Producing the implementation is a coding process executed by several (teams of) developers. The developers write source code statements in order to translate solution ideas into algorithms and data structures. The implementation typically consists of many source code entities, which are the distinct building blocks of the system (e.g., up to hundreds and thousands of files or classes in procedural or object-oriented programming languages, which are distributed over many folders or packages). Due to the size and complexity of software systems, however, it is obviously not feasible to manage software development efficiently on the source code level. Abstractions are needed to handle the sum of source code entities and to manage the software system as a whole from a global perspective. Such abstractions enable efficient development and allow accomplishing the essential demands mentioned above. *Software architectures* – introduced as an auxiliary construct into the software lifecycle – promise to provide these abstractions.

The notion of software architectures was first introduced by [Zachman 1987]). Software architecture is the conceptual tool for efficiently managing and evolving software systems. They define the fundamental decomposition of a software system [IEEE-Std.1471 2000] and are based on the principles of “divide and conquer” [Endres 2003] and “separation of concerns” [Parnas 1972], [Dijkstra 1982]). The applicability, the usefulness, and the benefits of software architectures have been widely accepted in both research and industrial practice (e.g., see [Bosch 2000], [Clements 2003], [Hofmeister 2000], [Jazayeri 2000], [Perry 1992], [Rozanski 2005], [Shaw 1996], or [Tyree 2005]).

Consequently, in the 1990s, architecting became an integral part of any modern software engineering approach. This is, amongst others, reflected in [Boehm 1995], who states that "if a project has not achieved a system architecture, including its rationale, the project should not proceed". The software architecture is typically defined before any implementation activities start – in fact, it is the first solution-oriented artifact (in contrast to previous lifecycle activities, which constitute problem-oriented views). As soon as there is a first draft of the architecture, it is possible to predict whether or not the final system will meet the needs and concerns of the stakeholders. The abstractions provided by the software architecture enable effective communication among stakeholders and thus, allow sound decision making, which iteratively refines the architecture. Eventually, the abstract solutions designed in the architecture are transformed into their concrete counterparts in the source code – the implementation. To cover all the different stakeholder concerns and still provide simple, expressive documentation, several perspectives must be taken to describe the architecture. Such documentations comprise multiple architectural views on the software system (e.g., see [Kruchten 1995], [Davis 1997], [Hofmeister 2000], [Herzum 2000], [Clements 2002a], [Rozanski 2005], [Bayer 2004], or [Knodel 2006a] for examples of architectural view sets). When comparing the most commonly used view sets, all comprise the *structural view* (or something named similarly). The structural view describes the decomposition of the software system. It captures the static structure of a system in terms of layers, subsystems, components, and connectors, the interfaces provided by them, as well as the relationships and dependencies between the various elements and to the environment [Knodel 2006a].

The structural view is the most important architectural view for developers. Developers – being the stakeholders who realize the software system – receive task assignments on the basis of the information provided in structural views. They execute the assignment and, ideally, the developers would comply with all the decisions made by the architects. However, in practice, most development organizations fail to meet this challenge. Hence, what sounds like a straightforward task is hampered by the following reasons (this list is based on the work by [Eick 2001], [Gurp 2002], [Hochstein 2005], [Parnas 1994]):

- Developers have to bridge the abstraction gap between architecture and source code. The solutions sketched out by the architecture are documented on an abstract level. Developers, transforming and refining these abstract solutions into concrete implementations, have to translate abstract concepts into working solutions. However, the developers may fail at interpreting the documentations or may lack a clear understanding of the motivation behind the abstract solution.

- The primary goal of the developers is to get the system running. Developers work under tight time schedules and have constant effort pressure; they are typically overloaded with feature requests, development tasks, and so on. Their primary goal is to implement, solve the problem, and get a running version that fulfills the acceptance criteria of testing. For this reason, developers might use shortcuts or workarounds and may ignore the architecture.
- Developers have to switch the development context for each task assignment. When developers work on one development task (e.g., changing one component), a certain set of architectural decisions applies. When switching the context (i.e., working on another component), a different set of decisions might be valid. Nevertheless, developers may neglect the switch and continue working with a different context in mind.
- Developers work with a local, limited scope, while the architecture is balanced from a global viewpoint. They focus on a concrete problem only. However, the architecture might prescribe that developers implement locally sub-optimal solutions in order to satisfy global constraints. The developers may ignore these constraints by optimizing their local implementation.

All these arguments provide explanations for one dilemma of modern software development organizations: Almost all implementations exhibit significant *structural violations*. The coding process produces an output – the implementation – that diverges from its input – the structural view of the software architecture. In other words, the implementation lacks compliance[1], whereby structural violations denote the items where the realization (i.e., the actual or implemented system) deviates from its respective counterpart – the specification (i.e., the planned or intended system).

An analysis of industrial practice covering various software systems distributed across diverse application domains such as embedded systems or information systems revealed that there was not even a single system that the developers implemented in full compliance with the architecture. On the contrary, all systems analyzed featured substantial structural violations [Knodel 2006c]. Other researchers confirm that the lack of compliance is a practical problem in industrial practice (for instance, see [Murphy 2001], [Bourquin 2007], or [Rosik 2008]). But not only industrial software systems lack compliance: open source software systems face the same problem. The most prominent example here is probably the Mozilla web browser, where [Godfrey 2000] observed

[1] Please note that in this thesis, we refer to compliance as the compliance of structural views with the respective counterparts in the implementation. Thus, (unless stated otherwise) compliance means structural compliance, violation means structural violation, etc. Please refer to Appendix A for an overview of views and compliance checking.

significant architecture decay within a relatively short lifetime – the browser was still under development after a complete redesign from scratch. Another prominent study is reported in [Garlan 1995], where architectural mismatches resulted in a number of issues (e.g., excessive code, poor performance, need to modify external packages, need to reinvent existing functionality, unnecessarily complicated tools), which eventually hampered successful reuse of components.

The lack of compliance bears an inherent risk for the overall success of the development organization: The architecture as a communication, management, and decision vehicle for stakeholders becomes unreliable, delusive, and useless. Decisions made on the basis of the architecture are risky, because it is unclear to which degree these abstractions are actually still valid in the source code. Hence, structural violations seriously undermine the value of the architecture. It is unclear whether or not the development organization will meet the essential demands of the requested functionality delivered while meeting effort, quality, and time constraints for the software system under development. Even worse is the long-term perspective during maintenance and evolution, which was already observed by [Lehman 1985] stating that "an evolving program changes, its structure tends to become more complex". The source code surpasses innovations designed in terms of the architecture and can prevent their introduction. Because all decisions made to obtain the goals were derived from the architecture, the imperative need for architecture compliance becomes apparent.

The discipline of *architecture compliance checking* emerged from this need. Since compliance is always measured relative to distinct aspects, the inputs of compliance checking are always twofold: the architectural view as the planned specification and the system artifact as the actual realization, while the output is a collection of violations. Hence, structural compliance checking requires the structural view and the source code as input in order to reveal the structural violations. All structural compliance checking techniques are based on the same principles:

- First, the structural view of the architecture is processed in order to create a structural model comprising the architectural entities and the dependencies among them.
- Second, a snapshot of the source code is processed using some kind of reverse engineering technology [Chikofsky 1990]. The processing creates a source code model comprising the source code entities and the dependencies among them.
- Third, the structural model and the source code model are aligned on the same level of abstraction (e.g., lifting the source code model onto the level of the structural model). The alignment exploits expert

knowledge, architecture documentation, or implementation guidelines to bridge the abstraction gap.

- Fourth, the two models (now on the same level of abstraction) are compared against each other. The differences between the two models are detected and marked as structural violations. Backward traceability to the models allows locating the violations in the structural model or in the source code model for further analysis or processing.

As an analytic quality engineering technique, structural compliance checking can only be performed when all input is available. The developers have to first execute the coding process and thus, deliver the source code before compliance checking is possible. It is typically applied late in the software lifecycle [Lindvall 2003], which means that structural violations are only detected late in the development process. Structural compliance checking has been proven as a sound analytic quality engineering technique adopted by industry (e.g., see [Feijs 1998], [Krikhaar 1999], [Murphy 2001] [Postma 2003], [Riva 2004], [Knodel 2006c], [Kolb 2006] or [Bourquin 2007]).

To put it another way, structural compliance checking reveals the dilemma of development organizations: their implementations lack compliance with the architecture. However, structural compliance does not contribute to solving this problem. Knowing about violations does not remove them from the source code[2]. The late interception due to the late application of compliance checking leaves the development organization to decide between two fundamental options. None of them is really appealing to the development organization because they have substantial drawbacks:

- **React:** On the one hand, the development organization has the opportunity to react to structural violations and invest effort to remove them.
    - **Prompt reaction:** The short-term removal of structural violations causes an overhead effort in the ongoing project – at a late point in time. The overhead effort is spent on conducting workshops and meetings to discuss the compliance checking results and find appropriate remedies. The actual removal is a special kind of refactoring – changing the source code without changing the external behavior. These refactoring activities require a re-understanding of the source code causing structural violations and then implementing the change to achieve structural compliance. To avoid unwanted side-effects

[2] Please note that unless stated otherwise, we consider that the architecture takes over the implementation, which means we assume that the structural violation is caused by improper implementation and not by inadequate architectural design.

  introduced by the modifications made, all quality engineering activities have to be repeated (e.g., regression testing), including compliance checking, which, of course, may reveal newly introduced or forgotten violations.
  - **Deferred reaction:** Because of tight schedules or resources, the development organization may decide to plan for a special refactoring project that tackles only the structural violations. However, such a project has to be justified by higher management. To convince higher management to promote such a project is not trivial because neither added value nor innovation is visible to the users of the system. And a gradually growing risk is best expressed by the popular saying: "The later something is done, the more effort is required to do it".
- **Ignore:** On the other hand, the development organization may decide to ignore the structural violations (or, if the organization does not apply compliance checking at all, it faces the same problems without even knowing about it).
  - **Disregard in the short term:** Traceability between architecture and source code is not given anymore. The architecture as a communication and decision vehicle is no longer reliable and useful. The lack of up-to-date, consistent, and traceable architecture documentation causes a maintenance problem. Changing the system becomes error-prone due to unknown side-effects. The return-on-investment of software as a valuable asset decreases and, eventually, the evolution becomes unnecessarily time-consuming and effort-intensive.
  - **Disregard in the long term:** Detached from their architectures, the implementations evolve uncontrolled, disorderly and chaotically; in short, all benefits produced by well-defined architectures are lost. Over time, the implementation degenerates more and more. Eventually, this will cause the need for the documentation of the architecture to be reconstructed. Numerous approaches, methods, techniques, and tools for architecture reconstruction (sometimes also called architecture recovery) have been developed (the works by [Chikofsky 1990], [Deursen 2004], [Koschke 2005], [Knodel 2006b] and [Pollet 2007] present overviews on this research field). Architecture reconstruction, however, addresses only the symptoms of a lost architecture. But as stated in [Wallnau 1996], "although source code is often the most reliable arbiter of what a system does, it does not reflect all of the attributes of an application necessary to develop a true system-level understanding". Development organizations have to invest immense effort and resources in order to be successful in

the reconstruction – research in this broad field over the past twenty years shows that many development organizations suffer from the reconstruction burden and, more often than not, fail to meet this challenge.

Whatever the development organization decides to do along these alternatives, it will result in the essential demands of functionality, time, effort, and quality not being met – either due to a short term reaction creating overhead effort or in the long term, by affecting all demands negatively. To tackle this problem, this thesis proposes a new approach that turns analytical compliance checking into a quasi-constructive quality engineering technique. The primary contribution of this approach – called *live architecture compliance checking* – pro-actively detects the introduction of structural violations into the source code.

Live compliance checking pursues two primary goals: first, to achieve structural compliance during construction and second, to sustain compliance during the course of the evolution. It is executed continuously and constantly from day one of the implementation. Any modification made by any developer is analyzed immediately and, if necessary (i.e., if a violation was detected), live feedback is given straight to the developer originating the violation. This instant notification allows prompt removal of these violations. The constant repetition of this feedback whenever the same or another developer touches the same source code file (no matter at what point in time) spreads the information about the violation among the team of developers. Awareness for structural violations is raised early (i.e., almost in real time) and forwarded to its creator and all other developers working with the same source code elements – as long as the violations remain in the source code. The assistance provided by live compliance checking educates developers regarding the intended structural decomposition by constantly reminding them. Potentially, this education reduces the total number of violations introduced in the first place. In all cases, it prevents the drift between the structural view and the implementation. The high frequency (i.e., continuous live feedback) of compliance checking executions with fast response times turns the analytical technique into a quasi-constructive quality engineering technique.

In other words, live compliance checking sustains the upfront investment the development organization made into architecting. It raises the awareness of developers regarding structural violations, just after they have been introduced. Developers can react immediately and remove the violation by refactoring – as soon as it has been introduced and while they still have context information in mind. The mean time for *structural repair* – the time from inserting a violation until its removal – is reduced to almost zero (of course, only if developers pay attention to the live feedback provided by live compliance checking). Thus, development organizations can rely on the software architecture as an instrument for the successful and efficient development and evolution of software systems – a demand that is gradually gaining more and more importance and, at the same time, a challenge that currently most development organizations fail to meet.

The remainder of this introduction continues with a discussion of the role of compliance checking in the software development process (see Section 1.1) and pinpoints the effects of compliance on effort (see Section 1.2). The characterization and empirical evidence for the benefits of evolving a system with structural compliance – three replications of a controlled experiment providing evidence that compliance allows evolution with savings in effort, at least for the experimental task executed by the participants – motivate the proposed approach of this thesis: prevention of structural violations by live compliance checking. Section 1.3 introduces this new approach to achieving compliance by construction and sketches its essential requirements. Then Section 1.4 discusses the primary contribution. Finally, this introduction concludes with the outline in Section 1.5, which gives an overview of the structure and content of the remaining sections of this thesis.

1.1 Compliance Checking

The discipline that provides principles, techniques, methods, and tools for systematically developing and maintaining software systems is called software engineering.

Definition 1 Software Engineering

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software [IEEE-Std-610.12 1990].

Software engineering applies a defined development process to produce a software system that achieves certain functionality and certain quality goals. As an example, Figure 1 depicts the phases of a simplified development process based on the V-Model [Broehl 1995]. The V-Model comprises a set of artifacts that capture information about the system under development. Most modern development processes include the artifacts as depicted in Figure 1. The information flows step by step from left to right, passing all artifacts (i.e., from requirements to the accepted system in use). Each artifact allows the development organization to apply quality engineering activities.

Definition 2 Quality Engineering

Quality engineering subsumes all activities in an organization that contribute to the quality of end products. Methods and techniques enabling the achievement of a certain level of quality can be *distinguished into two categories: constructive and analytic quality engineering techniques (the latter are also called quality assurance techniques).*

Definition 3 Analytical Quality Engineering

*Analytic quality engineering is performed **after** the creation of the objects under examination to assess, verify, or validate them with respect to a certain quality.*

Definition 4 Constructive Quality Engineering

*Constructive quality engineering is performed **while** the object under examination is created and aims at proactive, a priori, preventive minimization of quality decay in the first place. Hence, they aim at achieving a certain quality by construction.*

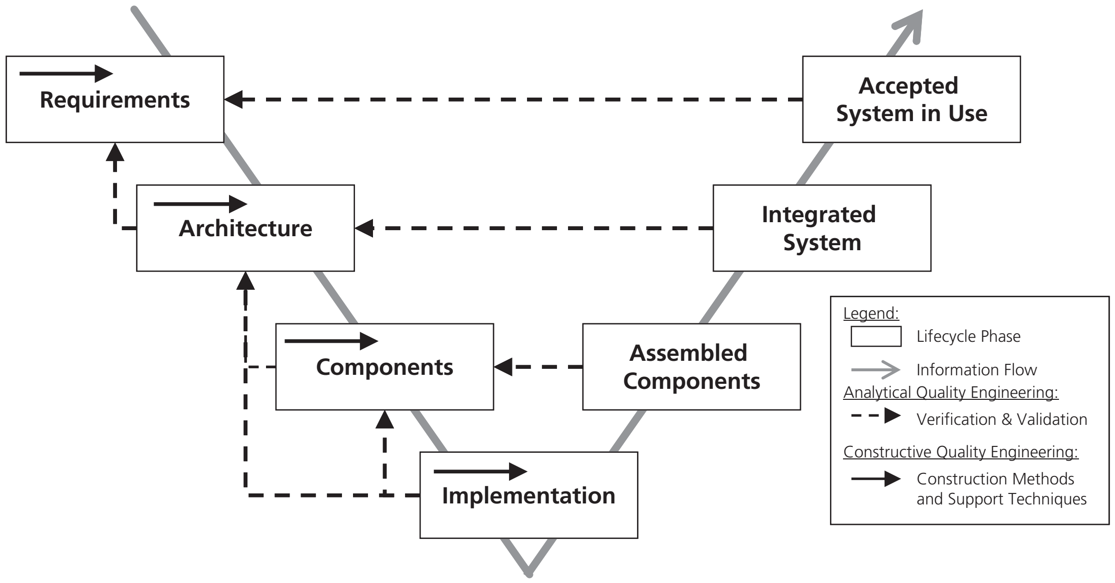


Figure 1 Simplified Development Process

On the one hand, constructive quality engineering techniques aim at building quality into the artifacts, which means minimizing defects during the construction of products (e.g., systems, components, documents). They prevent defects from being introduced. The construction activities aim at producing artifacts (i.e., in Figure 1, first requirements, then architecture and components, and finally yielding the implementation) with the desired quality. Each artifact refines the previous one by lowering the abstraction level towards the source code.

On the other hand, analytic techniques focus on artifacts in a stable state (i.e., after construction). They detect defects that need to be corrected afterwards. Analytical activities on the artifacts (i.e., in Figure 1, assembled components, integrated system, and eventually the accepted system in use) make sure that the resulting systems really exhibit the desired quality. Analytical quality engineering techniques are further decomposed into verification and validation (V&V) activities:

Definition 5 Verification

The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [IEEE-Std-610.12 1990].

Definition 6 Validation

The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [IEEE-Std-610.12 1990].

In short, V&V evaluates whether the product is being built right (verification) and that the right product is being built (validation). Applying these definitions means that structural compliance checking is a verification technique ensuring that the implementation realizes the architecture right (i.e., that structural compliance is achieved). However, it does not guarantee that the right implementation is produced (i.e., that the implementation satisfies the requirements). Hence, validation of architectures is not in the scope of this thesis[3].

As depicted in Figure 1, two artifacts are inputs to compliance checking: the architecture and the implementation. Because the architecture prescribes the fundamental decomposition, compliance checking – as an analytical technique – can be executed as soon as an integrated system is available.

Definition 7 Software Architecture

Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution [IEEE-Std.1471 2000].

Definition 8 Implementation

Implementation is the result of the process of translating a design into hardware components, software components, or both [IEEE-Std-610.12 1990].

Definition 9 Integration

Integration is the process of combining software components, hardware components, or both into an overall system [IEEE-Std-610.12 1990].

[3] Techniques for validating an architecture range from approaches that use questioning, structured interview, checklist, measuring, simulation (e.g., [Bosch 2000]), to scenario-based (e.g., [Clements 2002b]) techniques. Most popular are scenario-based evaluations. [Ionita 2002] and [Babar 2004] provide a comparison of selected scenario-based approaches.

The architecture is then input to component engineering and implementation. Based on a well-defined architecture documentation, developers refine the abstract concepts and models prescribed by the architecture. They implement the abstract solutions specified in the architecture. Architectural elements are realized as a set of source code files; architectural inter-element relationships are codified as usage dependencies using the capabilities provided by the respective programming language. They write code and codify algorithms and data structures to satisfy the requirements on a software system. The resulting concrete solutions are implemented source code.

When the writing of the source code has been completed, verification of the implementation is possible. The analytic quality engineering technique for verifying that the system's implementation was built right is called architecture compliance checking.

Definition 10 Architecture Compliance Checking

Architecture compliance checking is a technique that verifies to which degree the realized architecture A' complies with the specified architecture A. The inputs to architecture compliance checking are an architectural view and the respective counterparts in the software system; the outputs are a set of architectural violations.

The goal of architecture compliance checking is to reveal where the traceability between architectural views and the resulting software system is no longer given. This thesis analyzes structural architecture compliance checking. Structural compliance checking reveals violations introduced during the implementation phase.

Definition 11 Structural Violation

A structural violation is an architectural component or a relationship between components that has a counterpart in the source code, which is not realized as specified.

By its nature, compliance checking is a quality engineering instrument (see Definition 2). The application of compliance checking as an analytical quality engineering technique produces results late in the development process – at integration time. Analytic techniques focus on artifacts in a stable state (i.e., after construction). They detect defects that need to be corrected afterwards. Consequently, structural violations cause an overhead effort due to their late detection in the development process.

Applying live compliance checking as a quasi-constructive quality engineering technique aims at minimizing defects during the construction of products (i.e., the implementation). The early detection of structural violations right after their insertion allows their prompt removal. The fast response times for live feedback can reduce the mean time needed for structural repairs to the absolute minimum.

The next two sections investigate the architecture in more detail. While Section 1.2 delivers evidence for the overhead effort caused by lack of compliance and its late detection, Section 1.3 explains the principles of live compliance checking – the novel approach introduced by this thesis.

1.2 Effects of Architecture Compliance

In general, compliance means that certain characteristics articulated by stakeholders are, in fact, fulfilled by the actual product, typically realized by different stakeholders.

Definition 12 Compliance

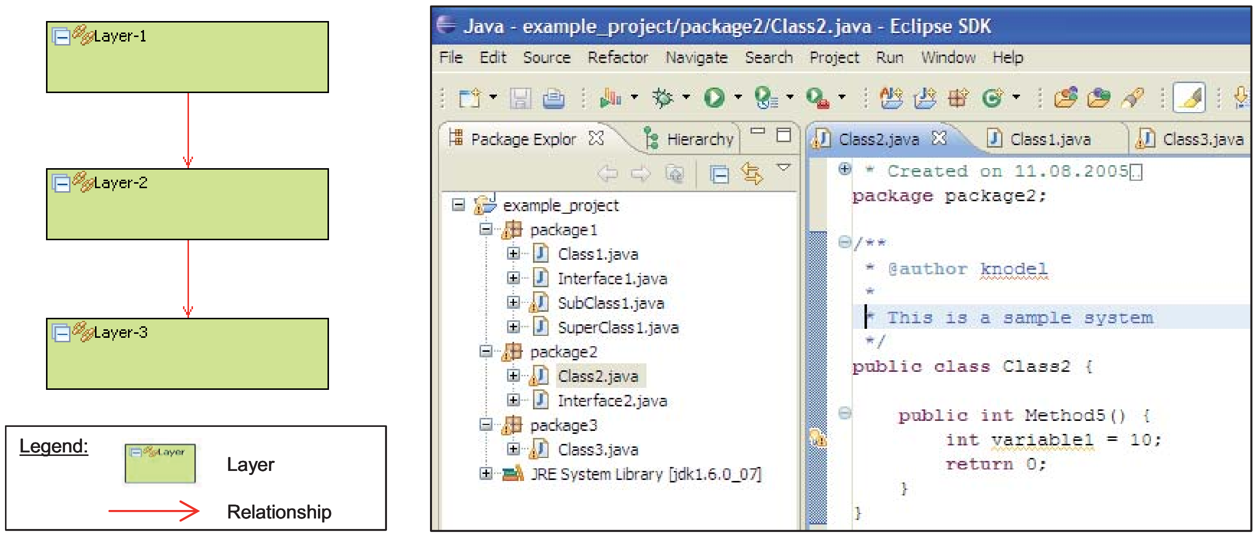
Compliance is the act or process of complying with a desire, demand, proposal, or regimen. Complying means to conform, submit, or adapt (as to a regulation or to another's wishes) as required or requested [Merriam-Webster 2009].

Compliance is always measured relative to two distinct aspects: on the one hand the intention, plan, or specification and on the other hand the facts, actual, or realization. Consequently, the translation of the compliance definition into the field of software architecture yields the following definition.

Definition 13 Architecture Compliance

Architecture compliance captures the degree of having accomplished required or requested demands realized in the implementations of software systems. Architecture compliance means that the specified architecture A_{spec} is equivalent to implemented architecture A_{impl}, hence $A_{spec} \leftrightarrow A_{impl}$.

Here, the architects define the required or requested demands, while the developers carry out their realization in terms of the source code in the implementation. If they code the structure as prescribed by the architecture, structural compliance is achieved. Of course, it is possible to define other types of compliance (e.g., see Appendix A for an overview), but these other types are outside the scope of this thesis. Having compliance between structural view and implementations thus means the absence of structural decay in the system. Traceability from the abstract elements of the structural architectural view to its concrete counterparts in the implementation is ensured and vice versa. Typically, traceability enables a one-to-many mapping, which means that one architectural element is realized by many source code elements. Figure 2 depicts an example structure comprising three abstract layers (left) and the corresponding implementation (right) in Java. Each layer is realized by a distinct package, and traceability is given by the respective numbering of layers and packages. This simple example illustrates the abstraction gap between architecture and implementation.



```
 * Created on 11.08.2005
package package2;

/**
 * @author knodel
 *
 * This is a sample system
 */
public class Class2 {

    public int Method5() {
        int variable1 = 10;
        return 0;
    }
}
```

Figure 2 Example: Structural View (left) with Strict Layering and Sample Implementation (right)

To motivate live compliance feedback for sustaining structure in the implementation – the contribution of this thesis – we conducted several empirical studies. First, a survey of industrial systems exemplifies that lack of compliance causes an overhead effort for structural repairs or reconstruction. Second, a controlled experiment replicated three times provides evidence about the benefits of compliance in terms of effort savings in the further evolution of the system.

1.2.1 Survey on Industrial Software Systems

This section analyzes in Table 1 the impact of lack of compliance for several industrial software systems[4]. In several case studies, we investigated the following research question: "*What is the impact of architecture compliance in the lifecycle of a software system?*".

In particular, our survey aimed at answering two questions: Do implementations of software development organizations lack compliance and if so, does this really cause an overhead effort for the development organization? This leads to the following null hypothesis H_0 to be tested and the alternative hypothesis H_1 – describing what is expected to happen:

- H_{L0}: The null hypothesis is that architectural compliance has no effect on the development or maintenance effort for a software system.

[4] Please note that the names of the software system, products, or architectural elements have been anonymized partly due to confidentiality reasons.

- H_{L1}: Lack of compliance creates an overhead effort for the development organization.

All cases described in Table 1 were elicited by informal interview with the stakeholders of the development organization (i.e., the architects, developers, or project managers) of the respective case. The author of this thesis assessed the compliance status using the Fraunhofer SAVE tool [Knodel 2009a], but was neither involved in development nor in maintenance activities at the development organization. Table 1 describes each case following a template: The *context* gives general information about the development organization and the system or product line under examination, *compliance status* reports on the degree of compliance (please refer to Section 2 for details on measuring compliance), and *impact* discusses the overhead caused or problems experienced due to the lack of compliance.

Case	Description
Case A – Embedded System	**Context:** Testo AG is developing a product line of climate and flue gas measurement devices. The software part of these measurement devices is developed by about 35 developers. They are responsible for maintaining and evolving the existing products and, of course, for developing new products. All measurement devices share the same reference architecture [Schmid 2005] and a common core of reusable components, called framework. After release to the market, the first three products were analyzed with respect to their compliance.
	Compliance Status: Compliance checking measured the following degrees of compliance: product P1 was 95.7% compliant, P2 89.8 %, and P3 only 72.7%. Quantified, the number of structural violations was in the range of 4-digit levels, especially P3 had more than 2000 distinct violations. Several reusable core components had dependencies on product-specific parts, which basically ruined their reusability (see [Kolb 2006] for more details).
	Impact: The compliance status resulted in a dedicated but unplanned restructuring project aimed at removing structural violations. The restructuring project bound significant resources of the development organization; workshops for communicating the compliance status alone consumed more than 10 person-days of effort (2-day workshop with more than 5 attendees on average, neither counting preparation nor time for solution finding and any structural repairs). The structural violations negatively affected reusability and imposed a major threat for the future evolution of the product line. The architects and developers invested significant effort in a dedicated restructuring project to tackle the structural violations (unfortunately, no effort data was tracked for this project).

Case	Description
Case B – Multi-Media System	**Context:** The development of a multimedia system was adapted to new hardware technology including a graphics component (implemented in C++, comprising approximately 180 kilo lines of code). At the time of the analysis, the component was still under development. To ensure adequacy of the graphics component, the architects were interested in the compliance status of the component.
	Compliance Status: The compliance status of the component was analyzed twofold. First, compliance with the intended decomposition into the three layers was analyzed. Second, the layer internals of the component were analyzed. Although the implementation was still in progress, minor violations had already made their way into the component. Only few – less than 1% – dependencies violated the top-level layering, but by that time, only two out of three layers had been realized (the third layer only existed in stubs). The detailed analysis of the layers detected further minor compliance issues (see [Knodel 2005c] for more details).
	Impact: Due to the fact that the implementation was still in progress the developers' attention was called to the structural violations. However, it is remarkable that there already was a lack of compliance, although there was not even a first release of the graphics component yet. Moreover, additional effort was spent on running a workshop to communicate the compliance checking results.
Case C – Information System	**Context:** The development organization is producing software for the management of stock market data. The case study analyzed a high-end asset management system (large-scale system programmed in Delphi, 2-tier client-server architecture, developed for more than 10 years). To address new market requirements only a subset of the current product functionality was needed, hence it was decided to extract one core component and reuse it in the new products.
	Compliance Status: By the time of the analysis, the core component existed only on paper. It was not possible to draw clear line between the component and other parts of the system. The component was heavily coupled to user interface, database access, and business logic and included a large number of dependencies on certain initialization and global variables. The core component was not encapsulated with clearly defined, explicit dependencies; it was rather an integral part of the system.
	Impact: The lack of compliance led to a decision (i.e., reusing the component), which could not be executed due to its blurred boundaries. Instead, resources had to be devoted to a reconstruction project, which analyzed reuse feasibility, identified potential risks, and redocumented the interface of the component. The negative findings of the reconstruction project revealed the technical pitfalls regarding potential reuse of the component, and the development organization decided to strive for an alternative solution (see [Knodel 2004] for details). In short, reuse failed because the implementation lacked compliance.

Case	Description
Case D – Information System	**Context:** Because of exceeded time and budget constraints, unreliable functionality, and an overall unclear status, the customer of a development project asked Fraunhofer IESE to conduct an independent architecture assessment for a software system developed by an external contractor (implemented in Java, comprising several million lines of code).
	Compliance Status: The compliance checking of the system addressed several structural viewpoints. All of them revealed substantial structural violations between intention and actual implementation (the percentage of violations was considerably higher than 5%). Most noteworthy was the analysis of the layering, where almost 33% (quantified: nearly 50000) of the dependencies were violations.
	Impact: The mismatch between architecture and implementation was so severe that the customer lost confidence in the abilities of the contractor to repair the structural violations. Eventually, the development project was canceled by the customer. Hence, the previous investments into this project were made in vain.
Case E – Testbed for Avionics	**Context:** TSAFE is a prototype of the Tactical Separation Assisted Flight Environment specified by NASA Ames Research Center [Erzberger 2001] and implemented at MIT [Dennis 2003]. Fraunhofer Center Maryland (CESE) turned TSAFE into a testbed to be used for software technology experimentation (implemented in Java, comprising 20 kilo lines of code). One of the technologies experimented with was architecture compliance checking (see [Lindvall 2005] for details of the study).
	Compliance Status: The TSAFE testbed was seeded with six detectable structural violations, which were then presented to the subject (a senior person skilled in applying compliance checking). The subject also had the architecture documentation at hand and was asked to inspect the issues.
	Impact: In this special setting, detailed data could be collected for the analysis of the structural violations. In total, six violations were detected by the compliance checking technique. The subject spent four hours just on inspecting the violations, judging their potential impact, and reporting on them. The effort data collected for this rather simple testbed is a good indicator for the potential overhead effort caused by lack of compliance and follow-up structural repairs.

Table 1 Compliance Survey on Industrial Software Systems

The findings of Table 1 show that lack of compliance is, in fact, a recurring, practical problem in industry, which many software development organizations face. Actually, we could not observe a single case where there were no violations at all. The list of cases in Table 1 is far from complete; we present more cases in [Knodel 2006c], and other researchers also report on the lack of compliance (e.g., see [Murphy 2001], [Koschke 2003], [Bourquin 2007], [Rosik 2008]).

For all cases in Table 1, we could observer an overhead effort caused by a lack of compliance. These are hints to accept the hypotheses H_{L1} – lack of compliance creates an overhead effort in software development organizations. This negative impact motivates the pro-active prevention of structural violations. However, so far, we have not shown yet that compliance really allows effort-efficient evolution. The next subsection will investigate this topic in detail.

1.2.2 Evidence of Compliance Benefits

In the research community, it is assumed that architecture compliance has a positive effect on the evolution of a software system. Changes (either modifications of existing parts or extensions) can be realized efficiently because the developers can rely on the information provided by the software architecture documentation. Hence, in order to motivate a new compliance checking technique – as we do with live compliance checking – we have to provide evidence that lack of compliance is one of the root causes for effort-intensive and time-consuming evolution.

Therefore, we designed a controlled experiment that addresses this claim. Our research goal was to understand the benefits of compliance in the evolution of a software system. We compared the effort for an evolutionary task on two functionally equivalent implementation variants of the same system – one with significant structural violations and the other one realized in compliance with the architecture. As a side-effect, we could observe whether or not the structure of the respective system was kept in place.

In total, we conducted three replications of the experiment. The first run took place at the Fraunhofer-Center for Experimental Software Engineering, Maryland, USA (CESE). The subjects were students working as interns at CESE. The second replication was performed as part of the project ArQuE (Architecture-Centric Quality Engineering, which is a German research project partially funded by the German Ministry of Education and Research (BMBF)). And the last run took place at the Technical University of Kaiserslautern, Germany, as part of the practical lecture "Grundlagen Software Engineering (GSE 2008)". Table 2 gives an overview of the three replications. Preceding the three runs, we conducted a pilot with two students to test the experiment materials and the solvability of the task. Both were able to accomplish the task successfully. Based on their feedback, we changed minor wordings in the material.

The group name encodes the context (first letter) and the group membership (second letter). Group A worked with the implementation containing structural violations, while Group B had the compliant implementation. Each time the participants of the experiment were randomly assigned to one group. All participants received the same material (except for the implementation variant) and had to solve the same evolutionary task.

Context	Group	Subjects	Subject Type
Pilot	P_A	1	computer science students, average experience: 6 semesters
	P_B	1	
CESE	F_A	2	computer science students, average experience: 7 semesters
	F_B	2	
ArQuE	I_A	4	professional software engineers, average more than 10 years of industry experience
	I_B	4	
GSE2008	K_A	9	computer science students, average experience: 6 semesters
	K_B	7	

Table 2 Compliance Experiment – Replication Overview

The hypotheses were also the same in all three replications. The first hypothesis captures the comparison of the effort required, while the other one addresses the correctness of the task:

- $H_{C0.1}$ – The null hypothesis is that compliance has no effect on the effort required to accomplish the evolutionary task.
- H_{C1} – The evolutionary task can be realized with less effort for the architecture-compliant implementation.

- $H_{C0.2}$ – The null hypothesis is that compliance has no effect on the correctness of the evolutionary task.
- H_{C2} – Compliance supports the subjects in accomplishing the correct solution (i.e., correctness here means being compliant to the reference solutions provided by an independent expert beforehand).

To operationalize the hypotheses, we used two already existing functionally equivalent implementation variants of the TSAFE testbed[5] [Lindvall 2005], which were implemented based on the same reference architecture. Variant A actually was a predecessor of variant B, which underwent major restructuring.

The experimental task was stated as follows: "*To support distributed development and outsourcing, TSAFE has to be refactored into distinct components. Each TSAFE component has to be realized in a separate Eclipse project, which can then be managed and evolved by an independent development group. After the refactoring, ensure that TSAFE is working correctly: no compilation errors, and pass of system test.*" As support all subjects received the architecture documentation

5 Please note that the two variants are different from the variants discussed in case E in Section 1.2.1.

and technical material on Eclipse and its task-relevant features. Then the subjects conducted the task independently.

We collected effort data in person-hours and measured the correctness of the task a-posteriori (i.e., we compared the individual solutions to the expert reference and measured the matching degree).

The average results of the three replicated runs for the different groups are listed in Table 3, while the box plots in Figure 3 and Figure 4 detail results depicting the corresponding effort data and task correctness of all subjects (excluding the pilot) per group.

Run	Group	Effort (in minutes)	Correctness (in %)
CESE	AVG(**F_A**)	114,00	61,00
	AVG(**F_B**)	50,00	100,00
ArQuE	AVG(**I_A**)	103,25	66,00
	AVG(**I_B**)	88,00	100,00
GSE2008	AVG(**K_A**)	116,22	73,11
	AVG(**K_B**)	37,71	99,00
Total	AVG(**A**)	**112,47**	**69,60**
	AVG(**B**)	**55,08**	**99,46**

Table 3 Compliance Experiment – Results

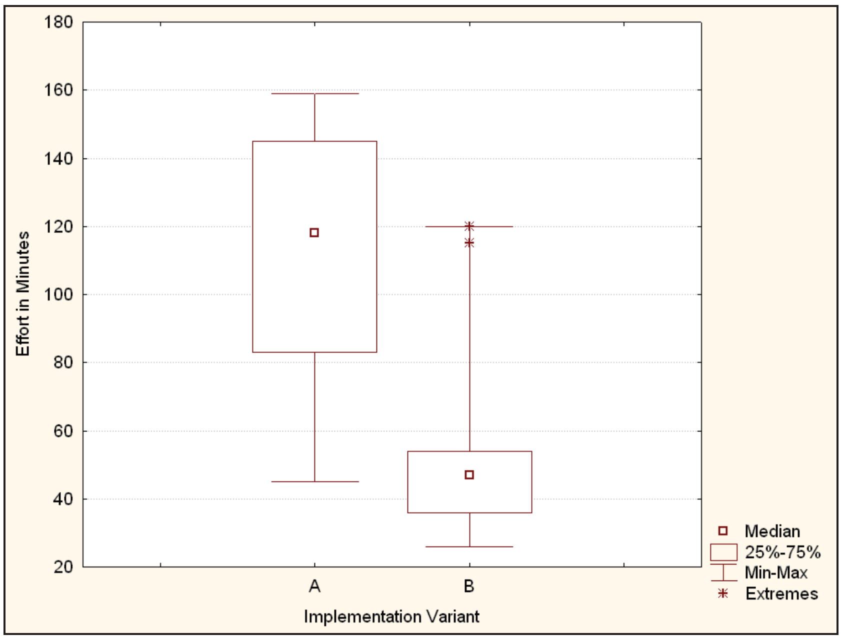


Figure 3 Compliance Experiment – Box Plot Effort Data

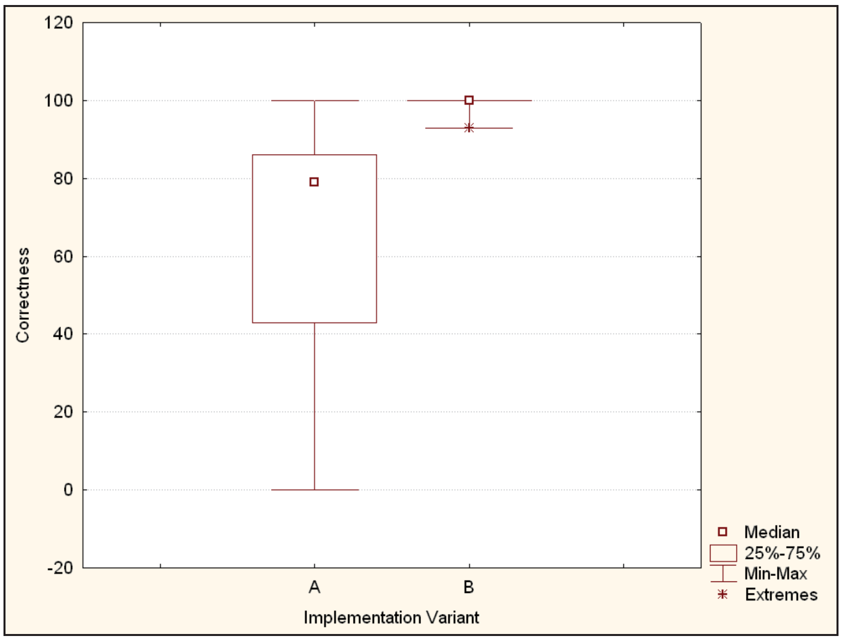


Figure 4 Compliance Experiment – Box Plot Correctness

Overall, we can make the following observations for the effort data (see Figure 3):

- The box plot shows good separation for the effort data.
- On average, subjects in group B (working with the architecture-compliant implementation variant) required less effort to solve the task. This is true for all three replications.
- In group A (the implementation variant lacking compliance), there are larger deviations from the average effort data than in group B.
- The best participant of group A (45 minutes) required almost the same effort as the median of group B (47 minutes), whereas the worst participant of group B (120 minutes) required almost the same effort as the median of group A (118 minutes).

With respect to correctness (see Figure 4), the following observations can be made:

- The box plot shows good separation for correctness. All but one participant of group B (working with the architecture-compliant implementation variant) accomplished 100% correctness for the evolutionary task.
- In group A (the implementation variant lacking compliance), there are larger deviations from the average correctness than in group B.
- The best participant of group A accomplished 100% correctness. The worst participant of group B (93% corrected) outscored more than 75% participants of group A.

The briefing questionnaire analyzed the background of the participants. Answers could be given on an ordinal scale with five values ranging from 1 (“none at all”) to 5 (“professional”). The average experience of the participants in both group was very similar (Java: A=2.60 vs. B=2.23; Eclipse: A=2.47 vs. B=2.31; Refactoring: A=1.33 vs. B=1.38).

Debriefing questionnaires revealed further interesting findings. Answers could be given on an ordinal scale with six values ranging from 1 (“don’t agree at all”) to 6 (“totally agree”).

- The participants further understood the task well (A=4.60 vs. 4.69) and were comfortable in applying the Eclipse refactoring (A=5.27 vs. B=5.23).
- The participants agreed that the goal of the task (A=4.13 vs. B=4.31), the task description (A=4.80 vs. B=4.77), and the architecture description including the component (A=4.40 vs. B=4.85) were clear.
- The participants agreed that the task was realistic (A=4.40 vs. B=4.38).

The results show that compliance of the implementation – as the only varying factor between the two groups – allows effort-efficient evolution of software systems. The lack of compliance caused, on average, an effort twice as high, or 204% (average of group A/average of group B). Moreover, all but one participant of group B solved the task 100% correctly, while group A had only an average of 69.60%. For a detailed discussion of the experiment and its threats to validity, please refer to [Knodel 2009b].

1.2.3 Summary

In short, we presented a discussion of many industrial cases where structural repair due to lack of compliance consumed substantial, unplanned effort in the lifecycle of the software system. The three replications of the controlled experiment could isolate compliance as one of the root causes for high maintenance and evolution effort.

Based on this empirical evidence, we claim that sustaining compliance is a worthwhile research theme. The solution proposed by this thesis is a new approach: We turn compliance checking into a quasi-constructive quality engineering activity that pro-actively enables developers to adhere to the intended structure – while changing the implementation, they receive live compliance feedback on all changes made. Hence, investments into architecture – made in advance – pay off while the system evolves. Compliance grants traceability between architecture and code and the structural decomposition is sustained.

1.3 Principles of Live Compliance Checking

To harvest the fruits (i.e., the effort saving) promised by compliance, we had to define an approach that is able to prevent structural decay and sustains architectural structures over time. The main underlying idea of this approach is to transform compliance checking into a quasi-constructive quality engineering technique. This section outlines the key principles of live compliance checking (to be detailed in the remainder of the thesis).

To understand the requirements and constraints imposed by the live feedback approach, we delineate it from the regular analytical approach, where the compliance of single system snapshots is checked offline. Snapshots here represent one distinct version of source code of the system at a certain point in time. Figure 5 depicts the five conceptual phases of analytical compliance checking – abstracted from concrete techniques.

- **Architecture:** The architect models the intended structural decomposition of the system in terms of components and relations among them. This description of the to-be plan is the input for the developers to start the implementation. Besides, the structural model serves as input for the compliance check in the analysis phase.
- **Implementation:** Developers translate the abstract solution into source code (i.e., algorithms and data structures) adhering to the plan. By refining the abstract, coarse-grained entities, they implement many fine-grained, concrete code elements. Typically, teams of developers concurrently produce the source code or modify and extend existing code. Eventually, the source code is stored in a repository. The repository allows analyzing a distinct snapshot of the software system.
- **Analysis:** The analysis processes the predefined structural model and one distinct snapshot of the source code to run the compliance check. It is possible to apply model-based or rule-based compliance checking techniques (see Section 3 for a comparison). However, independent of the technique chosen, the results are equivalent. The checking results distinguish between compliance and violations.
- **Communication:** The architect reviews the checking results and defines structural repair tasks for the respective developers. The same developers causing the violations would be the ideal candidates for correcting it.
- **Correction:** The correction phase then realizes the structural repairs. Again, teams of developers concurrently refactor violating code parts. Correction leads to another implementation phase, which means iterating again over all phases to avoid unwanted side-effects and verify the corrected violations.

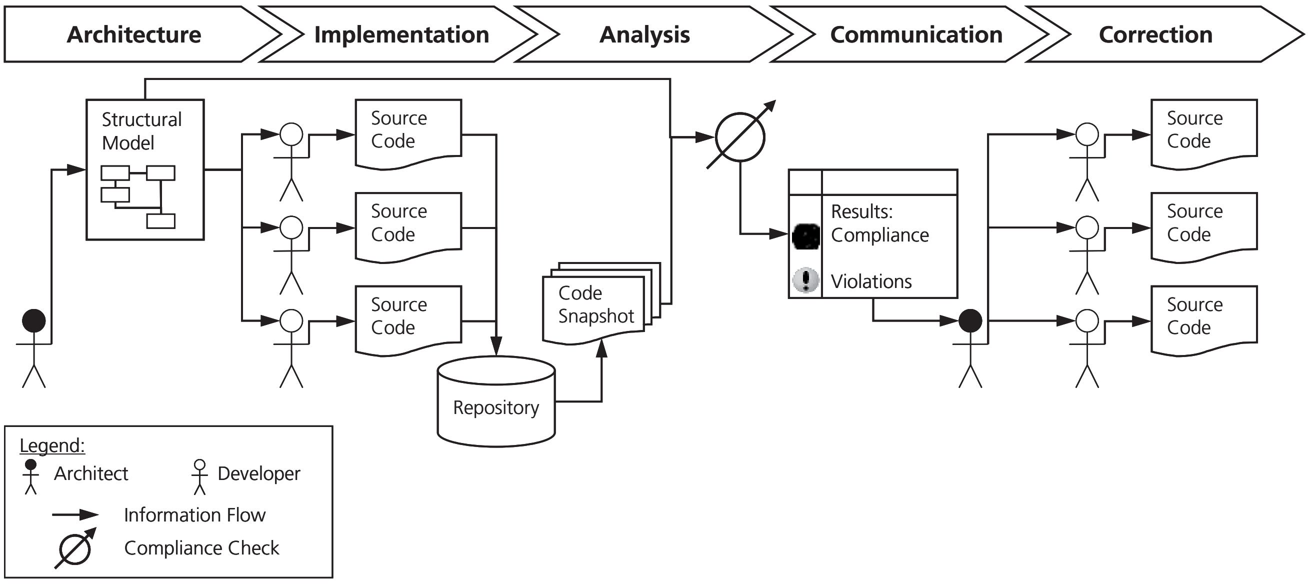


Figure 5 Analytical Compliance Checking of System Snapshots

Figure 6 shows how live architecture compliance checking works. In contrast to Figure 5 and contrary to regular analytical compliance checking, we have only two phases, architecture and implementation:

- **Architecture:** The architecture phase is the same as for analytical compliance checking – the structural model is the input for the follow-up phase.
- **Implementation (enriched with Live Analysis and Direct Communication):** As for analytical compliance checking, developers, of course, produce source code on the basis of the structural model. But in contrast to it, the implementation phase is enriched by live analysis of all deltas (i.e., the code just written) and direct communication about violations to the developers causing it (i.e., live feedback to the originators of violations). Because they receive instant feedback, developers are enabled to react promptly. Their minds are still in the current context (i.e., they are still working on the same task, they just wrote the violating source code lines). Thus, they can immediately repair the structure with minimal, close-to-zero effort. In other words, the compliance check and the correction happen while the developers are implementing. Thus, the developers are constantly trained and educated regarding the architecture.

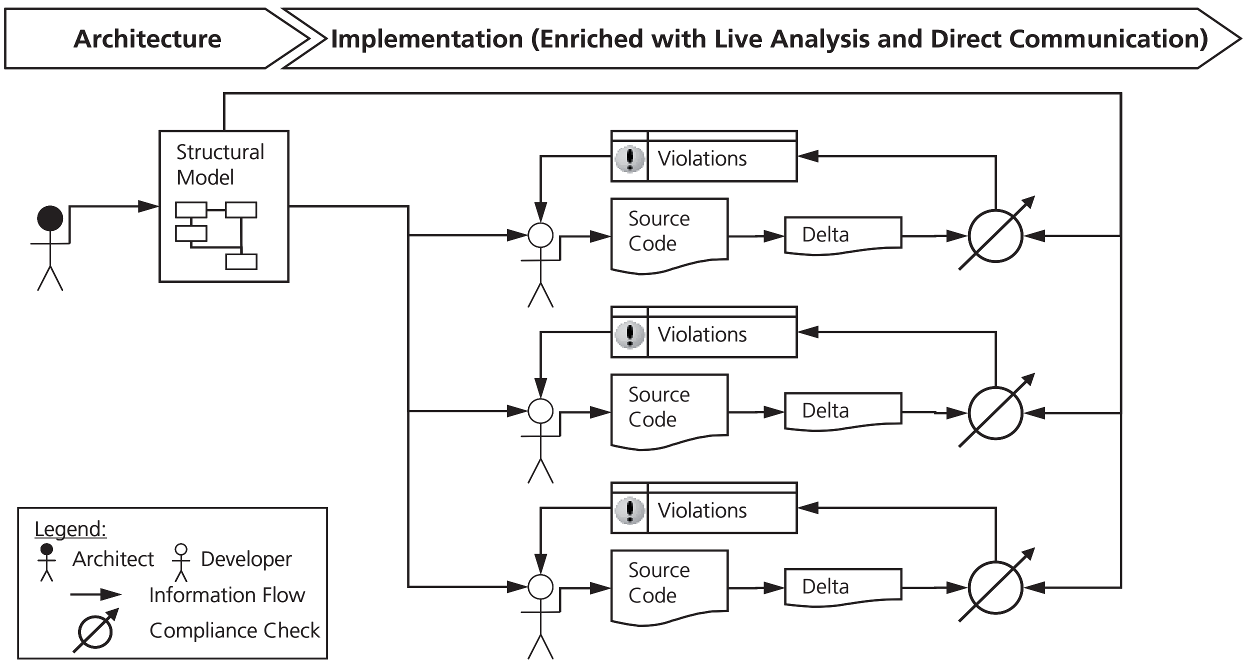


Figure 6 Live Compliance Checking

The points in time for analytical compliance checking can be formulated as follows: $t_{Architecture} << t_{Implementation} << t_{Analysis} < t_{Communication} << t_{Correction}$. Except between analysis and communication, there is a considerable delay (i.e., several days, weeks, or months) between the stages. Live compliance checking executes the four phases (i.e., implementation, analysis, communication, and correction) at the same time. The high execution frequency with live feedback justifies the classification as a quasi-constructive quality engineering approach, which leads to $t_{Architecture} << (t_{Implementation} = t_{Analysis} = t_{Communication} = t_{Correction})$. While there is still a delay between the architecture and the follow-up phases, all other phases are performed concurrently.

To be considered as quasi-constructive, compliance checking has to satisfy these timing constraints – and thus has to be executed *live*. Based on these characteristics and the concepts shown in Figure 6, we can deduce a set of essential requirements for live compliance checking as listed in Table 4.

Requirement	Description
Live Feedback	To empower developers to promptly remove structural violations with minimal or zero effort, they need to be made aware of the violations as soon as possible. The least delay obviously has immediate feedback: just after the source code has been written, developers receive live instant feedback on compliance.
Ease of Use	Any developer shall be able to interpret the live feedback. There should not be any extra effort to understand the feedback; further, there should not be a need for special training or support.

Requirement	Description
High Execution Frequency	The high execution frequency of compliance checking requires computer-aided, tool-supported automation. The results have to be computed without humans being involved in the regular execution. Of course, for initialization and major changes (e.g., adaptation of the structural model, new developers), the architect has to manually manage the configuration of live compliance checking.
Delta Analysis	The analysis of local deltas defines the limited scope for compliance checking. Each developer only receives feedback on violations that are within the respective modification scope (i.e., the file currently being edited). The delta analysis allows direct feedback directed to the originator or to the developers currently changing the source code files causing violations.
Distributed Team Support	Organizations typically produce software systems with many distributed teams of developers. In order to be successful live compliance checking has to support this distribution. Further, it has to scale to the number of developers implementing the system and to the size of the software system.
Smooth Integration into Environment	One characteristic of live compliance checking is the avoidance of unnecessary context changes. We consider the integration as smooth if developers are not required to switch to another context (i.e., by opening a web site or another tool). Hence, in order to not distract developers from their current task, the feedback results have to be presented in a non-intrusive manner. Ideally, the feedback presentation is presented as part of the source code editor of the integrated development environment (IDE) that the developers use to write code.
Robustness	Live compliance checking starts on the first day of development. It is executed all the time and while the system is still being written. Compliance checking has to be robust towards incomplete source code and compilation errors.
Commit Control	Source code commits to the repository (i.e., the configuration management system) should be under commit control. The architects have the option to apply either strict control (no commit possible for source code files containing violations) or loose control (violations can be committed).
Separation of Roles: Architect and Developer	The two roles of compliance checking – architect and developers – have to be acknowledged in live compliance checking, too. Architects define and manage the structural view. They are the only ones in control of changing the view and updating it, if necessary. Furthermore, the architects are interested in a global view on the whole system. They have to analyze the full picture, spanning over local changes made by the developers. They require architecture visualization and in-depth analysis capabilities to reveal architectural flaws. In contrast to the architect, developers have a local viewpoint; they only need to see the impact of their local modifications. Hence, they can ignore structural violations in other parts.

Table 4 Essential Requirements on Live Compliance Checking

A solution meeting the requirements stated in Table 4 leads to a tool-supported approach for live compliance checking. Such a solution would empower developers to sustain structure during the evolution of the software implementation. Sustainable structures in implementations yield less time-consuming and less effort-intensive evolution because the benefits promised by architectures are valid over time. The effort savings are achieved due to the following characteristics achieved by live compliance checking:

- **Live Compliance Checking Characteristic (LCCC) 1: Live compliance checking continuously educates and trains developers on the architecture**: The constant live feedback educates developers and trains them on the structural decomposition as specified by the architect. The continuous feedback produces a learning effect: Over time, less structural violations are introduced.
- **Live Compliance Checking Characteristic (LCCC) 2: Live compliance checking allows the prompt removal of structural violations**: The developers' awareness is raised immediately after the violations have been introduced. This enables developers to remove the violations with minimal effort because they are still in the problem context and no effort for re-understanding the source code is required.

In short, live compliance checking promises sustainable structure, which yields effort savings for the development organization. This thesis introduces such an approach to achieving architecture compliance by construction. The next section sketches the contribution and characterizes it threefold: The scientific, engineering, and empirical building blocks of this thesis are summarized (and detailed in the remainder of the thesis).

1.4 Contribution

The research questions addressed in this thesis are centered on architecture compliance. In particular, we investigate the following questions:

- What is the impact of architecture compliance on software implementations?
- How can we achieve the construction of a software system with compliance? How can we sustain compliance in the evolution of a software system?
- Is it possible to turn analytical compliance checking into a quasi-constructive quality engineering technique? What are the effects of live compliance checking? Is this technique adequate for achieving compliance in the construction of software implementations? Does it sustain structures over time?

- How can we support live compliance checking with tools? What are the requirements to enable live analysis and instant feedback? Can such a technology be successfully transferred to a development organization?
- Does live compliance checking educate developers on the architecture? Does it cause a learning effect? And will developers eventually use the live feedback to remove structural violations?

1.4.1 Research Method

To examine these research questions, we applied the principles of the experimental software engineering paradigm [Basili 1993]. Figure 7 summarizes the stages of the experimental software engineering paradigm (the boxes from left to right map to the respective steps and list the respective section in this thesis relevant to the step).

- First, we observed industrial software development organizations in order to identify a practical problem. In our survey covering various application domains, we were able to show that the evolution of software implementations is time-consuming and effort-intensive. A common factor shared by all software implementations analyzed was the lack of compliance – causing substantial overhead effort for the development organization in reaching its development objectives and for applying refactoring or restructuring projects to repair the implementation.
- Second, context factors influencing the problems were analyzed to identify the underlying scientific problem and to investigate it in detail. We conducted three replications of a controlled experiment with compliance as the only varying factor. Solving a sample evolutionary task for the variant comprising structural violations led to effort data twice as high (204%) than solving the same task for the compliant variant. The results constitute the research questions tackled in this thesis: How can we successfully sustain structure in software implementations over time?
- Third, an innovative solution was defined and introduced – live compliance checking. This new approach continuously monitors source code modifications, pro-actively detects structural violations, immediately provides developers with live feedback, and enables developers to react promptly. Hence, compliance is sustained, which avoids decay in software implementations.
- Fourth, we performed empirical evaluation to prove that the solution actually addresses the problem (i.e., that there is a scientific benefit). For this purpose, we conducted an experiment where 19 students developed a system over a period of 35 days. The students were assigned to two groups, both applying a regular development approach – one group with live feedback on compliance, the other one without. The experiment provided evidence about the positive

effects of live compliance checking – resulting in 60% less structural violations for the supported group.

- Fifth, we further empirically evaluated the effects to show the desired benefits in industry. In two case studies, we observed that compliance degrees of up to 99% are feasible – after investing effort for structural repair and with regular feedback on compliance (though not live). The sustained compliance increased the productivity of the development organization. While spending the same effort as before, the development organization could produce, evolve, and maintain more systems at the same time. Though compliance might not be the sole factor responsible for this fact, the industrial stakeholders confirmed in interviews that it is at least one of the crucial factors.

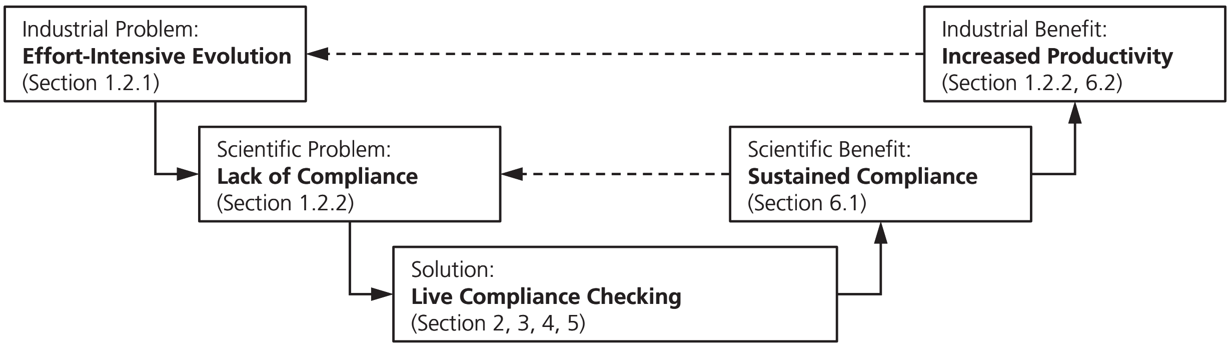


Figure 7 Experimental Software Engineering Paradigm: Characterization of Problem, Solution, and Benefits

1.4.2 Proposed Solution

The *empirical building block* of this thesis distinguishes between the eligibility of compliance as a worthwhile research theme on the one hand and the validation of live compliance checking on the other hand. In the empirical part, in particular, we provide evidence for the following hypotheses – the first two addressing compliance in general, the latter two addressing regular (or live) feedback:

- $\mathbf{H_{C1}}$ – Evolution consumes less effort for implementations that are architecture-compliant.
- $\mathbf{H_{C2}}$ – Compliance supports structure in remaining compliant over time.
- $\mathbf{H_{F1}}$ – Live feedback on compliance reduces structural violations in implementations.
- $\mathbf{H_{F2}}$ – Compliance increases the productivity of the development organization.

As depicted in Figure 7, the empirical motivation for this thesis is given in Sections 1.2.1 (addressing H_{C1}) and 1.2.2 (addressing H_{C2}), while evidence for the positive impact of live compliance checking and regular feedback is reported in Sections 6.1 (addressing H_{F1}) and 6.2 (addressing H_{F2}). Further, the effort reduction shown in the three replications of the experiment reported in 1.2.2 is another hint for the increased productivity in an industrial context.

The solution introduced by this thesis is split into two distinct building blocks – scientific and engineering (see Figure 8). The *scientific building block* comprises three parts:

- The measuring compliance part (see Section 2) presents a formal definition of the meta-models for the structural view, the source code model, and the mapping required between them. Based on these models, the compliance metric as frequently applied throughout this thesis is defined. The compliance metric is a relative measure that captures the percentage to which a software implementation accomplishes the structure specified.
- The techniques part (see Section 3) characterizes first the base technologies underlying any compliance checking technique: reverse engineering and its discipline fact extraction, which mines the source code for relevant information. Then the survey on the state of the art in analytical compliance checking reveals the equivalence in expressiveness of the two most prominent (and practically the only relevant) techniques – Reflexion models and dependency rules. The comparison of the two techniques shows that the Reflexion model technique outscores dependency rules in terms of applicability, explicitness, and ease of use with respect to live compliance checking.
- The approach part (see Section 4) finally introduces how the concepts of live compliance checking have been composed into a well-defined solution. This new approach defines adapted processes for architect and developer. Further, we discuss the positive effects of how the high frequency of live feedback can reduce the overall development effort.

Besides the scientific contribution, the *engineering building block* (see Section 5) presents the tool support for live compliance checking. To systematically counteract the drift between architecture and implementation, the compliance checking tool has to be fully automated, integrated with the development environment, and scale for the whole development organization.

As part of the thesis, we developed the SAVE LiFe tool – <u>S</u>oftware <u>A</u>rchitecture <u>V</u>isualization and <u>E</u>valuation with <u>Li</u>ve <u>Fe</u>edback (see Figure 9). SAVE LiFe is a client-server variant built on top of the Fraunhofer SAVE tool [Knodel 2009a].

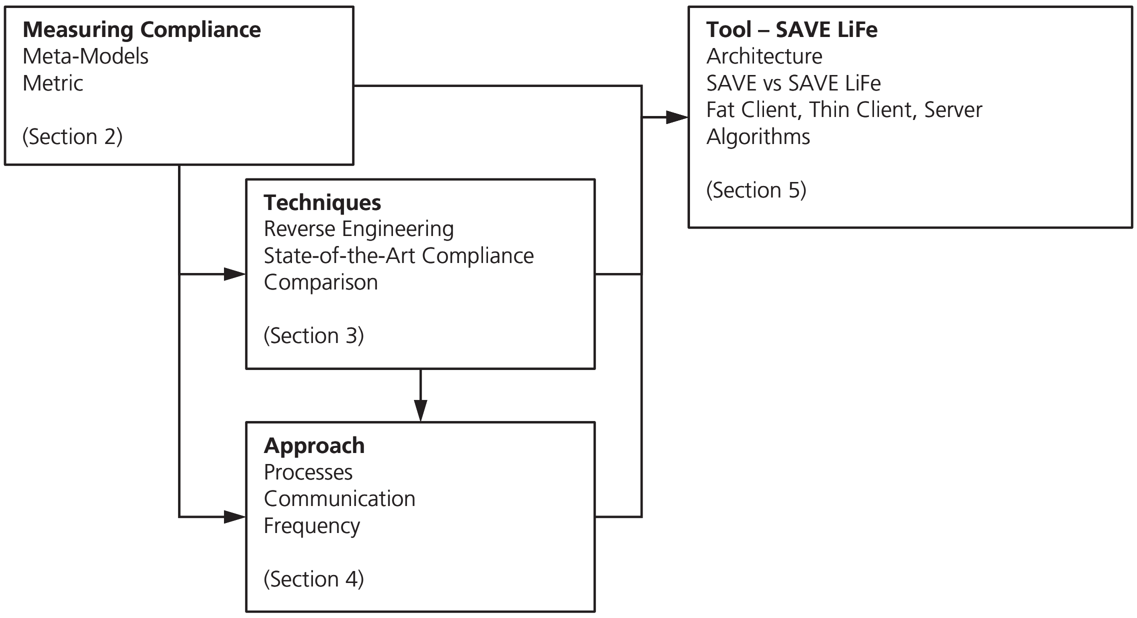


Figure 8 Overview Proposed Solution – Scientific (left) and Engineering (right) Building Blocks

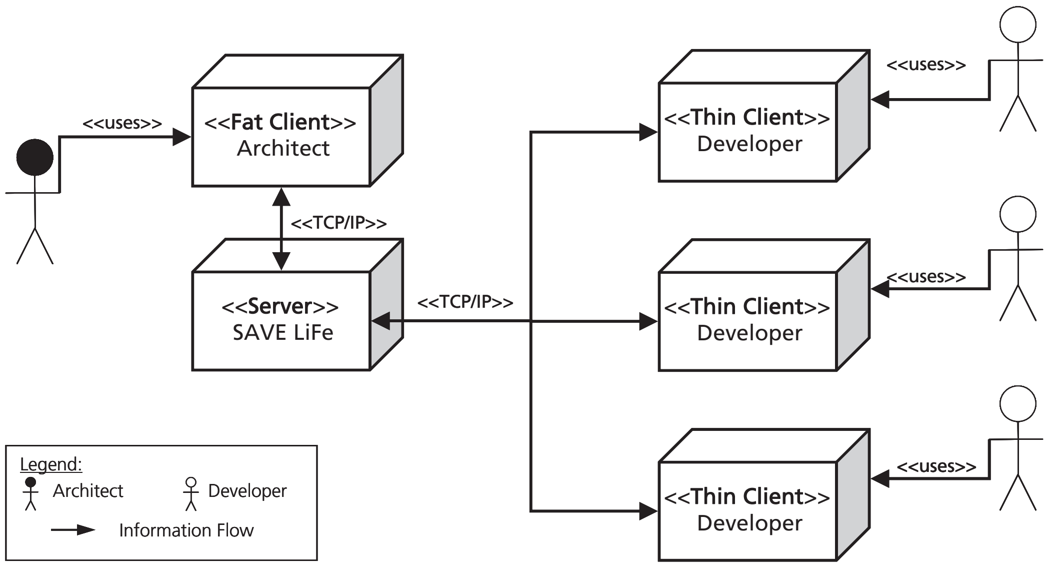


Figure 9 Solution – SAVE LiFe: Server, Fat Client, and Thin Clients

Fraunhofer SAVE is an Eclipse plug-in for goal-oriented analysis, compliance checking, and optimization of implemented software architectures. SAVE is a joint development of Fraunhofer IESE (Institute for Experimental Software Engineering IESE in Kaiserslautern, Germany) and the Fraunhofer Center Maryland (Center for Experimental Software Engineering in College Park, Maryland, USA). It realizes analytical compliance checking for implementation snapshots. SAVE comprises a set of extractors, analyzers, and generators that extract information from system artifacts, perform an arbitrary kind of computation, visualize the results, or generate system artifacts.

The engineering contribution differentiates SAVE LiFe – the client-server variant – from its ancestor SAVE – the snapshot analysis tool. Moreover, Section 5 introduces how SAVE LiFe meets the essential requirements listed in Table 4. We further discuss the architecture of SAVE LiFe, including its deployment to a central server, to a fat client for architects, and to thin clients for developers. SAVE LiFe is a client-server variant as depicted in Figure 9. The server manages a centralized repository that stores the information needed to execute the compliance checking and provides the computational logic for executing the analysis. The fat client allows the architect to manage the structural architectural view and to define the structure that the implementation shall comply with. The developers' thin client enables the execution of compliance checking in real time and visualizes live feedback in the source code editor. Modifications made by the developers are sent to the server and the server corresponds with live feedback on the compliance of the modifications made. The clients perceive the feedback as live because of low response times due the analysis scope being limited to the local delta of the respective developer. Both clients (fat and thin) and server are fully integrated into the Eclipse [Eclipse 2009] development environment.

In short, SAVE LiFe enables automated live architecture compliance checking and empowers developers to counteract structural violations. Thus, it is an instrument for realizing software implementations with sustainable structures – provided that developers have pay attention to the live feedback they receive.

1.5 Outline

The introduction (i.e., this section) presented the primary idea and motivation of this thesis. We stated the research questions, derived hypotheses from it, and sketched the proposed solution. The remainder of this thesis is structured as follows.

Section 2 continues with the presentation of meta-models relevant for compliance checking. Based on these models, we define a generally applicable metric for compliance. Then Section 3 summarizes the state of the art of analytical compliance checking techniques, compares them, and shows their equivalence in terms of expressiveness. Section 4 discusses the necessary adaptations and new concepts enabling live compliance checking. Further, a theoretical model for effort savings due to live compliance checking is introduced. Then Section 5 presents the tool support for live compliance checking – SAVE LiFe (which is the acronym for Software Architecture Visualization and Evaluation with Life Feedback).

Section 6 presents the validation of the effects of feedback on compliance checking in one experiment and two industrial case studies. Finally, Section 7 concludes this thesis by summarizing the results achieved. We further discuss the limitations, open questions, and sketches for future work. This thesis ends with final remarks in retrospective.

2 Measuring Architecture Compliance

The scope of this thesis is to investigate the impact of the compliance of a software implementation with the structural view of the software architecture on the lifecycle. Therefore, we aim at making statements on the effects of compliance on the development effort. For this purpose, it is necessary to quantify compliance. In this section we define a metric that determines the degree to which the state of having accomplished required structural demands is realized in the software implementation.

The compliance metric (see Section 2.4) measures two factors: the internal composition (i.e., the degree to which a component is realized by the right set of compilation units) and external dependencies (i.e., the degree to which specified dependencies are realized by the compilation units of the implementation) of each individual components. Hence, the compliance of the overall implementation aggregates the single component values.

The metric definition builds on top of the meta-models of the two distinct aspects between which compliance is measured. Hence, we first introduce the meta-model of the structural view (see Section 2.1) and followed by a generic source code model (see Section 2.2). The generic source code model abstracts the compliance metric from concrete programming languages. While the structural model represents abstract architectural concepts, the source code model captures concrete implementations. To bridge the gap between these two models, we introduce the concept of mapping (see Section 2.3), which defines 1-to-many relations and links elements of one meta-model to the other. Hence, this mapping allows identification of the respective counterparts on each level.

To see the metric in action, we present several simple examples combining different architectures and their implementations and illustrating the measurement of compliance (see Section 2.5).

2.1 Meta-Model of the Structural View

The software architecture aggregates a set of architectural views to describe the fundamental organization of a system. Figure 10 (using the UML notation [UML 2008]) depicts a simplified meta-model of a software architecture. It presents the subset of elements of the IEEE standard 1471 “Recommended Practice for Describing Software Architecture” [IEEE-Std.1471 2000] related to architectural views.

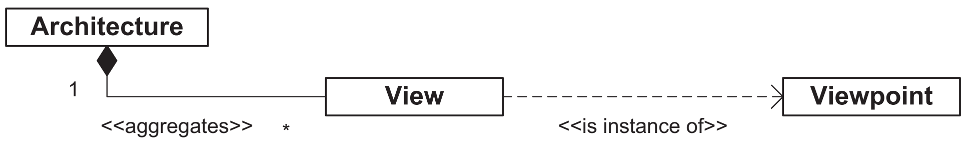


Figure 10 Simplified Meta-Model of Software Architecture

The structural view captures the static structure of a system in terms of layers, subsystems, and components, the interfaces provided by them, and the relationships between the various elements. The structural view only describes the static structure of a system and therefore does not provide any information about dynamic aspects and behavior.

The structural view (see Figure 11 for the meta-model) is represented using several structural models that decompose the system into architectural elements. Architectural elements have distinct responsibilities and encapsulate certain functionalities. To achieve their objective, architectural elements interact with other elements. These inter-element relationships enable the interplay of the architectural elements needed to eventually realize the functional and quality requirements of the system.

Figure 11 depicts the meta-model of the structural view using the UML notation [UML 2008]. An architectural element is a hierarchical entity that can contain other architectural elements. The architectural element is a generalization of concrete elements (e.g., layers, subsystems, components, or clusters; please note that Figure 11 only depicts the most commonly used elements). All architectural elements can act as containers and may contain other elements. Each architectural element can aggregate a set of inter-element relationships (e.g., dependency rules, connectors). An inter-element relationship links two architectural elements together. Table 5 explains the model elements of Figure 11 in more detail.

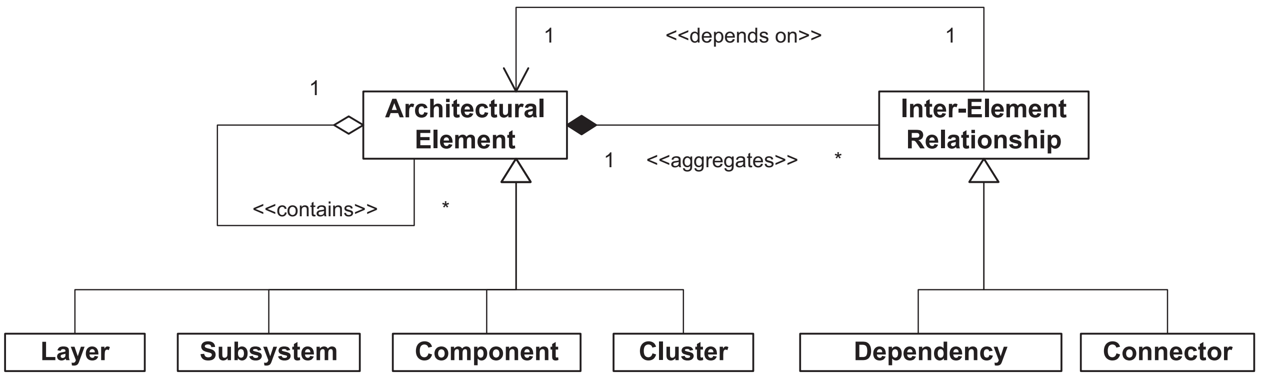


Figure 11 Meta-Model of the Structural View

Model Element	Description
Architectural Element	An architectural element is an abstract hierarchical container that can be instantiated by concrete elements.
Layer	Layering decomposes the structural view into several horizontal abstraction levels. A layer encapsulates functionality on different levels of abstraction (e.g., user interface, business logic, service, and hardware abstraction layer). Layering allows only top-down inter-element relationships. Strict layering enforces a strict hierarchy so that each layer is only allowed to use the layer directly below it.
Subsystem	A subsystem is a grouping element, which ideally has high cohesion and low coupling to other subsystems. A subsystem can contain either other subsystems or components.
Component	Components are the building blocks of a software system. Components have a predefined interface that encapsulates their internals. The internals of components realize the functionality the component provides. Single components comprise many source code elements, which implement the functional and quality requirement specified for the component.
Cluster	Clustering decomposes the structural view into several vertical clusters. Clustering allows elements in one cluster to only access other elements in the same cluster or non-clustered elements. Strict clustering enforces access only within a cluster. Clustering and layering are often used jointly to achieve vertical and horizontal decomposition.
Inter-Element Relationship	An inter-element relationship is an abstract, direct dependency from one architectural element to another that can be instantiated by concrete inter-element relationships.
Dependency	A dependency specifies how one architectural element may depend on another. It defines the type of access allowed (e.g., include directives, method invocations or function calls, read or write accesses to variables, inheritance, etc.). Dependencies may use regular expressions or patterns to define criteria matching a set of architectural elements.
Connector	A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components (e.g., shared representations, remote procedure calls, message-passing protocols, and data streams).

Table 5 Elements of the Structural View Meta-Model

The structural view of the architectures may have more than just one structural model, but all are an instantiation of the same meta-model. Having several structural models separates the different concerns of stakeholders: not all information is relevant to everyone. Furthermore, having only a single structural model yields one complex and complicated structural view of the architecture, without any encapsulation. To optimize clarity, readability ease of use, and maintenance, decomposition into several structural models is an appealing and preferable option. In practice, several but consistent structural models comprise the structural view of the system's software architecture.

2.2 Meta-Model of the Source Code

The source code of a software system consists of the written statements implemented by the developers. The source code is written in a specific programming language prescribing the constructs that developers can use to realize solutions for algorithms, data structures, and so on. Figure 12 (using the UML notation [UML 2008]) depicts the generic source code meta-model.

The source code model captures the static structure of a system at development time. In contrast to the structural view, the source code model comprises a number of elements that is entire orders of magnitude higher. These source code elements are the key of the generic source code model. Through their genericity, they represent corresponding code elements spanning many different programming languages. Hence, any programming language requires interpretation in order to assign the programming language constructs to the elements of the generic source code model.

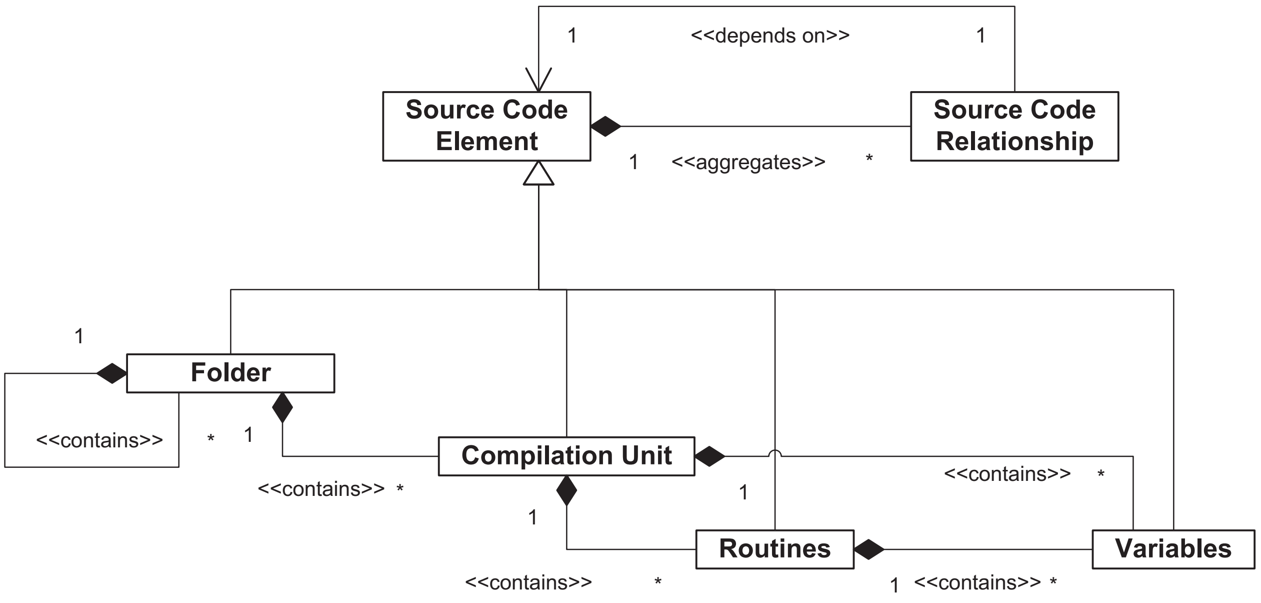


Figure 12 Meta-Model of the Source Code

Source code elements are interconnected by source code relationships. A source code relationship is a directed connection from one concrete code element to another. Again, the concrete dependencies implementable by programming language constructs have to be assigned to the generic source code relationship. Table 6 explains the model elements of Figure 12 in more detail.

Model Element	Description
Source Code Element	A source code element is an abstract representation that can be instantiated by concrete elements.
Folder	A folder represents either a grouping element that is visible in the file system (i.e., a directory) or the constructs of the programming language (e.g., packages in Java). Folders are hierarchical elements that can contain other folders, compilation units, or both.
Compilation Unit	Compilation units are the source code elements that are written by the developers. Compilation units are distinct elements that are processed individually by the compiler. Compilation units represent classes or files. They contain routines or variables.
Routines	Routines are closed fragments of source code within compilation units, which perform specific tasks and have a predefined signature (i.e., routines can have parameters and return values). Dependent on the programming language, routines are often referred to as subroutines, functions, methods, procedures, or subprograms. Routines are executed (i.e., called by or invoked by) by other routines.
Variables	Variables represent identifiers in the source code. They are symbolic representations used to bind a variable to a memory location. The variable stores values of a data object in that location so that the object can be accessed and manipulated at a later point in time.
Source Code Relationship	A source code relationship represents a dependency from one concrete source code element (e.g., see above) to another. The type of relationship depends on the capabilities of the programming language (e.g., in Java, it is possible to implement imports, method invocations, variables accesses, inheritance, etc.). Source code relationships are directed, which means they have an origin and a target.

Table 6 Elements of the Source Code Meta-Model

Due to its genericity, the source model is instantiated for specific programming languages. Such an instantiation may involve the addition of further concrete and programming language-specific elements; however, the meta-model depicted in Figure 12 has been sufficient when analyzing software systems implemented in programming languages like Java, C/C++, or Delphi. Table 7 presents the assignment of programming language constructs to the generic source code model. Here, we chose two representatives, Java for object-oriented and C for procedural languages. As Table 7 shows, all relevant language constructs can be assigned to the meta-model. However, it is possible to filter the source code model and focus only on a limited set of elements (e.g., not delving into details and having all source code relationships go out from folders or compilation units).

Model Element	Language Constructs in Java	Language Constructs in C
Folder	Java Package	Files System Directory
Compilation Unit	Java Class	C Implementation File (.c) C Header File (.h)
Routines	Class Method	Function
Variables	Class Instances Global Variables Local Variables	Global Variables Local Variables
Source Code Relationships	Class Import Class Inheritance Class Instance Access Method invocation Variable Access Interface Implementation	Header Include Function Implementation Function Call Variable Access

Table 7 Assignment of Programming Language Construct to Source Code Model Elements

2.3 Meta-Model of the Mapping

Architectures do not prescribe the structure in full detail; they rather provide a sketch and the rules that define how the architectural elements and their inter-element relationships should be translated into source code. Hence, the structural view captures the decomposition from a global system perspective. However, local decisions (i.e., details on the source code level) are still made by the developers. So it is not surprising that functionally equivalent systems realized based on the same architecture but coded by two different developers most likely yield two different implementations.

To confine the diversity in implementation, there are typically mapping instructions for developers. Such instructions prescribe how to name the source code elements and hierarchically structure them. Ideally, the structural models would be clearly reproduced by the hierarchy of folders and compilation units. Due to the abstraction gap, many source code elements represent one architectural element, respectively the same holds for relationships. In forward engineering [Chikofsky 1990], the developers create and name source code elements based on these mapping instructions – and create the *internal composition* of architectural elements. Examples of mapping instructions are naming conventions like prefixes for all compilation units that encode the component name, representations of each subsystem as a distinct folder in the file system, or distinctive mapping models designing and detailing an arbitrary decomposition of architectural elements. One of the responsibilities of an architect is to provide developers with these mapping instructions.

In cases where the mapping instructions are outdated, obsolete, or unknown, remedy comes from the field of reverse engineering [Chikofsky 1990]. The mapping instructions – often the whole architecture documentation, too – have to be extracted from the source code because the documentation was lost, underwent significant changes so it is no longer possible to match the mapping instructions with the implementation, or the documentation never existed at all. The analysis of existing software systems with the aim of recovering architecture-relevant information from artifacts is broad field of research – mostly referred to as architecture reconstruction (please refer to [Koschke 2005], [Knodel 2006b] and [Pollet 2007] for an overview).

By implementing architectural elements such as components, the source code is produced. Developers write many new source code files and modify existing ones; in other words, they fill an initial skeleton implementation with content. In doing this, developers use other source code files; hence, they create directed dependencies among the compilation units. The location of origin and target compilation unit classifies the relationship either as an internal or as an *external dependency*. Internal dependencies remain within a component, while external dependencies realize a relationship between two architecture elements such as components (of course, there might be many concrete instances for the relationship on the code level).

Documenting how to bridge the gap between structural view and source code is the purpose of the mapping. Mapping (see Figure 13 for the meta-model using the UML notation [UML 2008]) defines the relationship between architectural elements and source code elements and vice versa. The same holds for architectural inter-element relationships and source code relationships. Typically, one entity of the architectural level is represented by numerous entities on the source code level. The mapping has been defined as a distinct meta-model to achieve a clear separation of the two abstraction levels – the structural model and the source code model remain independent of each other, and there are no direct dependencies from one model to the other.

The mapping comprises all information needed to bridge the abstraction level gap from architecture to source code. Information about the respective counterparts can be extracted from the references of the element mapping and the relation mapping. Element mapping links architectural elements to source code elements. The extraction of corresponding counterparts is possible for both directions. The relation mapping provides the same link between architectural inter-element relationships and source code relationships. Table 8 explains the model elements of Figure 13 in more detail.

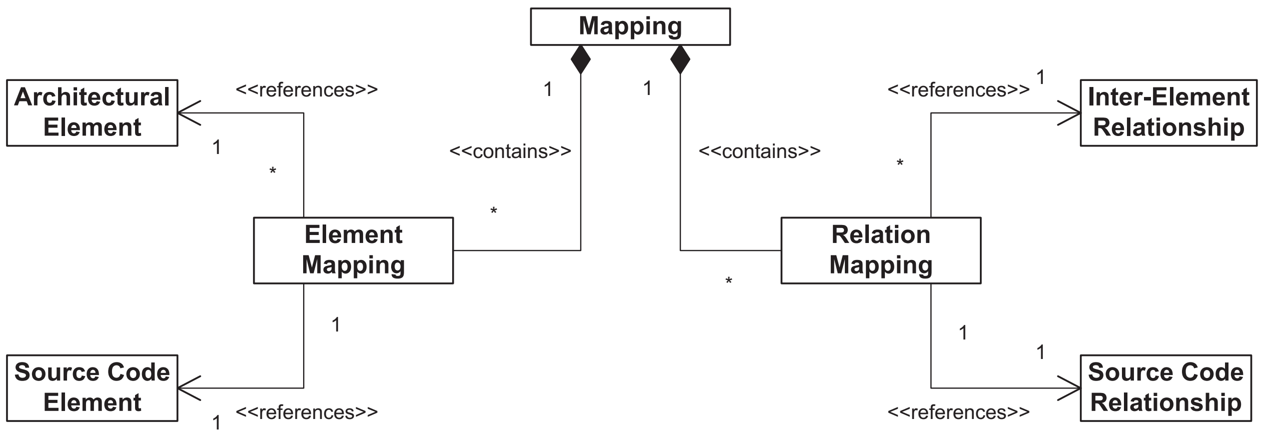


Figure 13 Meta-Model of the Mapping

Model Element	Description
Mapping	The mapping is a container for all mapping, either element mappings or relationship mappings. Mapping is the root to bridge the abstraction gap mapping between the architecture and the source code.
Element Mapping	The element connection links one architectural element to one source code element.
Architectural Element	See description above.
Source Code Element	See description above.
Relation Mapping	The relation mapping links one architectural element to one source code element.
Inter-Element Relationship	See description above.
Source Code Relationship	See description above.

Table 8 Elements of the Mapping Meta-Model

Having defined the meta-models for structural views, source code, and their mapping allows specifying the compliance metric. This metric measures compliance for concrete instances of the respective meta-models.

2.4 Compliance Metric

The compliance metric is a measure that is relative to exactly one set of one instance each of structural model, source code model, and mapping, respectively.

The compliance metric compares the architecture specified with the implementation realized by developers. This section first introduces basic formulas, and then continues with the formalization of inputs and the definition of operators. Finally, we define the compliance function, which aggregates measures from individual architectural elements.

2.4.1 Basic Formulae

Because we compare given classifications of an item (i.e., the realization of source code elements and source code relationships) with desired correct classifications (i.e., the specification of architectural elements and inter-element relationships), we can apply two measures from statistical classification. In this context, the comparison results in assignment to one of the following categories:

- **True positives:** specification equals realization
- **False positives:** realized but not specified
- **True negatives:** specified but not realized

To obtain a single measure representing the comparison results, we can compute the harmonic mean of precision and recall for the specification and realization, which is defined as their F-Measure [Frakes 1992].

Definition 14 Precision

Precision measures the degree of correctly realized elements among all realized elements.

$$precision = \frac{tp}{tp + fp}$$

where tp stands for true positive, and fp for false positive.

Definition 15 Recall

Recall measures the degree of correctly realized elements among all specified elements.

$$recall = \frac{tp}{tp + tn}$$

where tp stands for true positive, and tn for true negatives.

Definition 16 F-Measure

The F-Measure is the standard combination of precision and recall, defined as their harmonic mean.

$$F - Measure = \frac{2 * precision * recall}{precision + recall}$$

2.4.2 Formalization of Metric Input

Software systems comprise the architecture and the implementation.

Definition 17 Formalization of Software System

Let the software system be a tuple of the architecture, the implementation, and the function f_{map}, which enables traceability between the abstract architecture and the concrete implementation and defines the relationship between architectural elements and source code elements.

$$SYS = (A, I, f_{map})$$

where SYS stands for the software system, A for architecture, I for implementation, and f_{map} for the mapping function. Note that we explicitly ignore other system artifacts produced in the lifecycle of the software system.

The structural model (representing the architecture), source code model (representing the implementation) are mandatory inputs required to measure compliance.

Definition 18 Formalization of Architecture

Let the structural model of the architecture be a set of architectural elements with a set of inter-element relationships among them.

$$A = (S_{AE}, S_{IER})\ with\ S_{IER} \subseteq S_{AE}$$

where A stands for the architecture, S_{AE} for the set of all architecture elements with $S_{AE} = \{AE_1, \ldots, AE_i , AE_j ,\ldots, AE_n\}$, where AE stands for architectural element, S_{IER} for the set of all inter-element relationships, with $S_{IER} = \{IER, IER_{ij}\}$, where IER stands for an inter-element relationship between two architectural elements AE_i and AE_j.

Definition 19 Formalization of Implementation

Let the source code model of the implementation be a set of source code elements with a set of source code relationships among them.

$$I = (S_{SCE}, S_{SCR})\ with\ S_{SCR} \subseteq S_{SCE}$$

where I stands for the implementation, S_{SCE} for the set of all source code elements with $S_{SCE} = \{SCE_1, \ldots, SCE_i , SCE_j ,\ldots, SCE_n\}$, SCE for a source code element, S_{SCR} for the set of all source code relationships with $S_{SCR} = \{SCR, SCR_{ij}\}$, where SCR stands for an source code relationship between two source code SCE_i and SCE_j.

2.4.3 Lifting and Mapping Operator

Because the structural model and the source code model are on different levels of abstraction, we have to define a lifting operator and a mapping operator in order to bring the two models to the same level of abstraction. Hence, lifting and mapping are the tools for bridging the gap. They operationalize lifting source code model elements and relationship to the level of the architecture and vice versa.

Definition 20 Lifting Operator

The lifting operator f_{lift} lifts a given implementation to the abstraction level of the architecture using the mapping. The lifting operator can be executed for source code elements and source code relationships.

$$f_{lift}(S_{SCE} \cup S_{SCR}) \rightarrow (S_{AE} \cup S_{IER}) \textit{ with}$$

$$f_{lift}(b) \in \begin{cases} S_{AE};\ b \in S_{SCE} \\ S_{IER};\ b \in S_{SCR} \end{cases} \textit{or}$$

$$f_{lift}(S_{SCE}) \subseteq S_{AE}$$

$$f_{lift}(S_{SCR}) \subseteq S_{SCR}$$

where f_{lift} stands for the lifting operator, which defines the relationship between source code elements and architectural elements, and relationships respectively.

Definition 21 Formalization of Mapping

Let the mapping be an is-part-of relation between source code elements and architectural elements.

$$f_{map} = f_{lift}^{-1}$$

where f_{map} stands for the mapping function, which defines the relationship between architectural elements and source code elements, and relationships respectively.

Figure 14 illustrates the mode of operation of the lifting operator. The implemented architectural elements aggregate many concrete source code elements. This aggregation is depicted by the shadow of the architectural elements, which covers the source code elements resolved from the mapping. Further, the inter-element relationships aggregate the concrete source code relationships. Of course, relationships within one architectural element are possible, too.

Applying the lifting operator to the source code of an implementation I_{impl} produces one model – the implemented or realized architecture A_{impl}. This model is now on the same level of abstraction as the structural model – the specified architecture A_{spec}.

Definition 22 Lifting Operator Application

Applying the lifting operator f_{lift} to a given implementation I yields the implemented architecture.

$$\begin{aligned} A_{impl} &= f_{lift}\ (I_{impl}) \\ &= f_{lift}\ ((\{SCE_1\ \dots\ SCE_n\}), (\{SCR_1\ \dots\ SCR_n\})_{impl}) \\ &= (\{AE_1\ \dots\ AE_n\}), (\{IER_1\ \dots\ IER_n\})_{impl} \end{aligned}$$

where A_{impl} stands for one specific implemented architecture, and I_{impl} for one specific implementation.

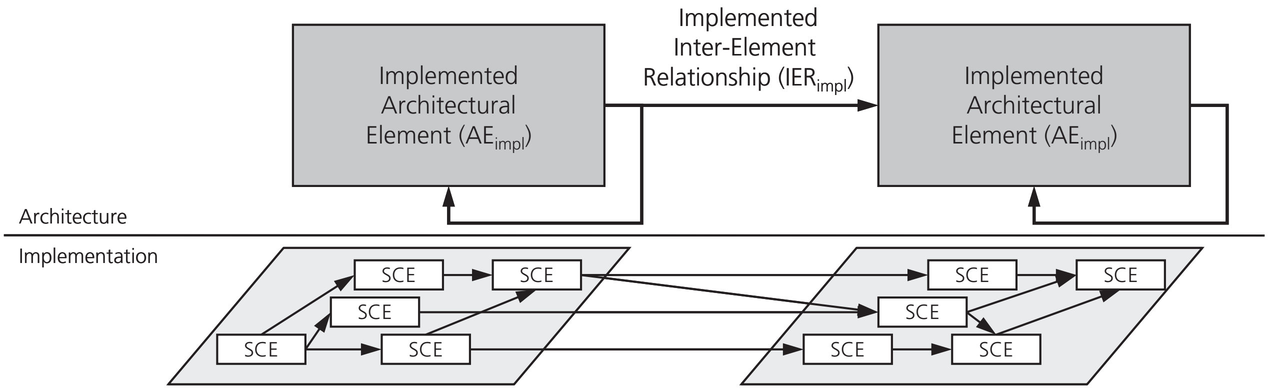

Implemented Inter-Element Relationship (IER$_{impl}$)
Implemented Architectural Element (AE$_{impl}$)
Implemented Architectural Element (AE$_{impl}$)
Architecture
Implementation
SCE

Figure 14 Illustration Lifting Operator

2.4.4 Compliance Function

The aim of the compliance function is to compare the specification with the realization.

Definition 23 Architecture Compliance Function

The architecture compliance metric is a function that computes a value between 0% (zero compliance) and 100% (full compliance) for a given specification and a given realization. The compliance function is the normalized arithmetic mean of the element compliance function, which computes the compliance for one architectural element.

$$\begin{aligned} AC &= f_{AC}\left(A_{spec}, A_{impl}\right) \\ &= \sum_{i=1}^{n} w_i^{size} * EC_i \quad with\ AE_i \in (A_{spec} \cup A_{impl}) \end{aligned}$$

where AC stands for architecture compliance, A_{spec} for one specific specified architecture, A_{impl} for one specific implemented architecture, EC for element compliance, and w^{size} is a weighting factor based on the size of an architectural element.

The architectural compliance metric is an aggregate of the individual measures for each individual architectural element – this measure we call the element compliance. The single architectural elements are normalized by means of a weighting factor w^{size}, which allows assigning a relative weighting based on an arbitrary size metric. The weighting factor can either be computed using code metrics (e.g., lines of code, number of routines, number of compilation units), complexity metrics (e.g., cyclomatic complexity [McCabe 1976], component complexity [Henry 1981]), or be assigned by the architect according to the subjective importance of the element. The sum of all weights combined has to be equal to one (hence, the default weighting factor 1/n expresses equal weighting among the elements).

Definition 24 Element Compliance Function

The element compliance metric is a function that computes a value between 0% (zero compliance) and 100% (full compliance) for a given specification and a given realization of an architectural element (i.e., the implementation). The element compliance is the product of the internal composition IC (i.e., it captures to which degree the planned decomposition structure matches the actual one) and the external dependencies ED (i.e., it captures to which degree the planned dependencies match the actual ones).

$$\begin{aligned} EC &= f_{EC}\left(AE_{spec}, AE_{impl}, IER_{spec}, IER_{impl}\right) \\ &= IC\left(AE_{spec}, AE_{impl}\right) * ED\left(IER_{spec}, IER_{impl}\right) \end{aligned}$$

where AE_{spec} stands for a specified architectural element, AE_{impl} for an implemented architecture element, IER_{spec} for a specified inter-element relationship, IER_{impl} for an implemented inter-element relationship, for IC for internal composition and ED for external dependencies.

The element compliance function is the product of two factors: the internal composition and the external dependencies.

2.4.5 Internal Composition

The internal composition of architectural elements captures the degree to which the mapping instructions for the source code elements (i.e., the counterparts as specified by the architect) have been implemented correctly by the developers. There are three categories for internal composition:

- **Correct composition (true positives):** The implemented architectural element has been composed as specified by the architect (i.e., AE_{spec} is equivalent to AE_{impl}). There are only correctly placed source code elements in the implementation of the architectural elements and the containment hierarchy of the source code element reflects the mapping instructions.

- **False composition (false positives):** The implemented architectural element violates the specification. Source code elements have been misplaced in the implementation of the architectural element (e.g., code elements belonging to a different architectural elements) or the decomposition structure does not adhere to the mapping instructions (e.g., naming convention violated).
- **Missing composition (true negatives):** There were no implemented architectural elements (i.e., no source code elements could be lifted) for the specified architectural elements.

The definition of true positives, false positives, and true negatives allows calculating the F-Measure for the internal composition.

Definition 25 Internal Composition

The internal composition IC captures the degree to which the planned decomposition structure matches the actual one.

$$IC\left(AE_{spec}, AE_{impl}\right) = F\text{-}Measure\ \left(AE_{spec}, AE_{impl}\right)$$
$$= \frac{2 * precision * recall}{precision + recall}$$

where IC stands for the harmonic mean of precision and recall of the comparison of the specified structure with the realized structure.

The architect has to decide whether or not architectural elements are well-composed. Hence, subjective expert ratings are required to identify flaws in the source code model containment (i.e., logically misplaced elements or poorly structured decomposition).

2.4.6 External Dependencies

The external dependencies of architectural elements capture the degree to which the specified architectural inter-element relationships have been realized by the developers. There are three categories for external dependencies [Murphy 2001]:

- **Convergent dependency (true positives):** a relationship between two architectural elements that was implemented as specified (i.e., IER_{spec} is equivalent to IER_{impl}).
- **Divergent dependency (false positives):** a relationship between two architectural elements that was implemented but not specified.
- **Absent dependency (true negatives):** a relationship between two architectural elements that was specified but not implemented.

The definition of true positives, false positives, and true negatives allows calculating the F-Measure for the external dependencies.

Definition 26 External Dependencies

The external dependencies ED capture the degree to which the planned dependencies match the actual ones.

$$ED\left(IER_{spec}, IER_{impl}\right) = F\text{-}Measure\left(IER_{spec}, IER_{impl}\right) = \frac{2 * precision * recall}{precision + recall}$$

where ED stands for the harmonic mean of precision and recall of the comparison of the specified dependencies with the realized dependencies.

By lifting source code elements to the level of the software architecture using the mapping, the external dependencies are lifted as well. Hence, objective, automated measurement is possible.

2.4.7 Summary

We defined the compliance metric as a composed measure comprising several aspects. Figure 15 summarizes the compliance metric graphically. Architecture compliance is the weighted average element compliance of all elements the system is built of. The individual elements' compliance investigates the internal composition on the one hand and the external dependencies on the other hand.

Both are computed by using the harmonic mean of precision and recall. Precision measures the degree to which correctly implemented elements or dependencies are found in the implementation, while recall measures the degree to which correctly implemented elements or dependencies are specified. For internal composition, precision computes the degree of correctly composed elements among all present elements (i.e., correctly and falsely placed elements) and recall the degree of correctly composed elements among all required elements (i.e., correct and missing elements). For external dependencies, precision computes the degree of convergent dependencies among all existing dependencies (i.e., convergences plus divergences) and recall the degree of convergences among all specified relationships (i.e., convergences plus absences).

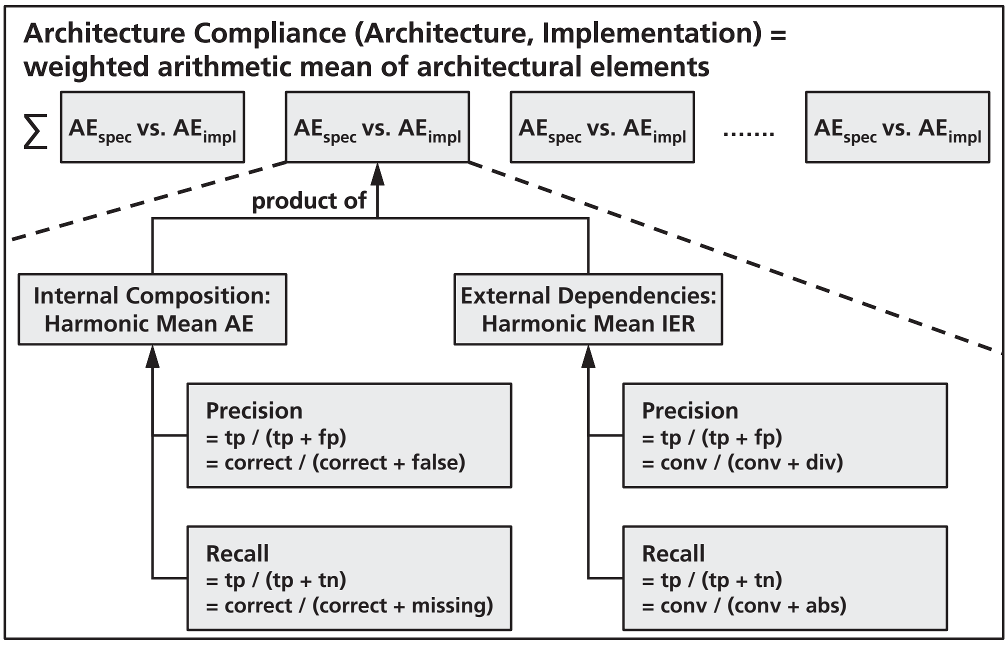

Architecture Compliance (Architecture, Implementation) = weighted arithmetic mean of architectural elements
$\sum$
AE_{spec} vs. AE_{impl}
AE_{spec} vs. AE_{impl}
AE_{spec} vs. AE_{impl}
....
AE_{spec} vs. AE_{impl}
product of
Internal Composition: Harmonic Mean AE
External Dependencies: Harmonic Mean IER
Precision = tp / (tp + fp) = correct / (correct + false)
Recall = tp / (tp + tn) = correct / (correct + missing)
Precision = tp / (tp + fp) = conv / (conv + div)
Recall = tp / (tp + tn) = conv / (conv + abs)

Figure 15 Compliance Metric

After defining the compliance metric, we now illustrate the metric in action. For this purpose, we use a simple example system where we vary architecture and implementation.

2.5 Metric Examples

This section presents several typical scenarios of a small hypothetical system with variations. For each variation, we calculate its architecture compliance.

All examples are based on the same structural view – a three-layered architecture (see Figure 16). The layering is strict meaning that each layer is only allowed to use and access the layer directly below (i.e., *Layer-1* can use *Layer-2*, *Layer-2* can use *Layer-3*; all other relationships are forbidden). The architect further defined mapping instructions that all source code elements have to encode the layering information. The example implementations should reflect this layering through a corresponding decomposition structure. The example implementations were created in the Java programming language.

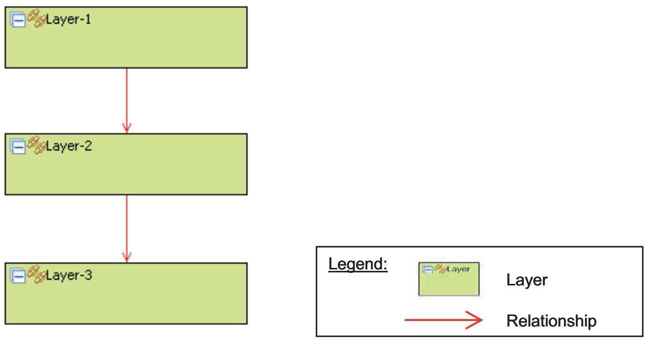


Figure 16 Structural View Specifying Three Layers Enforcing Strict Layering

2.5.1 Scenario: Beginning of Implementation

At the beginning of the implementation, as depicted in Figure 17, source code does not exist yet. For this reason, the internal composition and external dependency values for the architectural elements are both zero. Hence, the overall **architecture compliance is 0%**.

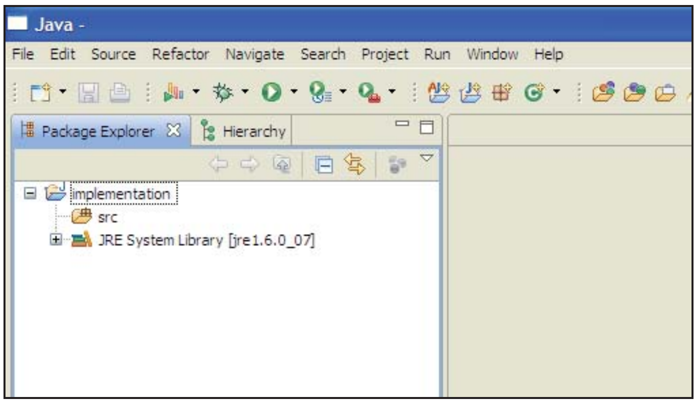
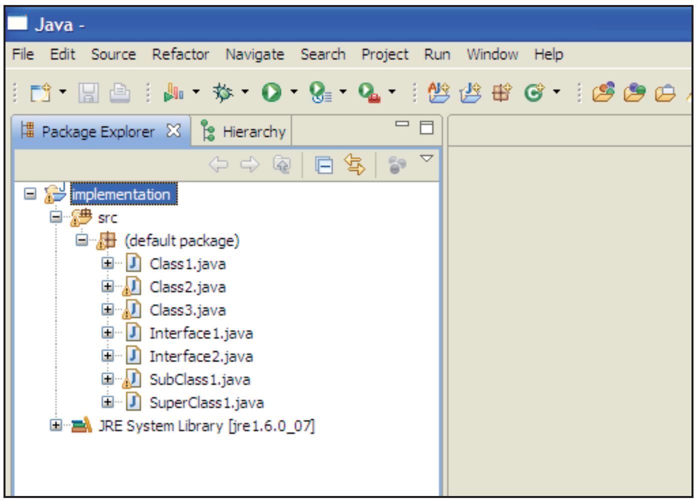

Java -
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
implementation
src
JRE System Library [jre1.6.0_07]

Figure 17 Scenario: Beginning of Implementation

2.5.2 Scenario: Composition Flaw

Figure 18 shows one implementation variant where all source code files have been composed into the same default Java package. The composition of source code elements does not reveal any structure, which indicates that the developers ignored the mapping instructions formulated by the architect. Hence, the overall **architecture compliance is 0%**, too.

Because none of the source code elements has been composed correctly according to the mapping instructions, the internal composition of each architectural element is zero. This example might be exaggerating but misplacements or poorly structured source code element hierarchies can be observed in practice, as the discussion of the case studies has shown. Furthermore, according to [Lehman 1985], initially well-structured implementations degenerate over time if no counteractive measures (e.g., refactoring or restructuring) are taken to prevent the structural decay.

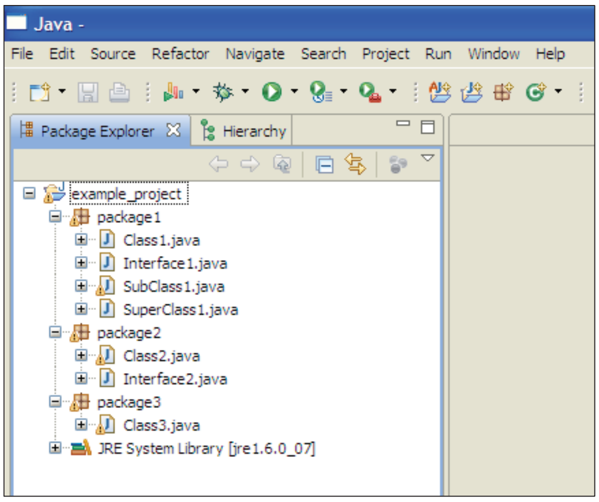

Java -
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy

```
implementation
  src
    (default package)
      Class1.java
      Class2.java
      Class3.java
      Interface1.java
      Interface2.java
      SubClass1.java
      SuperClass1.java
  JRE System Library [jre1.6.0_07]
```

Figure 18 Scenario: Composition Flaw

2.5.3 Scenario: Integration Flaw

The integration flaw scenario illustrates in Figure 19 a well-structured system where the internal composition is clear. Each layer has been realized in a distinct package, where the number of the package indicates the layer the package is representing. The internal composition for this implementation variant is 1 for all architectural elements. However, there is an integration flaw, which Figure 19 captures by showing only layer-internal dependencies as convergences and the required external dependencies as absences.


Java -
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy

```
example_project
  package1
    Class1.java
    Interface1.java
    SubClass1.java
    SuperClass1.java
  package2
    Class2.java
    Interface2.java
  package3
    Class3.java
  JRE System Library [jre1.6.0_07]
```

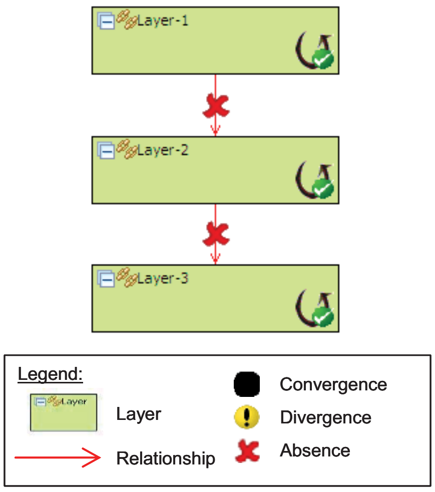


Figure 19 Scenario: Integration Flaw

Computing the element compliance for all three layers now results in the following degrees of compliance: *Layer-1* = 66%, *Layer-2* = 66%, and *Layer-3* = 100%. Note that we considered the internal dependencies within a layer as convergent true positives resulting in a precision of 1 and a recall of 0.5 for the layers *Layer-1* and *Layer-2*. Hence, for this implementation variant, the overall **architecture compliance is 77%**.

2.5.4 Scenario: Unplanned Growth

This scenario captures the case when the system is growing in an unplanned way. New architectural elements outdate the architecture documentation and cause a drop in compliance. Figure 20 depicts an example where the architectural element *Layer-4* is part of the implementation but is not specified in the structural view (see Figure 16). The internal composition of all three specified layers equals 1, which holds as well for the external dependencies of *Layer-1* and *Layer-2*. The element compliance for both *Layer-1* and *Layer-2* is 1.0. However, *Layer-3* comprises a violating use of *Layer-4*, which causes an external dependency value of 0.66 resulting in a element compliance of 0.66 as well. *Layer-4* as an unspecified element scores zero in its element compliance. Aggregating the values for the four architectural elements using the arithmetic mean results in an overall **architecture compliance of 66%**.

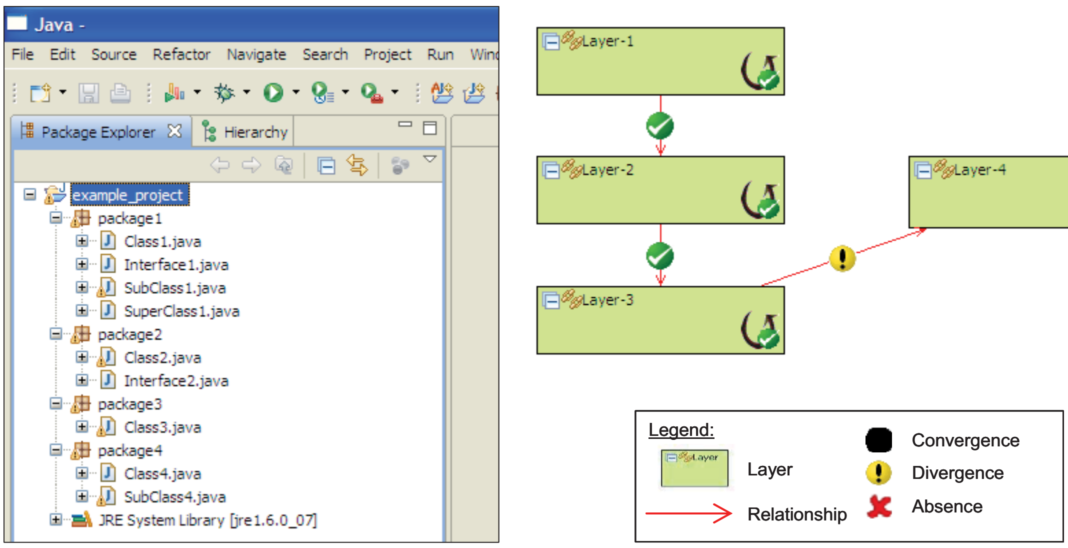
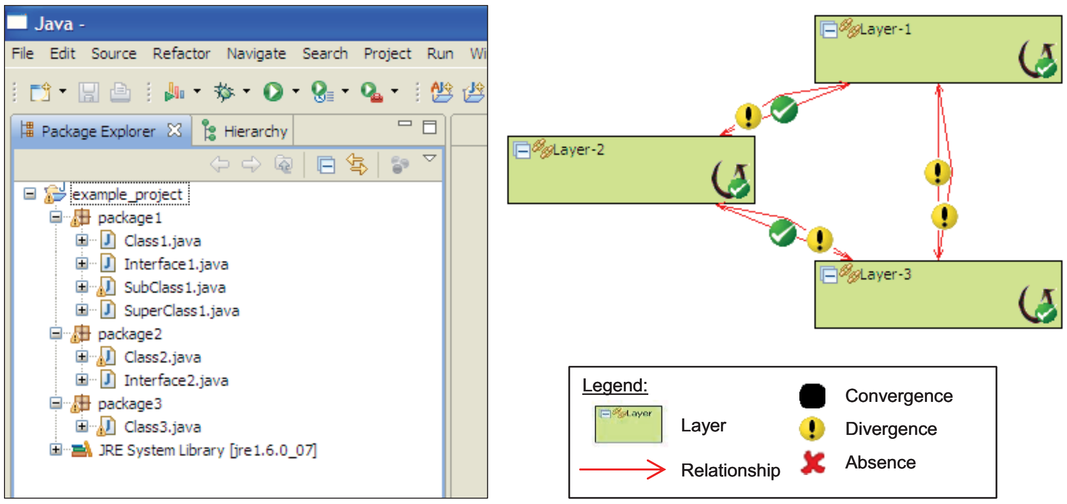


Figure 20 Scenario: Unplanned Growth

2.5.5 Scenario: Unplanned Interdependencies

The scenario with unplanned interdependencies is one of the most typical scenarios we observed in industrial practice. Although the internal composition of each architectural element has been realized correctly (IC is 1 for all elements), the elements are heavily coupled (as depicted in

Figure 21). There are a lot more actual dependencies among the architectural elements than originally planned. The values for external dependencies are 0.8 for *Layer-1* and *Layer-2*, where in both cases precision is 0.66 (2 convergences and 1 divergence) and recall is 1.0. In contrast, *Layer-3* has a value of 0.4 (1 convergence but 2 divergences). Aggregating the individual element compliance results yields an overall **architecture compliance of 66%**.

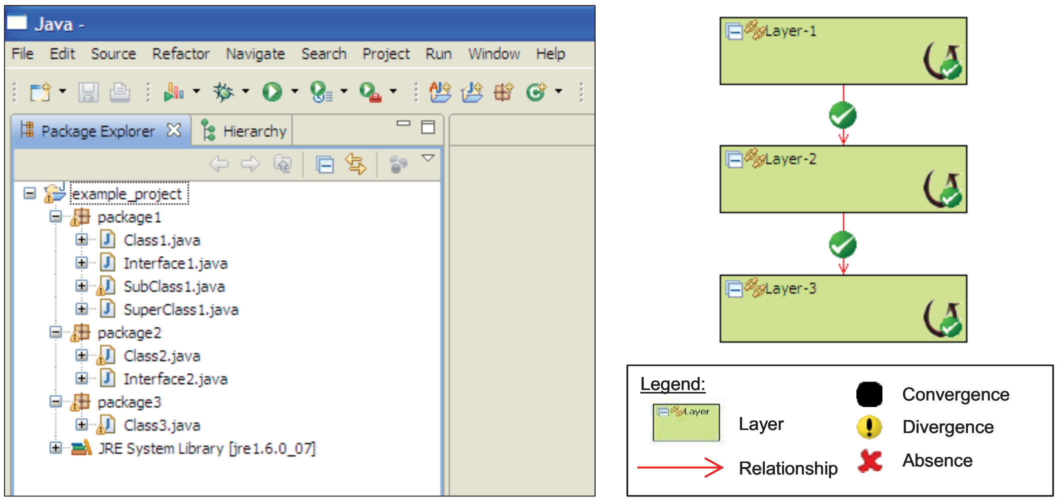


```
example_project
  package1
    Class1.java
    Interface1.java
    SubClass1.java
    SuperClass1.java
  package2
    Class2.java
    Interface2.java
  package3
    Class3.java
  JRE System Library [jre1.6.0_07]
```

Figure 21 Scenario: Unplanned Interdependencies

2.5.6 Scenario: Architecture-Compliant System

Figure 22 shows the rare case of an implementation that fully complies with its architecture. All architectural elements and the inter-element relationships have been realized as intended by the architect and hence, the overall **architecture compliance is 100%**.



```
example_project
  package1
    Class1.java
    Interface1.java
    SubClass1.java
    SuperClass1.java
  package2
    Class2.java
    Interface2.java
  package3
    Class3.java
  JRE System Library [jre1.6.0_07]
```

Figure 22 Scenario: Architecture-Compliant System

2.6 Conclusions

This section introduced the metric for measuring architecture compliance based on the meta-model of the structural view, the source code, and the mapping. The example with the structural view comprising strict layering and the different implementations illustrates how the architecture compliance metric works (assuming equal weighting).

The architecture compliance metric allows quantifying the degree to which the implementation of the system has been realized as specified. Obviously, full architecture compliance (i.e., compliance equaling 100%) is the optimal case. In practice, however, this optimal case is rarely achieved. Our experience with industrial partners shows that when compliance measurement is institutionalized and communicated as a clear development goal, compliance of up 98-99% can be achieved (e.g., see [Knodel 2008b]).

Experiences with compliance checking projects at Fraunhofer IESE has consolidated a rule of thumb: Compliance lower than 95% calls for special attention. It constitutes a major threat for the development organization. The 95% threshold is an indicator of systemic problems that may eventually affect the entire system. This status requires special activities, which include in-depth analyses of root causes or explicit planning and effort investments for refactoring activities to repair the implementation. Often, a compliance drop below the 95% threshold leads to even further structural degeneration (i.e., more and more violations over time). Development organizations have to identify the root cause for the lack of compliance (e.g., inadequate architectural solution, inadequate documentation of solution, lack of quality assurance, or insufficient education of software developers). However, to date, there is no empirical confirmation for this 5% threshold yet and so this rule of thumb has to be considered with caution.

Measuring compliance verifies only that the specified plan of the system matches the implemented facts. Note that the verification of compliance does not allow any other statements about the appropriateness of the architecture as a whole with respect to the envisioned development goals, functional requirements or quality requirements.

3 Compliance Checking Techniques

Analytical compliance checking techniques detect structural violations in the source code. They aim at verifying that the structural decomposition is in place and point to the violating source code statements.

This section shows that the most popular compliance checking techniques – the Reflexion model and dependency rules – are equivalent in expressiveness (see Section 3.3). Thus, independent of the concrete technique applied, compliance checking produces the same results because dependencies rules can be transformed into Reflexion models and vice versa (although the transformation may become a tedious task).

So in general, both techniques offer themselves as candidates for adaptation and usage as base technology in live compliance checking. For this reason, we investigated the applicability of these two techniques in several dimensions determined by the envisioned usage goal as a quasi-constructive quality engineering technique (see Section 3.4). A thorough analysis of advantages and drawbacks reveals that Reflexion models outscore dependency rules in their suitability for the intended use. In particular, the characteristics explicitness, ease of use, ease of learning, and low probability of false positives are clear advantages of Reflexion models. They make it possible to educate developers on the intended structure of the software implementation.

We continue with the derivation of the key principle of live compliance checking (see Section 3.5). Using the Deming cycle [Deming 1986], we distinguish the new live approach from its analytical siblings for system snapshots. The paradigm shift executes the checking not at distinct points in time but continuously and quasi-constructively while developers are writing the code.

The explanation of the underlying concepts and the mode of operation of both Reflexion models and dependency rules first introduces the generic reverse engineering archetype – extraction, abstraction, and presentation – to gain knowledge from existing artifacts (see Section 3.1).

We further discuss how the reverse engineering discipline fact extraction is applied in compliance checking (see Section 3.2). Fact extraction mines the source code for relevant information, which is then used by the checking technique to derive statement about the compliance.

3.1 Reverse Engineering

The ultimate goal of reverse engineering is to gain knowledge from existing artifacts. Knowledge is the dynamic capacity that enables a stakeholder to perform a task and to solve problems: "Knowledge means the confident understanding of a subject with the ability to use it for a specific purpose" [Wikipedia 2008]. Knowledge is always bound to individuals [Probst 1999] and is based on information, which in turn is based on data [Rus 2002].

Definition 27 Knowledge

Knowledge is the result of a learning process and can be seen as a function of (task-related) information, experience, skills and attitude at a given moment in time [Weggeman 1999].

Definition 28 Information

Information is data that is organized to make it useful for end users who perform tasks and make decisions [Rus 2002].

Definition 29 Data

Data consists of discrete, objective facts about events and entities but nothing about its own importance or relevance; it is raw material for creating information [Rus 2002].

Individuals create knowledge dynamically by interpreting the provided information units based on their own context, background, and experience. Therefore, knowledge is dependent on the individual and is tacit. Information on the other hand is independent of the individual and can be explicitly documented and thus, is easy to duplicate (see [Miller 2002] and [Sveiby 1997]). Stakeholders typically have the need to gain knowledge about a software system as a whole or about one specific aspect of the system to guide their decision-making. In particular, they require knowledge because they would like to resolve uncertainties, clarify unknown characteristics, or increase their level of confidence.

The basis for gaining knowledge is information on the subject – the software system. Reverse engineering supports this stakeholder goal by providing information – representations or abstractions (or both combined).

Definition 30 Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [Chikofsky 1990].

The reverse engineering archetype describes the typical steps of most reverse engineering processes (e.g., see [Müller 1994], [Mendoca 1996], [Ebert 2002]). The activities performed are named extraction, abstraction, and presentation – typically executed in iterative cycles, repeated and refined, if appropriate, until the intended goal has been obtained. Figure 23 depicts this reverse engineering archetype including the concluding activity – interpretation which illustrates how stakeholders use the results.

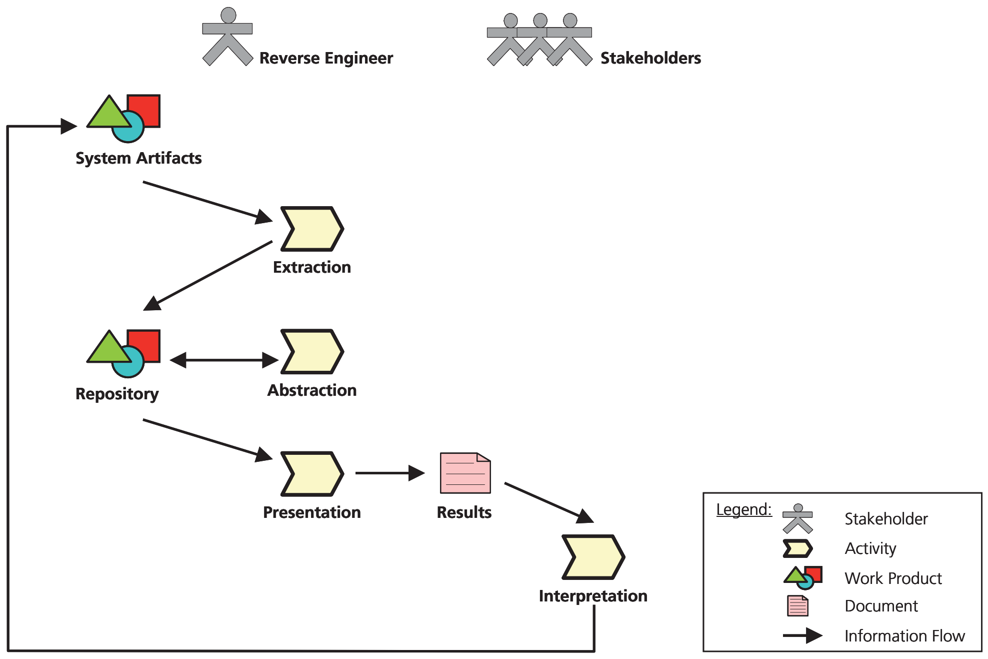


Figure 23 Archetype of Reverse Engineering

- **Extraction:** Extraction processes the raw data contained in system artifacts – these pieces of data are often called facts. A fact represents one basic piece of information about a software system (e.g., the source code comprises classes A and B, class A has a method A1, and method A1 calls method B1). Fact extraction spans manual inspection [Moonen 2002], lexical analysis (e.g., [Murphy 1996]) pattern matching (e.g., [Pinzger 2002], [Knodel 2003]), island grammars [Moonen 2001], configuration management systems (e.g., [Zimmermann 2005]), defect management system (e.g., [Fischer 2003]), and document analysis [John 2003]). Most popular, however, are static analyses [IEEE-Std-610.12 1990] based on source code or dynamic analyses [IEEE-Std-610.12 1990] based on run-time traces generated from instrumented system executions. Finally, all facts are aggregated in a repository, which is the foundation of all further analyses.
- **Abstraction:** Abstraction processes the raw data generated in the extraction and turns it into information. The repository allows performing queries, filtering particular data sets of interests, and, of course, storing new results produced during the abstraction. Single analysis techniques are applied (e.g., see the overviews in [Koschke 2005], [Knodel 2006b], [Pollet 2007]), but more often than not, an arbitrary combination of analysis techniques is applied (e.g., [Waters

1999], [Pinzger 2003], [Knodel 2005a], [Sartipi 2006]). Abstraction identifies the facts relevant to a given request and then abstracts these facts to the level of abstraction appropriate for their intended users, which can then be presented to the users. For each abstraction, concrete goals have to be defined in order to perform the respective reverse engineering analysis in an efficient way. In abstraction, we explicitly distinguish between basic and detailed analysis [Knodel 2004]. On the on hand, basic analyses are predefined, parameterized, repeatable analyses that can be applied with reasonable usage of available resources. These analyses can be regarded as a standard catalog that can be executed directly with only slight adaptation necessary. In our opinion, it pays off to have predefined, reusable basic analyses collected in a catalog to be executed by the reverse engineer on demand. Examples of such basic analyses are context analysis of code entities, reconstruction of class and inheritance hierarchies, call graphs, standard data flow analysis, design pattern recognition, and architecture compliance checking. On the other hand, detailed analyses, in contrast to basic analyses, may require significant additional effort for their application, for extending the analysis infrastructure, for fact extraction, and often require several iterations and calibrations until they produce adequate results. They aim at getting a deeper understanding of specific system aspects, and therefore, more effort is required. Also, the involvement of human experts is increased.

- **Presentation:** The presentation shows the information produced in abstraction to the stakeholders. The result reports may consist of text, tables, or visualized information. Visualization is a sound means to facilitate the understanding of complex correlations and offers a broad variety of concepts. The visualization of data offers human beings the potential to easily see complex correlations, which are not obvious by just looking at the pure data in a textual or tabular form [Knodel 2006d]. These abstractions are packaged into views on the system that the intended stakeholders interpret based on their knowledge of the system.
- **Interpretation:** Eventually, stakeholders interpret the information produced by reverse engineering. Interpretation follows one of three interaction cases – either refinement of reverse engineering analyses, derivation of action items, or keeping the status quo:
    - **Refinement of reverse engineering analyses:** The results produced by reverse engineering were either not sufficient, raised further questions about the system under investigation, or did not address the stakeholders' requests appropriately. The reverse engineering analyses have to be refined or different analysis techniques have to be selected.
    - **Derivation of action items:** Based on the reverse engineering results, stakeholders perceive potential root causes of problems, determine explicit threats, or have

confidence in the probabilities of certain risks. Hence, the stakeholders trigger counteractive measures that eventually lead to changes in the system artifacts. Once conducted, the same analyses can be repeated to evaluate if the changes had the envisioned impact on the system artifacts.

- **Keeping the status quo:** The reverse engineering results indicated the wanted or envisioned state and require no further action, but stakeholders' confidence in keeping the status quo has been confirmed.

3.2 Applying the Reverse Engineering Archetype

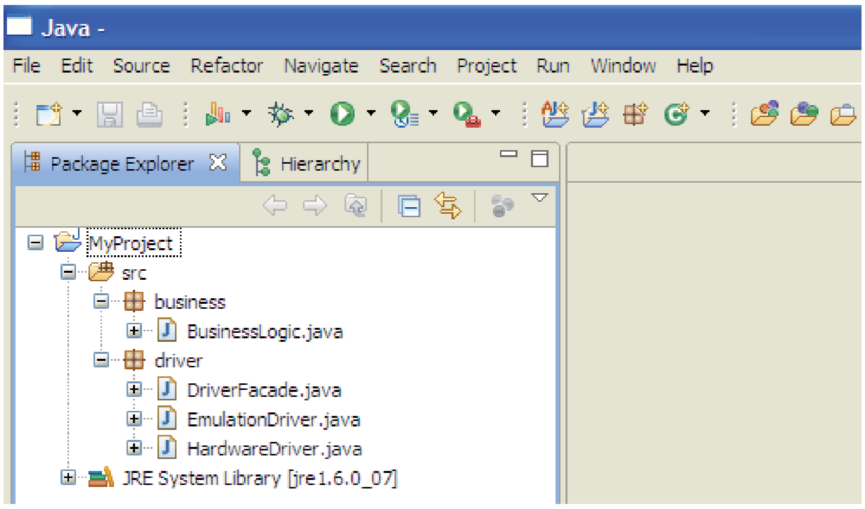
The discipline of reverse engineering, which processes existing artifacts and mines them for relevant information, is called fact extraction. Compliance checking uses fact extraction to distill the source code model from the implementation of the software system. The source code model is generic and abstracts from concrete programming languages. Because manually populating the source code model is not possible due to the size of modern software systems, fact extraction applies source code parsing or pattern matching to generate the facts about the system under investigation. Parsing and pattern matching are specific to one programming language but can be applied to all systems implemented in this language. They process certain language constructs and populate the fact base (or a repository) with selected, relevant pieces of data. All fact base entries are made according to a predefined format, which allows querying the fact base afterwards.

In this section, we present how fact extraction works for one toy example system called *DRVFaçade* implemented in the Java programming language. Appendix C lists the complete source code for this example. The example exemplifies the distinct steps according to the reverse engineering archetype (i.e., extraction, abstraction, presentation, interpretation) and uses the Fraunhofer SAVE tool [Knodel 2009a] for visualization purposes.

3.2.1 Context – System Artifacts

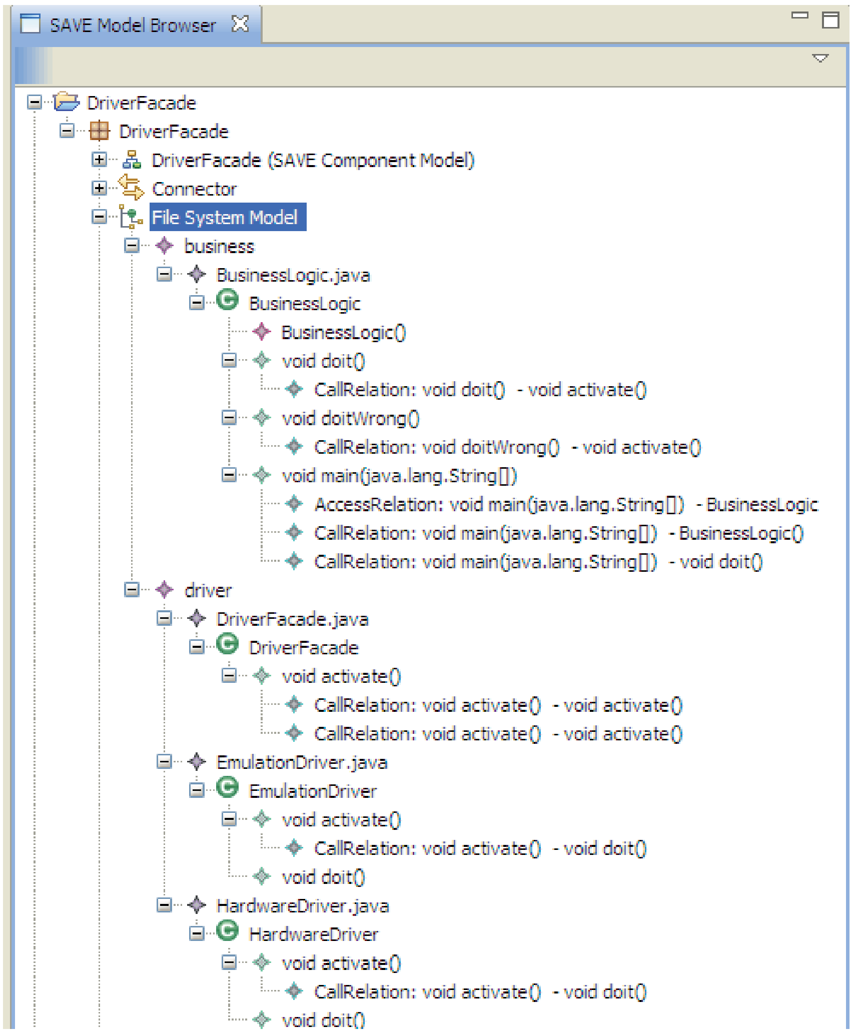
Figure 24 depicts the small toy example *DRVFaçade* in the Java package explorer of the Eclipse development environment. We can see that the Java project comprises two packages, one called *business* and the other one called *driver*. Both packages contain further Java files, which contain the source code realizing the functionality of the system.

The *DRVFaçade* system comprises two variants, one operating on hardware and the other emulating the hardware. Depending on the mode, the system class *DriverFaçade* determines which actual driver has to be executed by the class *BusinessLogic*. The package *driver* comprises the façade for encapsulating the *HardwareDriver* and the *EmulationDriver* classes.



```
MyProject
  src
    business
      BusinessLogic.java
    driver
      DriverFacade.java
      EmulationDriver.java
      HardwareDriver.java
  JRE System Library [jre1.6.0_07]
```

Figure 24 DRVFaçade: Source Code in Java Package Explorer



```
DriverFacade
  DriverFacade
    DriverFacade (SAVE Component Model)
    Connector
    File System Model
      business
        BusinessLogic.java
          BusinessLogic
            BusinessLogic()
            void doit()
              CallRelation: void doit()  - void activate()
            void doitWrong()
              CallRelation: void doitWrong()  - void activate()
            void main(java.lang.String[])
              AccessRelation: void main(java.lang.String[])  - BusinessLogic
              CallRelation: void main(java.lang.String[])  - BusinessLogic()
              CallRelation: void main(java.lang.String[])  - void doit()
      driver
        DriverFacade.java
          DriverFacade
            void activate()
              CallRelation: void activate()  - void activate()
              CallRelation: void activate()  - void activate()
        EmulationDriver.java
          EmulationDriver
            void activate()
              CallRelation: void activate()  - void doit()
            void doit()
        HardwareDriver.java
          HardwareDriver
            void activate()
              CallRelation: void activate()  - void doit()
            void doit()
```

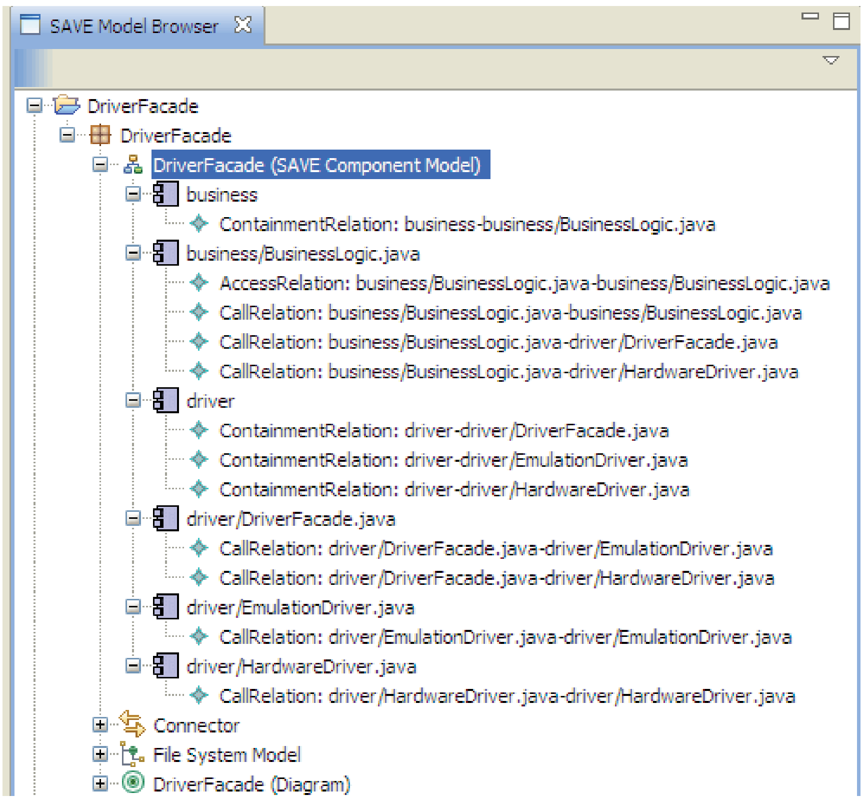
Figure 25 DRVFaçade: Source Code Model

3.2.2 Extraction

In the extraction, each source code file is processed and mined for relevant information about structure and relationships to other source code files. Figure 25 depicts the source code model of the example *DRVFaçade*. The structural elements of the Java packages, files, and classes have been extracted and are now represented as hierarchical nodes of the source code model. Further, the model comprises methods and the dependencies caused by them. Here we show the *CallRelation*, which stands for a directed method invocation from one method to another, and the *AccessRelation*, which represents the access to a class instance.

3.2.3 Abstraction

In order to visualize the example system, we created a structural model according to simple mapping instructions. Java packages and classes are mapped one-to-one onto architectural elements (i.e., packages are abstracted to subsystems, classes to components). Figure 26 depicts the structural model for this simple abstraction. It shows six architectural elements, with the elements *business* and *driver* representing the Java packages of the source code model, and the others representing the respective classes. The relationships among the source code elements have been lifted to the level of architectural elements.



```
DriverFacade
  DriverFacade
    DriverFacade (SAVE Component Model)
      business
        ContainmentRelation: business-business/BusinessLogic.java
      business/BusinessLogic.java
        AccessRelation: business/BusinessLogic.java-business/BusinessLogic.java
        CallRelation: business/BusinessLogic.java-business/BusinessLogic.java
        CallRelation: business/BusinessLogic.java-driver/DriverFacade.java
        CallRelation: business/BusinessLogic.java-driver/HardwareDriver.java
      driver
        ContainmentRelation: driver-driver/DriverFacade.java
        ContainmentRelation: driver-driver/EmulationDriver.java
        ContainmentRelation: driver-driver/HardwareDriver.java
      driver/DriverFacade.java
        CallRelation: driver/DriverFacade.java-driver/EmulationDriver.java
        CallRelation: driver/DriverFacade.java-driver/HardwareDriver.java
      driver/EmulationDriver.java
        CallRelation: driver/EmulationDriver.java-driver/EmulationDriver.java
      driver/HardwareDriver.java
        CallRelation: driver/HardwareDriver.java-driver/HardwareDriver.java
    Connector
    File System Model
    DriverFacade (Diagram)
```

Figure 26 DRVFaçade: Structural Model

3.2.4 Presentation

Figure 27 shows the structural model of the system *DRVFaçade* using the Fraunhofer SAVE tool. It displays the relationships between the subsystems and components as arrows. We can see that the *BusinessLogic* accesses the *driver* subsystem twice, once the *DriverFaçade* and the other time the *HardwareDriver*.

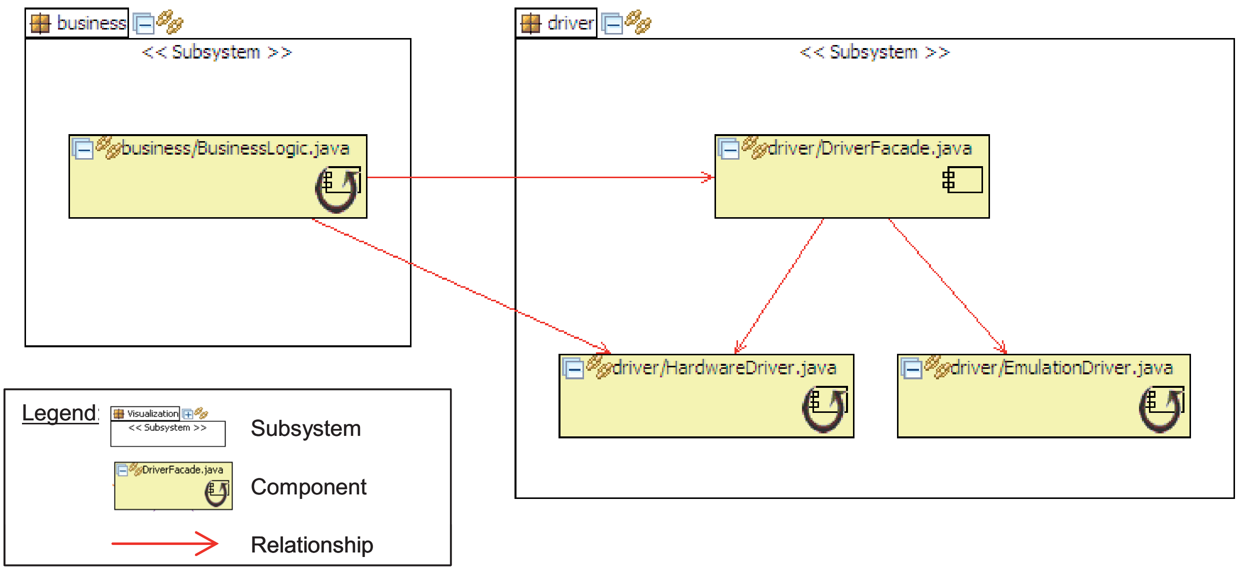


Figure 27 DRVFaçade: Visualized Structural Model in SAVE

3.2.5 Interpretation

We now use the example to exemplify how a severe maintenance problem is created by just introducing a single structural violation.

As already stated, the *BusinessLogic* is responsible for setting the mode for the driver execution – either emulation or the real hardware. The *DriverFaçade* determines which actual driver is currently set and eventually executes the right driver. The package driver comprises the *DriverFaçade* for encapsulating the *HardwareDriver* and the *EmulationDriver* classes. The architect intended this façade to be the only interface to the package driver. No other accesses are allowed.

Figure 27 shows the visualized structural model based on the implemented source code. As we can see, there are two invocations from *BusinessLogic* to the subsystem *driver*. Figure 28 lists the respective source code for two *BusinessLogic* methods – *doit()* and *doitwrong()* – implementing these dependencies.

```
public void doit(){
    DriverFacade.activate();
}
```

```
public void doitWrong(){
    HardwareDriver.activate();
    //EmulationDriver.activate();
}
```

Figure 28 Source Code of Methods *doit()* and *doitWrong()*

While *doit()* implements the access using the *DriverFaçade* as intended by the architects, *doitwrong()* directly invokes the *HardwareDriver*. This invocation is the cause of a structural violation. The developer of the method *doitWrong()* was either not aware of the façade as specified by the architect or ignored it.

Due to this structural violation, potential maintenance problems may arise:

- **The concept of separation of concerns is broken:** Replacing the *HardwareDriver* with a new implementation will cause the need to modify not only the *DriverFaçade* but the *BusinessLogic* as well.
- **The concept of localization of changes is violated:** Changes to the *DriverFaçade* (e.g., the introduction of logging or security functionality in one central place) are not reflected by the method *doitWrong()*.
- **Running the system might produce unwanted system behavior:** Using *doitwrong()* might activate the *HardwareDriver*, although the rest of the system is running in emulation mode.
- **Evolution becomes effort-intensive:** Introducing a new driver mode in the *DriverFaçade* requires not only implementing the new driver, but also reviewing and adapting all usages of the *driver* subsystem.

Compliance checking detects these structural violations, provided that the structural view has been documented. However, structural repairs by refactoring the implementation consume time and effort. The developers who are responsible might have introduced the violations months before. They might have been busy with other development tasks (or worse, they might even have left the development organization or worked in different projects). To remove the violations, the developers have to re-understand what they did and have to comprehend the architectural concept (in this case the façade) in order to implement an adequate, architecture-compliant solution.

3.3 State Of The Art in Compliance Checking

This section introduces the two main techniques for structural compliance checking – model-based and rule-based checking. Most prominent for model-based techniques are Reflexion models, while dependency rules are most typical for rule-based techniques. Both techniques share the same principles and instantiate the reverse engineering archetype. We first introduce the commonalities before delving into the specialties of each technique. Last but not least, we show the equivalence of expressiveness of these two techniques.

3.3.1 Commonalities in Structural Compliance Checking

Both techniques share common steps as depicted in Figure 29 using the [SPEM 2008] notations. Common to both techniques are the stakeholders involved: the architect and the developers as the roles providing input to compliance checking, and the reverse engineer (or quality engineer) who is responsible for conducting the compliance check. As an instantiation of the reverse engineering archetype, we can map the compliance checking activities to the fundamental reverse engineering activities.

Extraction from system artifacts encompasses the preparation of compliance checking, the inspection or parsing of the architecture documentation, and fact extraction from source code by applying parsing technology. In case of the architecture, parsers can analyze the repository or data files of the architecting tools used to manage and evolve its documentation. In case of the source code model, parsers analyze one distinct snapshot of the implementation. Hence, extraction produces the structural model (see Section 2.1) and the source code model (see Section 2.2).

Abstraction embraces two different activities – mapping and comparison. The names of the two activities already denote the key distinguishing mark of the two techniques. While Reflexion models compare two instances of models on the same level of abstraction, dependency rules resolve and evaluate textual rules defined to be valid for the architecture. The mapping is responsible for matching the specification with the implemented realization. Then the comparison does the actual verification. It verifies whether or not there are structural violations. And if so, it provides information on the specific location of each violation in the source code. Hence, abstraction processes the mapping instruction (see Section 2.3) to execute the compliance check.

Presentation continues with the visualization of the output – the resulting list of structural violations – which is then displayed for interaction with the stakeholders – the architect and the developers.

Visualization can use graphical elements for display diagrams or tabular lists or a combination of both. Configuration of the visualization (i.e., of the graphical elements it is composed of) is crucial for the perception of results [Knodel 2008c].

Interpretation then allows the stakeholders to take actions. Based on the information presented, they derive decisions on either applying counteractive measures, keeping the status quo, refining the results, or repeating the compliance check.

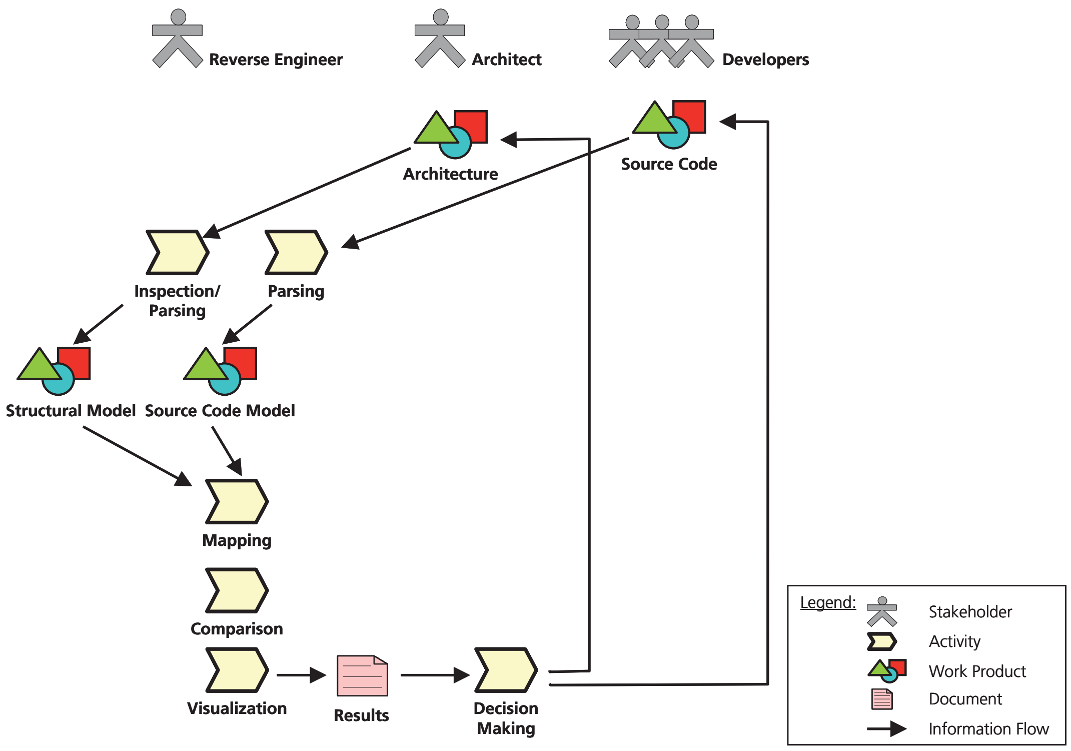


Figure 29 Principle of Structural Compliance Checking

The next sub-sections present a detailed description of each technique using the example compliance checking presented in Section 3.2. We discuss Reflexion models and dependency rules by giving an overview of the techniques, explaining how the techniques work, and giving an example application.

3.3.2 Reflexion Models

Reflexion models compare two models of a software system against each other: the structural model specified by the architect and a source code model implemented by the developers.

3.3.2.1 Overview

In [Murphy 1995], [Murphy 1997], and [Murphy 2001], Murphy et al. introduce the Reflexion models. Initially, the technique was proposed to "help an engineer use a high-level model of the structure of an existing system as a lens through which to see a model of that system's source code". It was applied in cases where no or limited information existed about the software system and its architecture. For instance, a developer of Microsoft with more than 10 years experience applied the Reflexion model technique on Microsoft Excel to reconstruct where it was necessary to identify and extract components from the source code. This developer specified and computed an initial Reflexion model (containing 15 components with 19 connections) of Excel in one day and then spent about four weeks interactively refining it [Murphy 1997]. We performed another further reconstruction-driven case study at Agilent Technologies on the firmware used to test integrated circuits – an embedded system of approx. 1.5 MLOC – [Knodel 2002]. The case study revealed the need for hierarchies within the structural models, which eventually led to an extension of the Reflexion model to overcome this shortcoming [Koschke 2003].

Knodel et al. [Knodel 2005b] and [Lindvall 2002] started to shift the application of the Reflexion model technique from reconstruction towards analytic quality assurance. We integrated the Reflexion model into architecting and first coined the term architecture compliance checking. In [Knodel 2006c], we reported on our experiences from conducting nine industrial and academic case studies. Our adaptations aim at counteracting the drift caused by structural violations – and hence, they propose using the Reflexion model technique as an analytic quality engineering technique for the verification of architectures.

Recent work by [Christl 2005] proposes a new mapping approach, which derives semi-automatic mappings based on similarity clustering of source code files and aims at reducing the expert effort in the mapping step. [Frenzel 2007] describes an extension that uses clone detection techniques and similarity metrics to transfer mappings from one variant to another within a product line context.

3.3.2.2 Mode of Operation

The Reflexion model requires a mapping between the two models to be compared, which is a human-based task. The structural model (see left side of Figure 30) specifies three architectural elements, namely components `hlm1`, `hlm2`, and `hlm3`. The arrows between the components indicate the prescribed inter-element relationships. Parsing the implementation produces the source code model (see center of Figure 30), which also comprises three elements, namely `scm1`, `scm2`, and `scm3`. The arrows in the source code model represent the source code relationships.

The results of compliance checking (see right side of Figure 30) – regardless of which concrete technique is applied – reveal that the developers have two flaws in the implementation: The inter-element relationship from hlm3 to hlm1 has not been realized (indicated by the X), and one additional dependency from `hlm2` to `hlm3` has been implemented (see exclamation mark). Only the inter-element relationship from `hlm1` to `hlm2` has been implemented as specified.

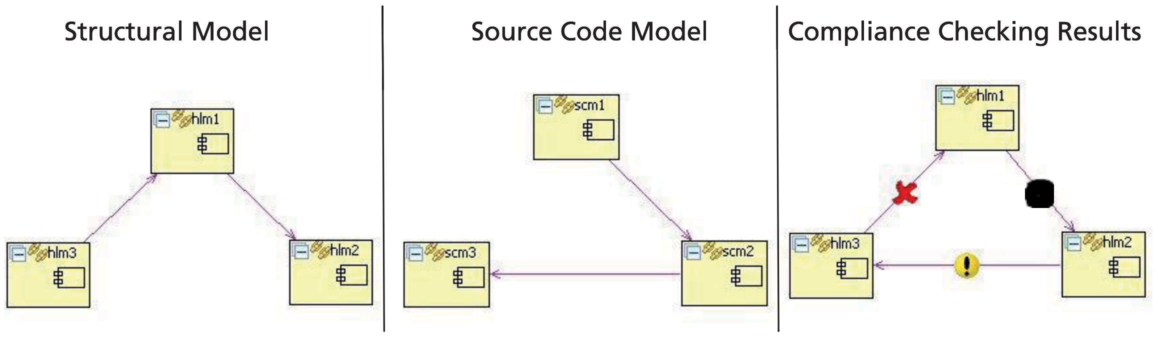


Figure 30 Reflexion Model Example: Structural Model (left), Source Code Model (middle), and Compliance Checking Results (right)

The mapping lifts the source code models to the abstraction level of the structural view. For the example given in Figure 30, the mapping defines how the architectural elements of the structural model are mapped to the elements of the source code model. For the example, this is a rather simple straightforward activity resulting in the following mapping: `hlm1` is implemented by `scm1`, `hlm2` by `scm2`, and `hlm3` by `scm3`. For both the specification and the realization, it is possible to describe the model as a set of inter-element relationships. The specification comprises the tuples A = {(`hlm3, hlm1`), (`hlm1, hlm2`)}, while the realization implements A’ = {(`hlm1, hlm2`), (`hlm2, hlm3`)}.

Having the mapping lift both models to the same level of abstraction allows the comparison, the actual computation of the results. The computation of the Reflexion model assigns one of three types to each inter-element relationship, which can be expressed as set operations: convergence (i.e., `Convergences = A ∩ A’`), divergence (i.e., `divergences = A’ \ A`), or absence (i.e., `absences = A \ A’`). Visualization dependent on the result type of the computation then shows the compliance checking results as depicted on the right side of Figure 30.

3.3.2.3 Application to DRVFaçade Example

Revisiting the example DRVFaçade as discussed above for Reflexion models requires formulating the architect's intentions as a structural model. Figure 31 depicts the layering as intended by the architect. The layer *Business Logic* uses only the layer *Abstraction Layer* to access the layer *Drivers*.

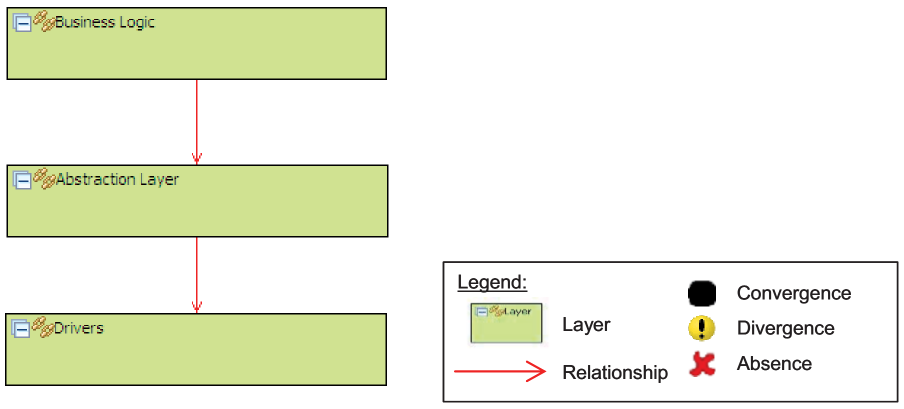


Figure 31 DRVFaçade: Specified Structural Model

We map the source code elements *BusinessLogic* to the architectural layer *Business Logic*, the element *DriverFaçade* to the layer *Abstraction Layer*, and the source code elements *HardwareDriver* and *EmulationDriver* to the layer *Drivers*. Using this mapping, we can execute the comparison and produce the compliance checking results depicted in Figure 32. The structural violation is highlighted by the exclamation mark icon. It represents the direct invocation of the method *doitwrong()* to the *HardwareDriver*.

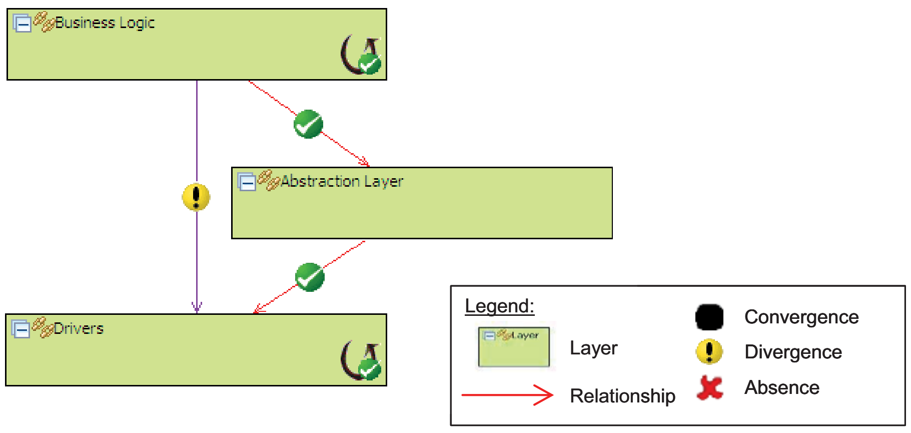


Figure 32 DRVFaçade: Compliance Checking Results with Reflexion Models

3.3.3 Dependency Rules

Dependency rules – sometimes also called relation rules – allow specifying allowed or forbidden inter-element relationships between two architectural elements.

3.3.3.1 Overview

Dependency rules specify the constraints for the interaction of architectural elements. The constraints allow, forbid, or enforce certain relationships among the architectural entities. In contrast to the Reflexion model technique, dependency rules do not require an explicit structural model as input. They rather allow textual statements on allowed and forbidden relationships among architectural elements or their counterparts in the source code. Hierarchies of architectural elements are not specified by the rules (i.e., inter-element relationships among leaf architectural elements can be checked without defining the super- architectural elements containing them) and one dependency rule can cover multiple inter-element relationships among different architectural elements.

First ideas for the verification of structural views with dependency rules have been envisioned [Harris 1995]. One of the first applications of rules for detecting structural violations is reported by [Carmichael 1995]. They manually evaluated how the components of an implemented system match the original design. To overcome the effort-intensive manual inspection, few automated approaches have been proposed, for instance [Areces 1998] using modal logic, directed colored graphs and relational algebra [Holt 1996], [Holt 1998], or relation partitioning algebra (RPA) [Feijs 1998]. Rules have been successfully applied to verify the architecture of large-scale software systems (e.g., [Bourquin 2007], [Postma 2003]).

Component rules are a special type of dependency (or relation) rules. They are defined for each single component and do not assume any knowledge about the rest of the software system. Component rules allow specifying simple ports for components that other components are allowed to call. These rules help to increase the information hiding of components on a level that might not be supported by the implementation language itself.

In Java, for example, declaring methods as public within a component is necessary for invoking such methods from other classes and packages located in the same component. All these public methods are also accessible from other components, although typically not all of them were intended to serve as an interface or a port of the component to the outside. The implementation language does not guarantee that entities outside the component boundary do not call the public methods within a component. Thus, component rules encapsulate components and allow opening only certain ports for component interaction. While dependency rules specify a relation between two components, a component rule defines which parts of it are accessible by other components.

Component rules are typically a natural language statement about the architectural elements. These statements have to be formalized into concrete rules specifying to which parts of the component access is granted and to which not. Specifying further by whom the access is granted or not turns the component rules into regular dependency rules. Typically, regular expressions are applied to match the names of component internals. Component rules are inspired by ports in architecture description languages (ADL) [Medvidovic 2000] and exported packages in OSGi [OSGi 2009]. In [Knodel 2007], we report on a first application of such component rules.

3.3.3.2 Mode of Operation

Dependency rules allow specifying allowed or forbidden relationships between two components. Such a rule is typically a natural language statement about the inter-element relationships of architectural elements. The statements can then be formalized into concrete rules, which consist of a rule type (allowed, forbidden, enforced), a source component, a target component, and a relation type (the type of inter-element relationship). For both the source and the target components of the rule, a regular expression may be defined for matching the names of components.

Dependency rules can detect similar defects as Reflexion models, but the mapping is done automatically for each conformance check by resolving the specified rules and regular expressions. In contrast to the Reflexion models, leaf-level relations of models can be checked without defining the super-components of a leaf component. One dependency rule can cover multiple relations of different components. Additionally, dependency can be used to specify allowed or forbidden connections to the context of an analyzed system, such as third-party libraries.

For the example given in Figure 30, applying the rules requires the mapping and concretization of rules. The rule types required are for allowed dependencies *may_use*, for enforced dependencies *must_use* (which, of course, implies *may_use*), for forbidden dependencies *must_not_use*, and for actually implemented dependencies `uses`. This classification is based on the approach by [Postma 2003] proposing *may_use, must_not_use, uses*. Depending on the architect's intention, the inter-element relationships in the structural model of Figure 30 can be assigned to be allowed (i.e., *may_use = {(hlm3, hlm1), (hlm1,hlm2)}*), enforced (i.e., *must_use = {(hlm3, hlm1), (hlm1,hlm2)}*), and forbidden (i.e., *must_not_use = {(hlm1, hlm2)}*). Hence, the specification is the unification of all three sets: *A = may_use ∪ must_use ∪ must_not_use*. The realization implements relationships, which are lifted by applying the same mapping again: `hlm1` is implemented by `scm1`, `hlm2` by `scm2`, and `hlm3` by `scm3`. So the lifted source code model presents the actual dependencies (i.e., *uses = {(hlm1, hlm2), (hlm2, hlm3)}*).

The computation of the rules assigns one of three types to each inter-element relationship, which can be expressed as set operations:

- *convergences = may_use ∩ uses*
- *divergence = uses \ may_use*
- *absences = must_use \ uses*

Visualization dependent on the result type of the computation again results in the compliance checking results as depicted on the right side of Figure 30.

3.3.3.3 Application to DRVFaçade Example

Revisiting the example DRVFaçade as discussed above for dependency rules requires formulating the architect’s intentions as rules. Table 9 lists the rules and their mappings to regular expressions.

Rule	Description	Regular Expression
R1	Any architectural element may use the Abstraction Layer.	*.* may_use DriverFaçade*
R2	It is forbidden for the *Business Logic* to use the layer *Drivers* directly.	*BusinessLogic must_not_use Driver*
R3	The *DriverFaçade* has to use all elements within the layer *Drivers*.	*DriverFaçade must_use .*Driver.java*

Table 9 DRVFaçade: Specified Rules

For rule R1, we map all source code elements of *BusinessLogic* as allowed origin of the rules, while the allowed target is the element *DriverFaçade*. However, this dependency is not enforced. For rule R2, we map the source code elements *HardwareDriver* and *EmulationDriver* to the layer *Drivers* and the source code elements of *BusinessLogic* to the layer *Business Logic*. Using this mapping, we can execute the comparison and produce the compliance checking results depicted in Figure 33. The results show the structural violation highlighted by the exclamation mark icon. It represents the direct invocation of the method *doitwrong()* to the class *HardwareDriver*. In the same way, we translate rule R3.

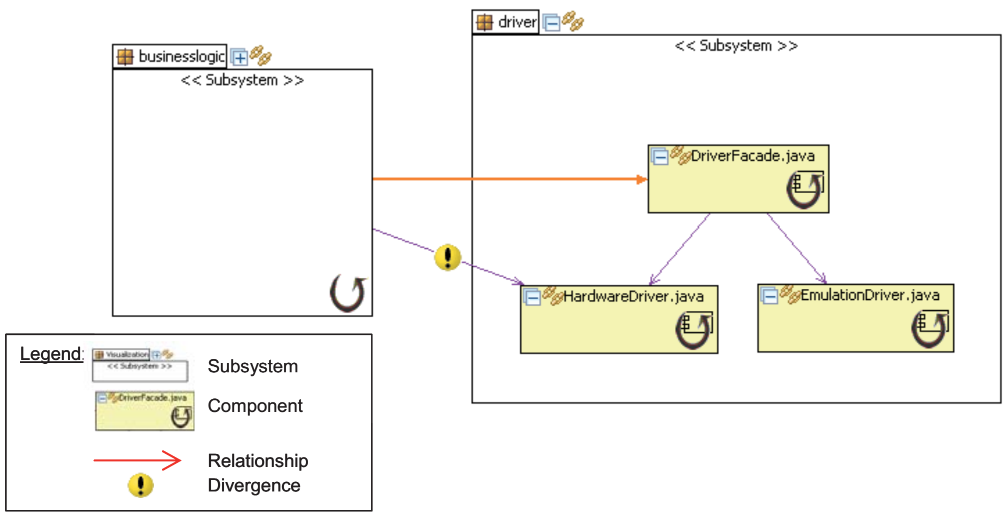


Figure 33 DRVFaçade: Compliance Checking Results with Dependency Rules

3.3.4 Equivalence of Expressiveness

The two architecture compliance checking techniques Reflexion models and dependency rules are equivalent in their expressiveness.

Definition 31 Equivalence of Sets

Let A and B be two sets: A and B are equivalent if:

$$A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$$
$$A \subseteq B \Leftrightarrow a \in A \Rightarrow a \in B(\forall a \in A)$$

Applying this definition to the sets produced by the different architecture compliance techniques shows the equality of the following sets:

- $convergences_{reflexion} = A \cap A' = may\_use \cap uses = convergences_{rules}$.
- $divergences_{reflexion} = A' \setminus A = uses \setminus may\_use = divergences_{rules}$
- $absences_{reflexion} = A \setminus A' = must\_use \setminus uses = absences_{rules}$.

By this, we can conclude that both the dependency rules and the Reflexion model produce the same results, and hence, they have the same expressiveness. The conclusion was already indicated by the application of the two techniques to the example system.

Thus, independent of the concrete technique applied, compliance checking produces the same results. However, there are certain differences in the applicability of each technique, which will be discussed in the next section, where the focus is rather on the qualitative assessment of the applicability of the technique as a basis for live compliance checking.

3.3.5 Tools for Compliance Checking

There are several architecture and source code analysis tools that allow checking compliance analytically (though they do not support live architecture compliance checking). The following list gives an overview of representative, mature, and industrial-strength tools[6]:

- **Bauhaus:** Bauhaus [Bauhaus 2008] is a reverse engineering tool suite, which supports the computation of hierarchical Reflexion models. In addition to the commercial tool, an academic sibling is a research tool for source code analysis and reverse engineering features (see [Raza 2006] and [Koschke 2008]).
- **jDepend:** The [jDepend 2008] tool analyzes the source code to measure the quality. The package dependencies (i.e., the coupling) can be used to control the structure of the software system.
- **jRMTool:** The [jRMTool 2008] is the original tool for applying Reflexion models technique [Murphy 2001]. The main drawback of this tool is the lack of support for hierarchies. The results can be visualized using Graphviz [Graphviz 2008], which is a graph drawing tool offering hierarchical layouts of trees as well as directed acyclic graphs and virtual physical layouts of undirected graphs.
- **Klocwork Insight:** [Klocwork 2008] is a reverse engineering tool, which main focus is on computing source code metrics and checking the code for security holes. Next to these main features, rules on an architectural level can be verified, too.
- **Lattix:** [Lattix 2008] is a tool that visualizes software systems in the form of a dependency matrix, which is a simple square matrix where both rows and columns denote compilation units of the system and dependencies are indicated by values in the respective cells of the matrix. Rules can be specified to define *may_use* and *must_not_use*.
- **Semmle .QL:** The source code query language proposed by [Semmle 2008] uses an SQL-like syntax to define and check architectural constraints. .QL relies on standard relational database systems to store facts about the software system under investigation.
- **SonarJ:** [SonarJ 2008] supports the analysis of Java source code. Similar to SAVE, compliance checking for dependency rules is possible.
- **Sotograph:** The Software Tomograph [Sotograph 2008] analyzes the software system and stores the information in a software repository, which allows formulating architectural rules as queries and thus checking the compliance of the system. The results can be visualized using graphs.

[6] Please note that the list of architecture compliance tools does not aim at completeness; we present only a subset of tools, which we believe to be most popular in industry.

- **Structure101:** This tool [Structure101 2008] lets the user define structural models in terms of layers and allowed relations among them, which can then be checked for compliance with the source code.
- **SAVE:** Last but not least, SAVE and its extension SAVE LiFe (see Section 5 and [Knodel 2009a]) are introduced in this thesis and are especially designed to enable live compliance checking and meet the essential requirements as stated in Table 4.

3.3.6 State Of The Art beyond Compliance Checking

Compliance is one crucial internal quality characteristic of the architecture of a software system. In addition to compliance checking, other quality engineering techniques for software architectures have emerged, namely architectural encoding in source code, scenario-based architecture evaluations, and architecture description languages.

3.3.6.1 Architectural Encodings in Source Code

A promising approach for avoiding architectural decay caused by structural violations is to directly encode architectural elements into the source code using specially defined constructs of the implementation language (e.g., [Aldrich 2002], [Lam 2003]). If programming languages provide mechanisms for denoting architectural elements like components or connectors, they can be more easily kept consistent with the code, and since the mechanisms are part of the code, changes in the code result in changed architectural elements. For example, a design may call for several components to belong to a certain layer of a layered architecture, which will be stated directly in the source code.

However, the most popular programming languages (e.g., C/C++, Java) do not support such mechanisms. So typically developers are not explicitly made aware of architectural decisions while implementing the solutions. Thus, changes or implementation decisions that affect the planned architecture might not be recorded appropriately.

3.3.6.2 Scenario-based Architecture Evaluation

Architecture evaluation aims at assessing whether or not a system to be constructed will meet its quality requirements. This kind of evaluation can be applied as soon as there is a first idea about the software architecture (either explicit or as a shared mental model). Two surveys give an overview of architecture evaluation (see [Dobrica 2002] and [Babar 2004]). The most prominent ones are scenario-based techniques. The software architecture analysis method (SAAM [Clements 2002b]) evaluates the modifiability of software architectures with respect to a set of representative change scenarios. The architecture trade-off analysis method (ATAM [Clements 2002b]) is also a scenario-based method, which extends SAAM to address other quality attributes. Its goal is to analyze whether the software architecture satisfies given quality requirements and how the satisfaction of these quality requirements trades off against each other. In [Bosch 2000], Bosch presents four architecture assessment techniques: scenario-, simulation-, mathematical model-, and experience-based assessments. These techniques also aim at the evaluating whether a system fulfils its quality requirements or not.

3.3.6.3 Architecture Description Languages

Architecture description languages (ADL) [Medvidovic 2000] provide formal notations for defining the architecture of a software system. The formal definition of the architecture enables the processing of the models specified by various tools for parsing, analysis, simulation, and code generation. Several ADLs have been proposed, for instance see [Allen 1997], [Batory 1997], [Medvidovic 1999], [van Ommering 2000] , or [Dashofy 2002]. Architecture constraint languages, a specialization of all-purpose ADLs, define formal notations for formulating constraints for the static structure of a system. Examples are the Structural Constraint Language (SCL) [Hou 2006] and LogEn [Eichberg 2008]. Due to their constructive nature, architecture description languages can typically not be adopted for existing systems.

3.4 Applicability of Compliance Checking Techniques

The main goal of architecture compliance checking is to detect spots in the source code that cause structural violations in the architecture. As mentioned above, the detection of such source code spots is independent of the selected compliance checking technique. This section compares Reflexion models and dependency rules by assessing their potential applicability as base techniques for live compliance checking. The comparison is an update of our previous work in [Knodel 2007].

3.4.1 Dimensions

Inspired by the Goal-Question-Metrics (GQM) approach [Basili 1994] (see Table 10 for the GQM goals) we derived several dimensions to determine the criteria for the comparison of the two compliance checking techniques.

GQM	Description
Object	Structural compliance checking techniques: – Reflexion models – Dependency rules
Purpose	Evaluation of their potential use as base technology for live compliance checking (i.e., as a quasi-constructive quality engineering technique)
Quality Aspect	Applicability
Viewpoint	Developer: user of the live compliance checking feedback Architect: defines, maintains, and evolves the architecture, which is input to live compliance checking
Context	Software architecture, software implementation, software evolution, quality engineering, and compliance checking

Table 10 GQM Goals

The dimensions were proposed by the author, but underwent refinements when two other members of the architecture group at Fraunhofer IESE thoroughly reviewed the dimensions. In total, we defined 16 dimensions:

- **Input:** The dimension input lists the system artifacts that are necessary in order to apply the compliance checking technique.
- **Involved Stakeholders:** The stakeholders are those persons, groups, or roles who are involved in the compliance checking activity. This dimension lists the stakeholders involved.
- **Automation:** This dimension lists the automated, tool-supported activities of the compliance checking technique.
- **Manual Activities:** The manual activities capture the interaction with the stakeholders for producing compliance checking results. The main activities must be carried out by the analyst with the involved stakeholders but they are not (semi)-automated by tools. The automated tasks, such as fact extraction or computation of the results, are ignored here, since all are automated by the SAVE tool.
- **Performance:** The verification performance captures the time required to compute the compliance checking results. For comparison purposes, a lab environment for both techniques was set up and compliance checking was executed for the same source code and the same structural view.
- **Scalability:** Scalability captures the degree to which the approach scales up for handling large-scale software systems. Scalability ranges from small via medium to large-scale systems; the rating is done based on project experiences and publications in the literature.
- **Violation Types:** The primary violation types dimension classifies the distinct groups of defects the compliance checking technique detects. In particular, we consider the following distinct groups: violating architectural elements, missing architectural elements, violating dependencies, and missing dependencies.

- **Explicitness:** Explicitness captures the degree to which the models used in compliance checking technique can be expressed without vagueness, implication, or ambiguity (i.e., the degree of leaving no question as to meaning or intent). The explicitness dimension captures the potential degrees to which developers can learn about the structural decomposition from the results provided by the compliance checking technique. Explicitness is measured on an ordinal scale with the values low (low education potential due to unclear, ambiguous results), medium, and high (high education potential due to clarity and unambiguity of the results).
- **Ease of Use:** Ease of use captures our subjective experiences regarding the intuitiveness of the compliance checking technique. We rate intuitiveness (how easily and intuitively an analyst can apply the approach) on an ordinal scale with the values low, medium, and high.
- **Ease of Learning:** Ease of learning captures our subjective experiences regarding how much training is needed for a new analyst to be able to conduct compliance checking and to produce useful results. This dimension ranges from high (few iterations and training required) to low (many iterations and a lot of training required).
- **Probability of False Positives:** The probability dimension captures the likelihood of false positives in the analysis results measured on an ordinal scale with the values low, medium, and high.
- **Maintainability:** The maintainability dimension captures the robustness of the architecture compliance checking approach with respect to code evolution. In particular, we consider the addition, modification, or removal of either architectural or source code elements.
- **Transferability:** The transferability dimension describes how the work products created in the compliance checking approach can be reused when evaluating another version of the system, a distinct variant of the system, or a different system. The first two are closely related to the system (i.e., mostly the same components, the same architecture), while the third is a completely different system with limited reuse.
- **Multiple Views:** Typically, there is not only one view of the static structure of the system; the architects often have different views, sometimes overlapping or even conflicting. This dimension captures how the approach is able to handle such multiple views and how easy it is to find out about the commonalities and variabilities in the created work products.
- **What-If Scenarios:** Restructuring scenarios captures the exploration of what-if analyses of the actual system. It aims at answering how the architecture compliance would change for different structural decompositions.

- **Validation Support:** This dimension captures how the architecture compliance checking approach supports the validation of the architecture, decision-making, trade-off analyses or scenario-specific analyses when reasoning about architectural or non-functional qualities. This dimension ranges from almost no, limited, medium, to high support.

3.4.2 Comparison

Table 11 shows the comparison results for the criteria defined above for the two major architecture compliance checking approaches – Reflexion models and dependency rules. Rows with only one entry for the two approaches indicate a commonality between them. The table assumes the general case of the application of the compliance checking technique, however, in special cases, there might be exceptions where cells of the table might be different. We derived the table based on our consolidated practical experiences gained in several industrial and academic applications.

The main **inputs** for the two approaches are obviously the source code and the structural view of the architecture, either as a model or formulated as rules. This information is usually obtained from the architecture documentation. The main **stakeholders** involved are the architect and the developers. The **automation** and the **manual activities** were already been discussed in the previous sections, where the two techniques were introduced and their mode of operation was exemplified. Except for the manual activities, all of the above-mentioned dimensions are common for both techniques.

Comparison Criterion	Reflexion Models	Dependency Rules
Input	- structural architectural view - source code	
Involved Stakeholders	- architect - developers	
Automation	- *parsing of structural view* - parsing of source code - computation of verification results - results presentation (e.g., tabular lists or visualization)	
Manual Activities	- formalization of models - decision-making	- formalization of rules - decision-making
Performance	equivalent, results produced fast	
Scalability	equivalent, applicable to large-scale software systems	
Violation Types	- violating architectural elements - missing architectural elements - violating dependencies - missing dependencies	- violating dependencies - missing dependencies

Comparison Criterion	**Reflexion Models**	**Dependency Rules**
Explicitness	high	low
Ease of Application	high intuitiveness	medium intuitiveness
Ease of Learning	high	medium
Probability of False Positives	low	high
Maintainability	- architectural elements: update structural model and mapping - source code elements: review mapping, refinements might be necessary	- architectural elements: update structural model and rules - source code elements: review rules, refinements might be necessary
Transferability	- version: no consequences for structural model and mapping - variant: adaptations to structural model and mapping - different system: no reuse possible	- version: no consequences for structural model and rules - variant: adaptations to rules - different system: no reuse possible
Multiple Views	yes	yes
What-If Scenarios	direct support with tracking	no direct support but tracking possible
Validation Support	limited	

Table 11 Compliance Technique Comparison

The **performance** in terms of computing the verification results is obviously dependent on the implementation and on the number of elements and dependencies that have to be checked. We conducted a series of evaluations with different systems (up to 500 KLoC) and varying configurations (mappings, number of rules, etc.) on a typical computer (processor 1,2 GHz, 1 GB RAM, Microsoft Windows XP Professional). The two approaches were realized in the same architecture analysis tool using the same fact extraction and visualization capabilities. The time required to compute the compliance checking results for the two approaches was less than 5 minutes for all configurations; other steps in the analysis like fact extraction and visualization of results took significantly more time. Thus, we consider the two approaches as equivalent in this dimension.

The compliance checking techniques themselves are more affected by the capabilities architecture analysis tools to parse and visualize large-scale software systems. Thus, we consider the **scalability** of the techniques as equal. The literature reports also on successful application to large industrial systems for both techniques.

The **violation types** of the two approaches have already been explained in the previous sections. Reflexion models can detect missing or violating architectural elements when performing the mapping step.

The **explicitness** of the two techniques is different. While the Reflexion models explicitly assign the source code elements to architectural elements, the resolution of the rules might create overlaps. One rule might assign a convergence to source code relationships, while they are divergent for another rule.

Ease of use is a subjective measure based on our experience. We rated the intuitiveness of Reflexion models as high since both can be applied straight forward without requiring any special training: The mapping can be done easily without any special training based on given documentation. In contrast, natural language rules can mostly be formalized but require in-depth knowledge of regular expressions. Further, the expression might become complicated depending on the elegance with which regular expressions can be formulated. For this reason we rated them as medium.

The **ease of learning** also differentiates the two techniques. Both the Reflexion models and the dependency rules support incremental refinements. They also allow trial-and-error refinements. Both can start with basic mappings or rules for parts where confidence is high. The left-out parts of the system can be expanded over time. We rated the dependency rules as medium due to the higher effort in order to apply regular expressions.

The **probability of false positives** (e.g., a computed divergence that is actually a convergence or vice versa) for Reflexion models is low, because the resolution of mappings can be reviewed and adjusted by the architect before the compliance checking is executed. It is high for the dependency rules because compliance checking is only based on regular expressions, which are only resolved at checking time.

The **maintainability** dimension addresses the ability of the compliance checking approach to evolve with the source code. We distinguish between two levels of evolution: changes resulting in addition, modification, or removal of architectural elements and changes affecting source code elements in the same way. Dependency rules are only affected by the addition of new components, because they might have a different naming convention not yet taken into account by the rules. Obviously, when adding new components, new rules are required and thus, have to be defined. When modifying components, the component rules have to be reviewed with respect to potential refinements. Changes to components imply for Reflexion models that both the structural model and the mapping have to be at least reviewed, if not updated. Changes in source code elements affected neither the Reflexion models nor the dependency rules.

The **transferability** differs between the compliance checking approaches: Reflexion models require at least rework of the mapping. Even for variants the architectural model might need a refinement, and it is not possible to transfer work products from Reflexion models when evaluating a different system. Dependency rules allow or forbid certain kinds of relationships. The rules can be applied organization-wide if naming conventions are applied consistently. However, their applicability has to be reviewed with respect to their usefulness in the context of different systems.

Reflexion models provide good support for **multiple views**. For example, consider a system with a layered architecture composed partially of reusable components distributed across the layers. The architects would be interested in the compliance of the layering on the one hand and in whether or not the reusable components have dependencies on the system-specific parts on the other hand. With Reflexion models, this is easily checked by creating two distinct structural models and mapping pairs, each capturing one of the two compliance checking scenarios. Dependency rules are able to reflect different views through different rules sets but the rule sets are all dependent on the decomposition hierarchy. Thus, views that crosscut hierarchies are difficult to define with dependency rules.

What-if scenarios for hypothesizing potential restructurings are supported by Reflexion models in two different ways: by creating artificial components as part of the architectural model and by defining a target architecture towards the implementation of the systems should be refactored to. Evaluating the implementation at constant intervals enables monitoring and tracking the progress made towards reaching the restructuring goals. Since dependency rules just operate on one model, they do not support what-if analyses. However, once the target architecture has been established, a new set of dependency rules can be created to check the compliance of the implementation regarding the target and thus, monitoring and tracking are supported.

The **validation support** offered by each technique itself is rather limited. The reasoning of the architects is dependent on their interpretation and decisions are not derived by the evaluation results only but use a significant amount of additional context information and rationales. Furthermore, there is no guidance on how to address architectural violations that have been detected.

3.4.3 Summary

The Reflexion models and the architectural rules share the same expressiveness, but the applicability of each technique has different benefits and advantages as the comparison of the two techniques has shown. Common to all approaches are criteria input, stakeholders, verification performance scalability, and validation support. We did not expect any major difference in these dimensions.

Although the two techniques are based on the same principle of compliance checking, the techniques differ in many ways. The main differences concern the dimensions maintainability, transferability, multiple view support, and what-if scenarios. Due to these differences, the architects have to decide which alternative fits better to their goals and the application context for architecture compliance checking.

Our goal – applying compliance checking as a quasi-constructive quality engineering technique – favors the high education potentials due to the explicitness of Reflexion models. Further, their ease of use and ease of learning make them an appealing choice in technology transfer projects because the overhead due to the new technology is limited. Moreover, the probability of false positives is lower than for dependency rules.

In short, goal-driven selection of the compliance checking technique is crucial. This section elaborated on our decision to choose Reflexion models as the base technology for live compliance checking.

3.5 Paradigm Shift towards Live Compliance Checking

In order for compliance checking to be applicable as a quasi-constructive compliance checking technique, a paradigm shift is required. We illustrate this paradigm shift by using the general approach for the achievement of a certain quality as proposed by Deming [Deming 1986]. This so-called Deming cycle consists of four consecutive steps: Plan, Do, Check, and Act (PDCA).

Using these four steps to describe the technique as the approach to achieving the quality compliance yields the following steps:

- **Plan:** The plan establishes the objectives that the resulting software system has to fulfill. In order to verify that the implementation is built right, the architect defines the plan – the specification of the structural decomposition. This decomposition prescribes the structure of the system in terms of architectural elements and inter-element relationships.
- **Do:** The developers do their work, which means they write source code. This step realizes the software system according to the plan.

The developers create the solution using algorithms and data structures using the basic constructs offered by the programming language. In other words, they implement the system from scratch or modify an existing system.

- **Check:** The check evaluates the results against the objectives and specifications and reports the outcome, which is exactly what the comparison does in architecture compliance checking. It compares the specification defined by the architects with the realization implemented by the developers. The compliance checking results are the outcomes.
- **Act:** In case deviations from the plan have been detected during the check step, the act step decides about counteractive measures for necessary improvement and creates a new plan to be realized by the developers.

Compliance checking and correction aim at shaping the implementation towards the plan. This means eventually iterating over all steps (Plan, Do, Check, Act) again and again until architecture compliance is finally achieved. Figure 34 depicts the Deming cycle, illustrating the consecutive execution of the steps at distinct points in time. Executing compliance checking now as a live analysis technique with direct feedback to developers requires major adaptations to these steps. In contrast, live compliance checking executes the latter three of the four steps of the Deming cycle concurrently (see Figure 35). Still, the plan step is the same; the architect is responsible for defining the decomposition that the implementation should adhere to. But the steps Check and Act are executed while the developers are executing the Do step (i.e., while they are writing the source code). The Check step is conducted continuously and constantly from day one of the implementation phase. Hence, for every single modification made by developers in the Do step, compliance checking is executed and provides immediate live feedback. The live feedback on compliance – just after the modification was made – allows prompt and direct reaction. The developers receive information on where the source code is not compliant, and can immediately remove the structural violation just introduced. Hence, this quick, high-frequency cycle of executing Do, Check, and Act concurrently for every source code modification reduces the mean time for structural repairs.

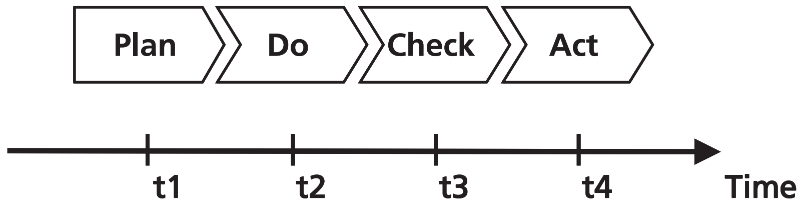


Figure 34 Deming Cycle for Analytical Quality Engineering

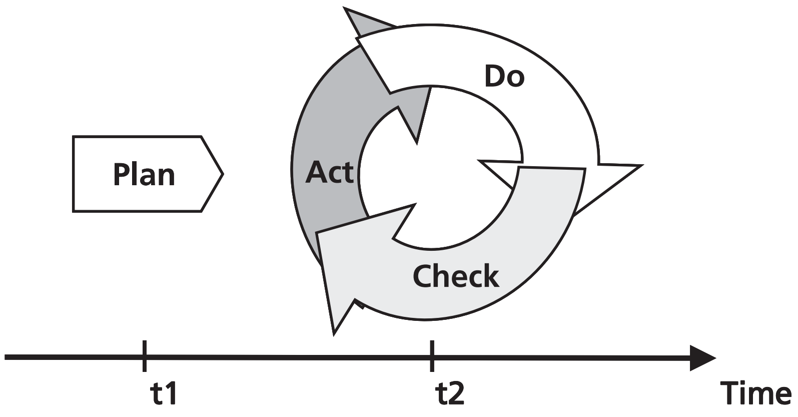


Figure 35 Deming Cycle for Quasi-Constructive Quality Engineering

Furthermore, the constantly repeated feedback presents a live education for the developers. They are trained over time and become aware of the intentions the architect had in mind with the structural decomposition. Eventually, this knowledge created by quasi-constructive compliance checking will lead to avoiding structural violations in the first place. Hence, developers will create realizations that do not require any or require significantly less refactorings due to structural violations.

To enable architecture compliance checking as a quasi-constructive quality engineering technique, the existing techniques must be modified. The adaptation towards their application in a forward engineering environment supporting distributed development teams yields the essential requirements introduced in Table 4 (see Section 1.3). Section 4 shows how these essential requirements and thus the idea of live compliance checking have been accomplished.

4 Live Compliance Checking Approach

The live compliance checking approach executes compliance checking with high frequency and delivers fast responses, which enables both a learning effect (i.e., developers are continuously trained on the architecture) and a prompt removal effect (i.e., the immediate detection of structural violations allows their immediate removal). Both effects combined cause the ultimate goal of this thesis to become a reality: the construction of architecture-compliant implementation with sustainable structures. In Section 4.2, we explain our line of argumentation: Live compliance checking acts like a just-in-time architectural compiler. Let us draw an analogy to regular compilers: Compilers process source code based on a predefined grammar and are able to detect syntax errors caused by statements of the developers that are mal-written. Similarly, the architectural compiler (as realized by live compliance checking) detects structural violations caused by source code statements mal-written the developers. In contrast to regular compilers, live compliance checking operates on a higher level of abstraction – that of the software architecture.

To institutionalize live compliance checking within a development organization, process adaptations are required. Section 4.1 introduces these adaptations, which comprise three different but interacting process parts. The overall process instantiates and is aligned to Figure 6 (see Section 1.3) – the conceptual view on live compliance checking. Each process part describes the activities from the viewpoint of the roles involved – architect, developer, and compliance checker. While the first two represent engineers of the development organization, the latter represents an automated system. Architects and developers interact with the compliance checker while doing their daily work – architecting and implementing. All three process parts are executed continuously (i.e., all the time, while development is ongoing) and constantly (i.e., every time a change is made, either to the architecture or to the source code). However, while the architect's interaction with the compliance checker is rather seldom, developers very frequently receive live feedback.

Due to the high execution frequency, developers learn about the architecture every time they cause a structural violation. We assume a learning effect over time, resulting in less violations created and a prompt removal effect for any violations actually created. Based on these assumptions we derive a simple theoretical model quantifying the hypothetical benefits of live compliance checking (see Section 4.3). Development organizations have to spend an average effort x for fixing one single structural violation, hence the effort $n*x$ is required to remove *n* structural violations. For live compliance checking, we have the conservative assumption that the number of structural violations is halved (due to the learning effect) and the effort to remove them requires only two-thirds (due to the prompt removal effect). Hence, we can conclude that the overall effort required to achieve compliance for a given implementation is *(2*x*n)/2*3 = (x*n)/3*. In other words, live compliance checking – hypothetically – reduces the overhead effort by 67%. Additionally, compliance allows reaping the fruits of the investments made into architecting.

4.1 Process Overview

The process for achieving architecture-compliant implementations of software systems is depicted in Figure 36. It comprises three process parts: architecting, coding, and compliance checking. Architecting and coding are executed by architects and developers, respectively, while compliance checking is a system that is triggered by either one of the other two process parts and then runs a sequence of fully automated activities.

The process realizing the principles of live compliance checking consists of two loosely-coupled loops. The first loop is executed by the (typically few) architects, while the second loop is executed by typically many (or several teams of) developers. The loose coupling is due to the fact that the architecture managed by architects is input for the coding carried out by the developers. Both loops are executed continuously (i.e., all the time, while development is ongoing) and constantly (i.e., every time a change is made, either to the architecture or the source code). Consequently, the process part for compliance checking is also triggered continuously and constantly.

The loop of the architect iterates over four activities:

- **A.1. architect:** Architects, by nature, of course design the architecture of the software system. They define, document, and evolve the fundamental organization of the software system. The abstractions provided by the architecture enable the efficient evolution of the software system. Once a first draft of the architecture has been completed and consolidated, the architecture is ready to be communicated. However, architecting never really stops. Due to new or changing requirements, business goals, or organizational objectives, the architecture evolves as long as the system is alive.
- **A.2. communicate Architecture:** Architects communicate the architecture to developers, either verbally or via documentation. The developers use the architecture (and additional information sources like requirements) to start the coding process.

- **A.3. publish Architecture:** Architects publish the latest state of the architecture (in particular its structural view) to the compliance checker. From that point on, compliance checking is active and can support developers with constructive, live feedback.
- **A.4. publish Compliance Status:** The compliance checker publishes the compliance status on demand. Architects can track compliance as a crucial quality and can access the overall compliance status of the system. The compliance checker allows publishing the compliance status. The overall compliance status of the system comprises the evaluation of the latest state of the complete source code. The architect may use advanced graphical visualization concepts to analyze the status from a global system perspective and navigate the compliance checking results. If necessary, the architect plans and makes refinements due to recurring compliance problems. Integrating such refinements starts another cycle of architecting, which eventually results in an update of the published architecture. Hence, the developers always execute the coding process against the latest published release of the architecture.

The loop of the developer also iterates over four activities. In contrast to the coarse-grained activities of the architect, the developers' activities are fine-grained and executed by each individual developer on his/her own.

- **D.1. code:** Developers code the source code files of the implementation. They write statements using the constructs offered by the programming language, which transforms solution ideas into algorithms and data structures. All source code combined eventually realizes the software system as specified (if everything works out fine). The input to the coding process is the architecture, which prescribes the intended structural decomposition into components and the relationships among them.
- **D.2. send Deltas:** Developers send the delta to the compliance checker. Any modification made by any developers is forwarded to the compliance checker, but instead of sending all the source code, only the locally changed delta of the developer is propagated; other unchanged source code is ignored. The sending of changed deltas is triggered and performed automatically (i.e., without direct involvement the developer) based on events within the integrated development environment.
- **D.3. send Live Feedback:** Developers receive live feedback sent by the compliance checker on the violations that are contained in the delta forwarded. If there are no violations, the developers can continue without interruption. Otherwise, the violations are highlighted smoothly in the source code editor. The feedback is tailored to the individual developers and focuses only on their local scope (i.e., it is directed at those developers who just submitted the changed deltas). The fast response time delivers the results while the

developers are editing the same source code fragments. Their minds are still in context, allowing prompt correction of the violations.

- **D.4. correct:** Developers correct the structural violations, which is equivalent to writing source code. Hence, the statements just written or modified automatically constitute the next delta to be forwarded to the compliance checker. Potential side-effects of the correction causing new structural violations are detected immediately, because the compliance checker processes the modifications instantly. In other words, the correction is equivalent to writing source, but with the purpose of achieving compliance, which thus starts another cycle of the coding process part.

The next subsections detail the process parts architecting, coding, and compliance checking. As depicted in Figure 36, the name of the message exchanged between compliance checking and architecting or coding indicates its size. While *publish* stands for long messages transferring full models, *send* represents rather short messages.

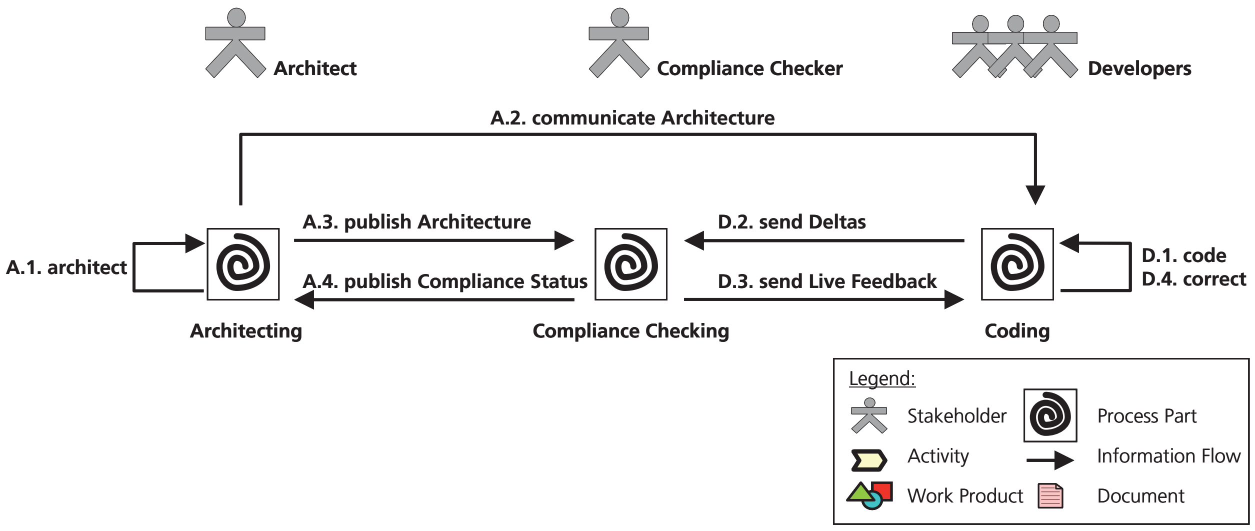


Figure 36 Live Compliance Checking: Process Overview

4.1.1 Architecting Process Part

The architecting process part comprises the cycle of activities conducted by the architect only (see Figure 37). All activities operate only on architecture-related work products and are obviously performed by the architect. For the sake of simplicity, we have ignored other lifecycle phases (e.g., requirements engineering).

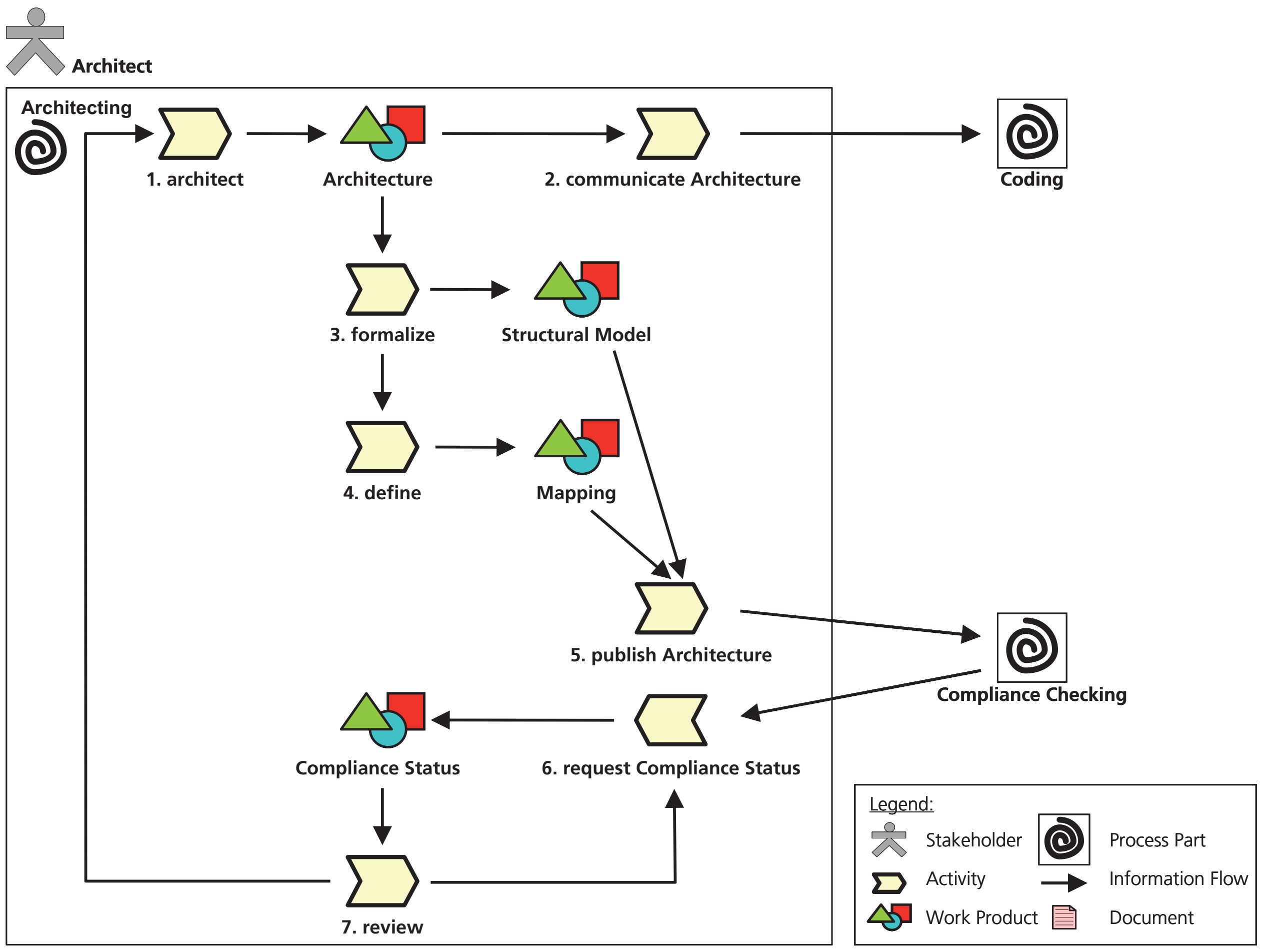


Figure 37 Live Compliance Checking: Architecting Process Part

Figure 37 list seven activities performed by the architect. While the first two activities (i.e., architect and communicate architecture) represent the regular workflow of architects, all other activities are extensions especially introduced as enablers for live compliance checking:

- **1: architect Architecture:** see description above.
- **2: communicate Architecture:** see description above.
- **3: formalize Structural Model:** To enable compliance checking, the architect formalizes the architecture, resulting in the structural model. If executed for the first time, the model is created, whereas later (optional) executions of this activity yield only minor refinements.
- **4: define Mapping:** The architect defines the mapping, which links the structural model to the source code. If executed for the first time, the architect has to define the mapping completely. Therefore, the architect requires knowledge about the source code model. Later executions of this activity are optional and typically result in refinements due to changes in the hierarchy of the source code model or new source code elements. Once the system is maturing, this activity is rather optional.

- **5: publish Architecture:** The architecture (i.e., the structural model and the mapping) are published; see above description.
- **6: request Compliance Status:** The architect requests the overall status from the compliance checker. The compliance status contains the collection of all deltas made since the start of development (i.e., the latest state of the source code). The published compliance status is made available to the architect on demand.
- **7: review Compliance Status:** The architect then reviews the compliance status. Depending on the status, the architect either continues with the next architecting cycle or requests an update of the compliance status at a later point in time.

4.1.2 Coding Process Part

The architecting process part comprises the cycle of activities conducted by the architect only (see Figure 38). All activities operate only on source code-related work products. The only exception is the architecture, which acts as the overall input to any coding activities. The coding process part is performed by any developer writing source code. For the sake of simplicity, we have ignored other lifecycle phases (e.g., testing).

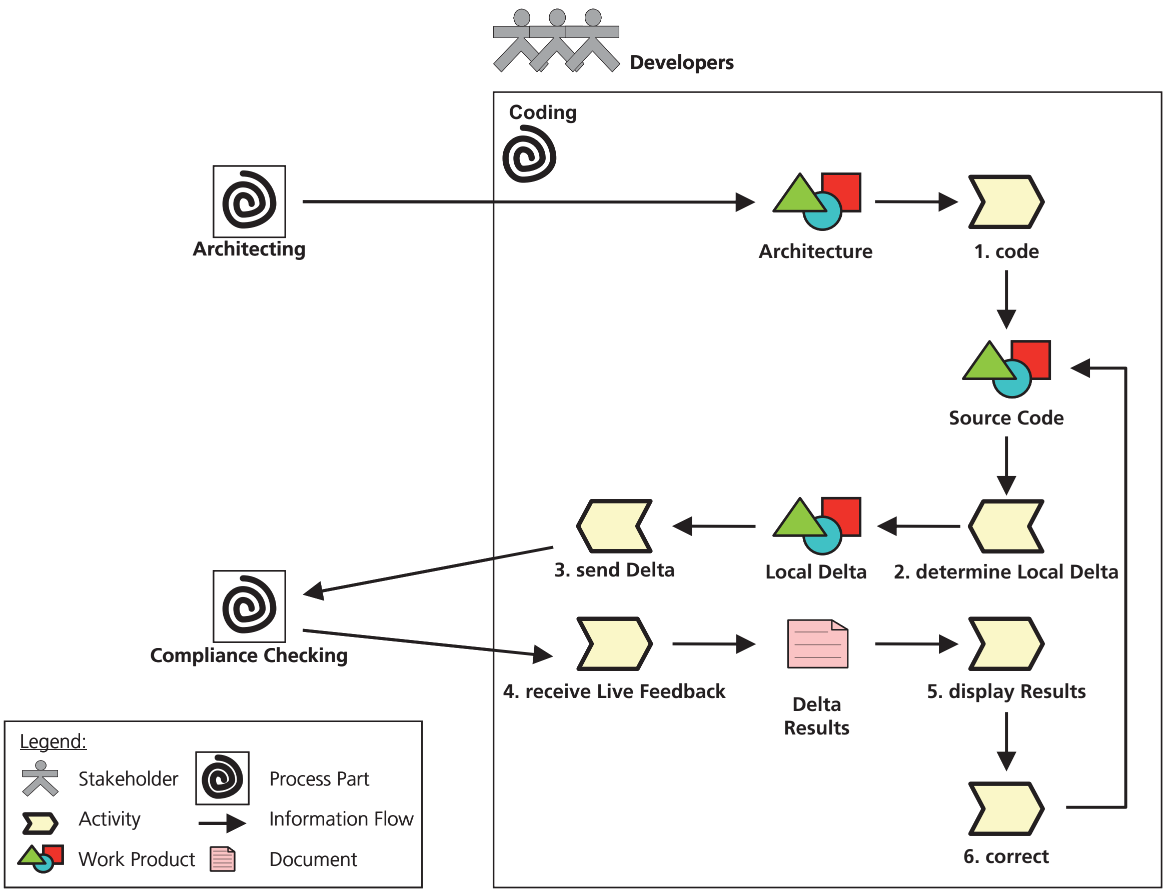


Figure 38 Live Compliance Checking: Coding Process Part

Figure 38 lists six activities performed by the developers. While the first activity (i.e., architect and communicate architecture) represents the regular workflow of architects, all other activities are extensions especially introduced as enablers for live compliance checking:

- **1: code Source Code:** see description above.
- **2: determine Local Delta:** The developer writes source code; the local modification scope is determined automatically. The modification scope comprises the locally changed compilation units – the delta or the addition in terms of source code just created by the developer. The determination uses features of the integrated development environment to monitor and track locally changed compilation units. At certain distinct events (e.g., saving a compilation unit, committing to the configuration management system) the delta determination is triggered automatically (i.e., without direct involvement the developer).
- **3: send Delta:** see description above.
- **4: receive Live Feedback:** The developers receive live compliance feedback on their individual modification by the compliance checker. The feedback is considered live because of the fast response time. In contrast to the analytical technique, live compliance checking delivers the results magnitudes faster because of limiting fact extraction, lifting, and checking to the delta only. The live feedback is received automatically by the integrated development environment and has no interaction with the developers. The results received include the set of violations relevant for the delta sent (i.e., the locally modified compilation units).
- **5: display Delta Results:** The live feedback provides the developers with the results of the compliance check. The integrated development environment displays the violations in the source code editor in a smooth way (i.e., non-intrusive, non-distracting but nevertheless appropriate). The presentation of results allows the developers to perceive the structural violations and their context (i.e., which statement causes the violations, what kind of violation it is, and what the architectural context is, such as origin and target component). Perceiving the results raises the awareness of the developers and empowers them to achieve architecture-compliant implementation by correcting the violations.
- **6: correct:** see description above.

4.1.3 Compliance Checking Process Part

Figure 39 depicts the compliance checking process part, which is fully automated. The compliance checking has two distinct entry points: one triggered by architecting and the other one triggered by coding. While the entry point used by architecting is a singleton (i.e., the architecture is managed in a central place), the entry point used by the developers can activate many compliance checking parts concurrently and execute them in parallel.

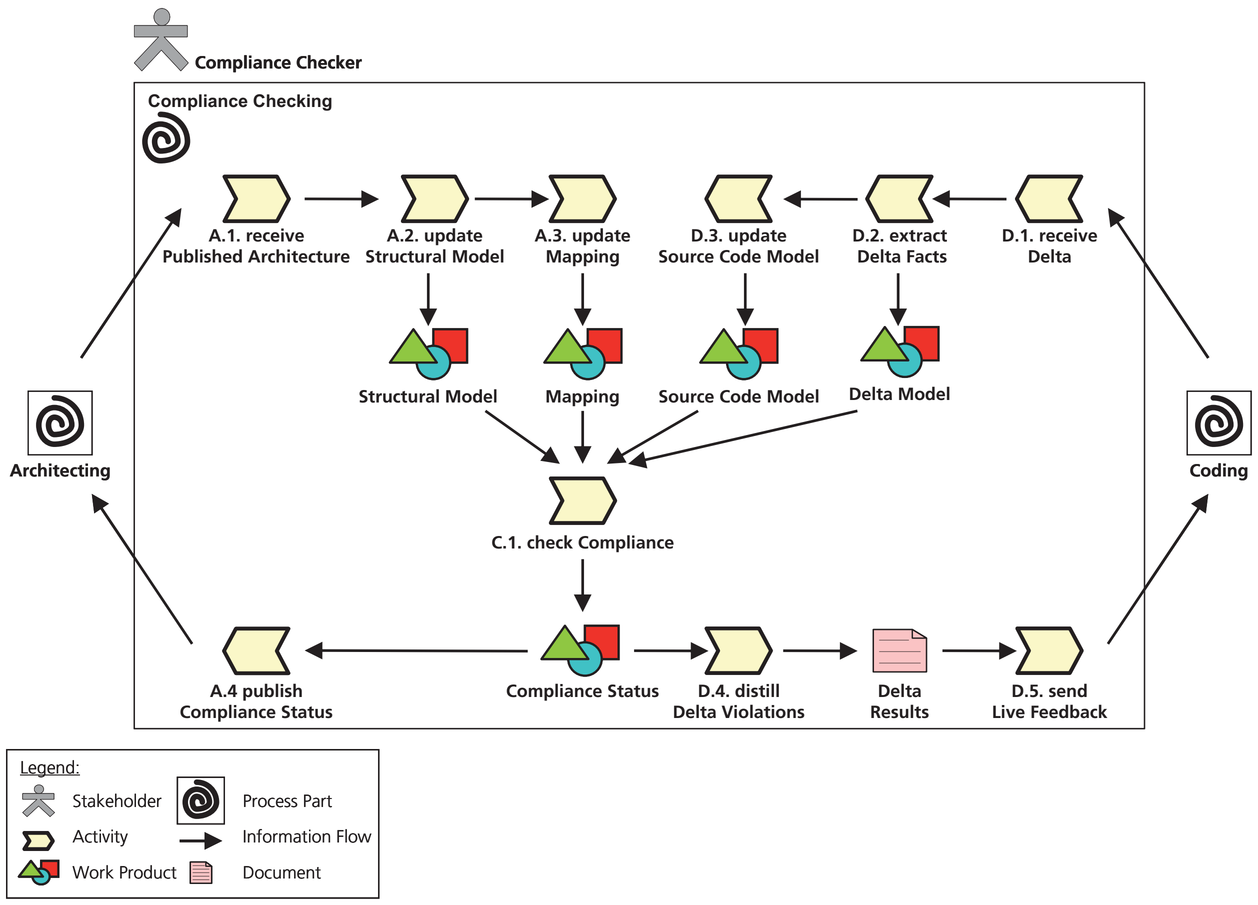


Figure 39 Live Compliance Checking: Compliance Checking Process Part

We now describe the compliance checking process part by its entry points. The entry point used by architects is the counterpart of the activity "publish Architecture" (process part "architecting", activities 5, see Section 4.1.1). Figure 39 depicts the activities on the right side:

- **A.1: receive Published Architecture:** see description above.
- **A.2: update Structural Model:** The structural model is updated based on the information published by the architect. The previous version of the model is replaced with the one published newly. At any update, the architect may have modified or refined the structural decomposition with which the developers should comply. As a side-effect, dependencies that were compliant before may change into violations when an update of the architecture is published and vice versa. This advantage propagates decisions on the structure directly to all developers concerned by the revised decision.
- **A.3: update Mapping:** In addition to the structural model, the architect may also update the mapping. Again, the previous version of the mapping is replaced with the one published newly and decisions are propagated immediately, too.

- **A.4: publish Compliance Status:** At any time, the architect can request the compliance status. This request is triggered on demand and activates the compliance checker to publish the current status, which comprises the overall result of compliance checking based on the latest state of the source code model (i.e., the aggregate of all deltas). The published compliance status is then the basis for in-depth analyses or review by the architect.

The entry point used by the developers is the counterpart of the activity "send Delta" (process part "coding", activities 3, see Section 4.1.2). Figure 39 depicts the activities on the left side:

- **D.1: receive Delta:** The compliance checker receives the delta sent by the developer. For every delta sent, a new cycle of the compliance checker handles the processing of the delta. Consequently, many deltas can be processed concurrently and compliance checking support is provided for many different developers.
- **D.2: extract Delta Facts:** The delta (i.e., the compilation unit comprising changes) undergoes fact extraction, which mines the delta for relevant information.
- **D.3: update Source Code Model:** The compliance checker updates the source code so that it comprises the latest source code state. The history of the model is kept, allowing browsing each state in a flip-book manner.
- **D.4: distill Delta Violations:** To prepare the response, the compliance distills the violations caused by the delta received by the developers. The set of delta violations is then sent back to the originator. The respective developer can now react to the live feedback and correct the source code in order to achieve architecture compliance.
- **D.5: send Live Feedback:** see description above.

The primary and most crucial activity of live compliance checking is – of course – to check the compliance. It is central to both entry points and most important for the process (see Figure 39). It computes the actual results based on the Reflexion model technique. The activity comprises six steps:

- **C.1: lift Delta:** The delta model as part of the source code model is lifted using the mapping; as a result, both models are on the same abstraction level as the structural model.
- **C.2: compare Models:** Model comparison updates the compliance status based on the structural model, the source code model, and the delta model. With the models of specified structure and the implemented system at hand, the comparison can be performed. The compliance status is updated in terms of added, modified, or deleted model elements, while unchanged elements retain their status.

- **C.3: compute Convergences:** The compliance checker computes convergences: dependencies implemented as intended by the architect.
- **C.4: compute Divergences:** The compliance checker computes divergences: unwanted dependencies comprised by the delta caused by developers.
- **C.5: compute Absences:** The compliance checker computes absences: dependencies intended but not (yet) implemented.

4.1.4 Summary

Executing all three process parts concurrently enables the overall process for live compliance checking. The process is integrated into the regular workflow of both roles, extended by new activities that are specialties of the new approach. The extensions for the architects enable them to track compliance from day one of the development, during implementation, and throughout the evolution. Modifications to the architecture are propagated to compliance checking and hence, the developers receive information on changing plans for the structure. They always implement against the latest published state of the architecture.

The extensions for developers are non-intrusive, automated, and integrated with the development environment. The source code editor displays the violations within the current modification scope (i.e., the deltas). The feedback is received live, while they are still editing the same or nearby statements. Developers “just” have to perceive the violations: Displaying them in the editor raises the developers’ awareness without distracting them from their current task. And once they are aware of the violations, the developers can remove them promptly. In other words, live compliance checking sustains the intended structure during implementation and ensures traceability between architecture and source code.

4.2 High Execution Frequency

The high execution frequency with constant live feedback turns compliance checking into a quasi-constructive quality engineering technique. It acts like a just-in-time architectural compiler for structural flaws in the implementation. The continuous application educates developers and trains them over time.

Over time, we assume that live compliance checking constitute two effects: a learning effect over time resulting in less violations created in the first place on the one hand, and a prompt removal effect for any violations actually created on the other hand. The following two sections investigate both effects – the live compliance checking characteristics.

4.2.1 Learning Effect

The learning effect (LCCC1) acknowledges the feedback on compliance. The approach of providing live feedback promises to educate and train developers about the decisions the architect has made regarding the structural decomposition of the software system. The learning curve effect [Wright 1936] states that the more times a task has been performed, the less time will be required on each subsequent iteration. Repetition of the same operation results in less time or effort expended on that operation.

Because it is likely that the developers mainly work in one part of the system (i.e., they specialize in one subsystem or component according to the principles of separation of concerns or divide and conquer), they will get experienced on the architectural constraints of those parts they are mainly working on. We believe that the same principle of this learning effect is applicable to structural violations. Each time a structural violation is detected, the developers are notified and gain more experience on the architecture. The developers receive the information about the structural decomposition already while they are in the process of writing code. They can immediately react to the feedback in order to make the solution of their current task compliant to the architecture.

Furthermore, the architect has the chance to identify recurring patterns of such violations. Analyzing these patterns enables an experience gain for both the architect and the developers. The architects can improve the architecture documentation with respect to the parts needing more explanation or improvements. Because of the experience gain, the likelihood that they will introduce the same or a similar structural violation again decreases over time.

In addition, the continuous monitoring of compliance makes it an explicit organizational goal. By giving regular feedback, the developers learn that it is important to be compliant to the architecture. This explicitness creates a peer pressure within the development organization, which further promotes the learning effect. When we imagine the live compliance checking as an architectural compiler, it is likely that the number of structural violations will be close to zero in the long run, especially when compliance has been stated as an explicit organizational goal by management.

Hence, we claim that there will be a learning effect in being architecture-compliant.

4.2.2 Prompt Removal Effect

The prompt removal effect (LCCC2) acknowledges the fact that the later in the project defects are found, the more effort is required to remove these defects. Structural violations are another type of defects that have no direct visibility to the end-user of the software system. They rather have an indirect impact on the software development process by making the architecture unreliable as a communication and steering vehicle and causing projects to not meet their goals with respect to time, effort, and quality. The removal of structural violations can cause an overhead effort when detected late in the project.

In contrast, live compliance checking detects structural defects with fast response time. This fast detection allows reducing the mean time for identification to seconds instead of several weeks or months (depending on the frequency in which analytical compliance checking is applied). According to the general law of software engineering (see [Boehm 1981], [Endres 2003]and [Pressman 2004]), the later risks are identified and solved, the higher the total effort for fixing them (it is commonly agreed that the effort increases; however, the factor by which the effort is increased differs and depends on the detection time). The same holds for structural violations: The refactoring effort for repairing the structure increases the later the violations are detected. However, if they just have been introduced, they can be removed easily and the required solution can be achieved differently. If they reside in the implementation for a long time the risk is that an integral part is being built wrong. Thus, the refactoring of this part becomes complicated and effort-intensive. When such structural violations are removed, significant effort must be spent on understanding the source code causing the violations.

An adequate solution removes the violations without creating new ones. Because of this, structural repairs are a non-trivial task. The developers have to understand the source code, the context, and the architecture decomposition. A study by [Fjelstad 1983] revealed that up to 50% of the effort for maintenance tasks is required just to re-understand the software to be changed. The live feedback providing the compliance checking results instantly removes the additional effort for re-understanding the context. The minds of the developer are focused on the problems because they are already working on solving them. The quick identification of violations reduces the overall time for structural repairs and the assumption that prompt removal requires less effort has been confirmed by various researchers in the literature.

Because this is acting like an architectural compiler, it is likely that the effort for repairing the structure virtually disappears because it is part of the task to be carried out anyway, whereby making the solution compliant is only a minor factor in creating the solution.

Hence, we claim the effort reduction for structural repairs because live compliance checking integrates these directly into the daily work and problem-solving activities (i.e., writing source code) of the developers.

4.2.3 Summary

Combining the above lines of argument indicate that we can accept that live compliance checking achieves the envisioned learning effect and the prompt removal effect.

We claim live compliance checking leads to fewer instances of structural violations, and at the same, that the effort required for their removal is reduced. The live compliance checking characteristics form the foundations for the theoretical model on effort savings.

4.3 Theoretical Model on Effort Savings

In order to be useful but simple, the theoretical model for achieving architecture compliance has to make certain assumptions. Section 4.3.1 presents these assumptions and explains in detail why they are made and how they might affect the theoretical model. Then a theoretical model is introduced, which captures the overhead effort caused by the removal structural violations. Finally, Section 4.3.3 discusses the overall impact of an implementation that is compliant to the architecture specified on the subsequent development and evolution. The validation in Chapter 6 analyzes the validity of the theoretical model by conducting a case study and experiments. The aim of the validation is to give evidence on the validity of the model and to confirm the assumptions made.

4.3.1 Assumptions

In order to be applicable, the theoretical model has to be simple and easy. Therefore, we make several assumptions that have to be considered when applying the model:

- The architecture is well-defined and correct. It overrides the implementation. If a structural violation is detected, we assume that the structural violation is caused by the source code (and it is not an architectural flaw).
- The architecture does not change. While developers are in the process of repairing, the architecture remains untouched, and no modifications are caused due to changes in the architecture.
- All structural violations are to be repaired in the implementations. In practice, there might be a few exceptions due to technical constraints

or a remaining set of violations may be ignored due to time and effort restrictions.

- All structural violations are distinct instances. There is no recurring pattern. It is not possible to implement one complex correction fixing many violations at the same time.
- Structural repairs do not re-introduce other violations as a side-effect. Although this might happen in practice (i.e., removing one violation but creating another one as a result of the correction), we assume that each developer will find an appropriate solution for structural repairs.
- Structural repairs are an atomic task. We assume that the atomic task of removing one violation consumes, on average, the same amount of effort independent of the developer carrying out the task.
- Structural repairs are conducted sequentially. One violation after the other is removed without interference or distraction by other tasks.
- Structural repairs do not require any communication between architect and developer. The tasks for removing the violations are clear and unambiguous.

4.3.2 Effort Savings

Compliance promises that the architecture is a reliable instrument for decision-making and communication. Achieving architecture compliance is an investment into the future. The effort for achieving compliance is the effort for structural repair (i.e., the effort spent to correct all violations):

Definition 32 Compliance Achievement Effort

The compliance achievement effort is the total investment to repair all structural violations in the implementation of a software system (i.e., to remove them from the source code).

$$E_{ComplianceAchievement} = n * x$$

where $E_{Compliance\_Achievement}$ denotes the compliance achievement effort, n stands for the number of violations, and x for the effort to repair one structural violation.

Based on the lines of argumentation on the consequences of high execution frequency (see Section 4.2), we can assume the impacts listed in Table 12 on the total number of violations *n* and the time *x* required to repair one single violation. Table 12 summarizes the effects in the description and lists the impacts for weak, medium, and strong impact.

We derive three scenarios for the learning impact. Compared to regular development without any compliance checking support at all, a weak learning effect halves the number of violations, with a strong learning effect resulting in the square root of the total number of violations. For the prompt removal effect, we base the weak effect on the data by [Fjelstad 1983], which quantifies the effort for re-understanding at 50% (i.e., not required due to live feedback). The zero value for the strong prompt removal effect assumes that removing violations is a (close to) zero factor in creating the solution at all.

Effect	Description	Impact
Learning Effect	Due to live feedback received constantly and continuously developers are educated and trained. Developers learn about the intended architecture over time. The impact of the learning effect is a decrease in the total number of violations. We claim on the learning effect according to our discussion in Section 4.2.1.	weak effect: $n/2$
		medium effect: $n/3$
		strong effect: $\sqrt{n}$
Prompt Removal Effect	Due to the live feedback received developers become immediately aware of violations.. The developers can repair the structure promptly as part of their daily work. We claim on the prompt removal effect according to our discussion in Section 4.2.2	weak effect: $x/2$
		medium effect: $x/4$
		strong effect: $x*0$

Table 12 Impact Factors on Compliance Achievement Effort

Figure 40 depicts the impact for both effects graphically, while Figure 41 shows the overall graphs for a combination of the two effects, both times assuming weak effects.

The theoretical model shows in Figure 41 the substantial effort savings, which – hypothetically – can be achieved due to live compliance checking. Even if we only assume weak effects for learning and prompt removal, we can see that the compliance achievement effort is reduced to *(n*x)/4* compared to analytical compliance checking, which does not even take into account the overhead effort for workshops, meetings, and communication required to propagate the compliance checking results in the analytical application case. Assuming the effects to be medium active, we would have a compliance achievement effort of *(n*x)/12*, which reduces the effect even more. And if developers automatically react to feedback on violations and repair them as part of their daily work –with virtually no overhead – we can assume strong learning and prompt removal effects. The compliance achievement effort in this case would be *(√n*x*0)=0*, hence the overall effort for making implementations compliant is close to or even equal to zero. This is just a barely discernible factor of the effort spent for regular development.

Of course, we speculated as to the quantification of the learning effect and the prompt removal effect. However, as we have discussed above, there are reasonable arguments for these numbers. In addition to the concrete savings in terms of compliance achievement effort, we expect side effects on the evolution due to increased compliance.

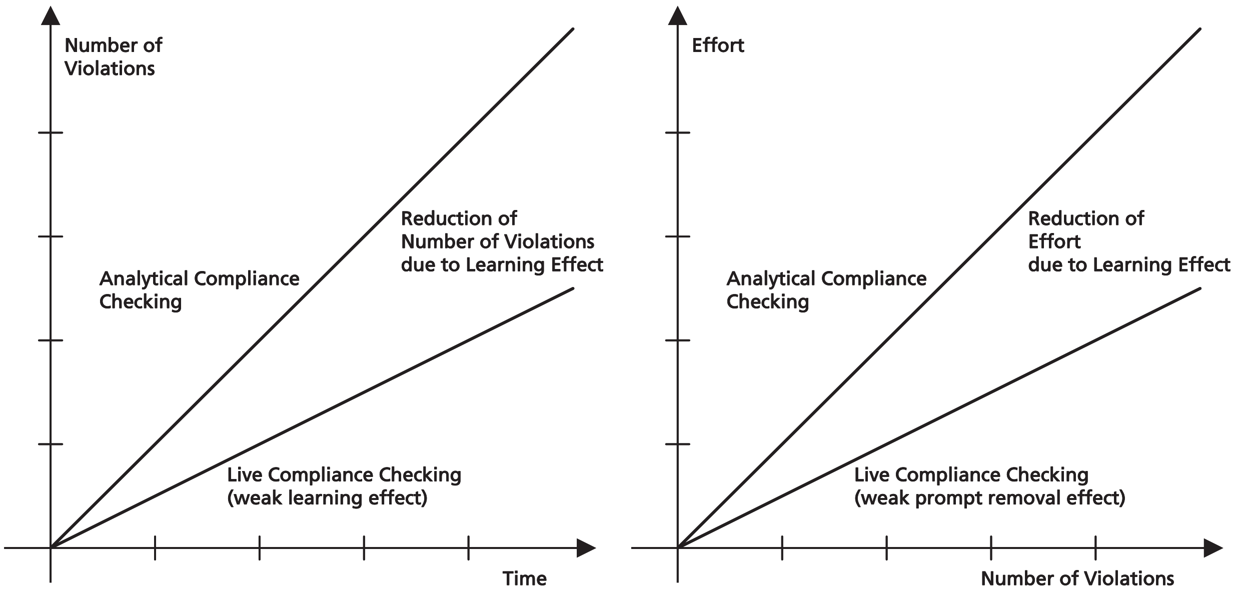


Figure 40 Effort Saving Learning Effect (left) and Prompt Removal Effect (right)

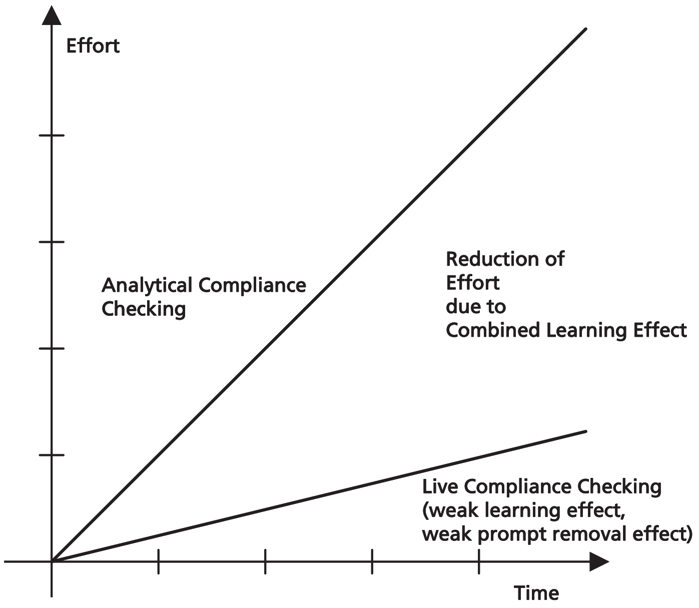


Figure 41 Effort Saving Combined Effects

4.3.3 Compliance Impact on Evolution

Live compliance checking reduces the overhead effort as described. At the same time, of course, it leads to higher architecture compliance of the software implementation, which sustains the investments made for architecting. Table 13 characterizes these side-effects and their impact on the architecture-centric evolution. The architecture is the conceptual instrument for dealing with the inherent complexity that software systems have. By providing critical abstractions, the architecture makes it possible to manage the development activities. If implementation is compliant to the architecture, the envisioned benefits of well-defined architectures can be achieved (although there is no guarantee). In contrast, the lack of compliance almost guarantees that the benefits promised by high-quality architectures will not be obtained.

Table 13 discusses the side-effects of compliance on the key responsibilities of software architectures. These responsibilities have been compiled from key publications in the field of software architecture (e.g., see [Bosch 2000], [Clements 2003], [Hofmeister 2000], [Jazayeri 2000], [Perry 1992], [Rozanski 2005], [Shaw 1996], or [Tyree 2005]).

Architecture Responsibility	Side-effects of Compliance on Evolutions
The architecture serves as a mediator for stakeholder communication.	Architecting enables stakeholders to reason about the software system. However, this reasoning becomes (partially) void when the implementation lacks compliance. It is unclear whether or not the abstraction imposed by the architecture is still valid, which jeopardizes the overall success of the development project.
The architecture serves as vehicle for efficient project planning, management, and controlling.	The decomposition provided by the architecture allows defining, handling, distributing, and progress tracking of work assignment for developers. Lacking compliance, the task assignments are made in vain. Developers write source code in a disorganized manner, which potentially worsens the lack of compliance. Furthermore, controlling progress becomes difficult.
The architecture serves as vehicle for the efficient maintenance of software systems.	The [IEEE-Std-610.12 1990] defines maintenance as the process of modifying a software system after delivery to correct faults, improve its attributes, or to adapt it to a changed environment. To perform maintenance tasks efficiently (and typically these tasks face tight time pressure), the architecture is required as a map of the system. To locate the points of modification, to learn about risks and understand potential side-effects, and to plan the change, the architecture is crucial. When lacking compliance, maintenance becomes risky and consumes more effort than planned to first reconstruct the map (i.e., the architecture).
The architecture serves as an instrument for the successful and controlled evolution of software systems.	Software systems continuously change to correspond to the frequent and increasing requests for new features, new functionality, or customer-specific customizations. The architecture is the means to plan for future modifications, prepare envisioned extensions, and define placeholders on how to integrate new components or variants. The consequence of implementations lacking compliance is that the plans are not reliable. The architecture no longer serves as a means to cope with the inherent complexity of the system.

Table 13 Compliance Side-Effect on Architecture-Centric Evolution

To conclude, the architecture is responsible for achieving the overall success in software development. Consequently, the success of architecting stands or falls with the compliance of the resulting implementation.

5 Software Architecture Visualization and Evaluation with Live Feedback

To be applicable, to justify the name "live", and to enable direct and fast feedback to developers, compliance checking needs to be tool-supported to automate the necessary activities. The tool realizes the requirements stated for live compliance checking and supports the process introduced in the previous section.

The tool is called SAVE LiFe – Software Architecture Visualization and Evaluation with Life Feedback (see Section 5.1). It consists of three parts – the fat client for architects, the central server, and the thin client for developers. This deployment into client-server-client reflects the roles defined in the process for live compliance checking (see Section 4) and is based on the meta-models defined for compliance (see Section 2). The key functionality of the tool is, of course, the compliance check based on Reflexion models (see Section 3).

Underlying the logic functionality of checking compliance is a basic communication platform that enables logical, bi-directional channels for data exchange between the server and the clients. On the one hand the communication platform establishes one channel for transferring models back and forth, which allows publishing and receiving large amounts of data. We call this channel model exchanger. On the other hand, a second channel is used for sending and receiving small data packages. This channel is called delta exchanger. While the first channel consumes network bandwidth and considerable time for the data transfer, the latter enables fast response and live feedback. Correspondingly, the architects publish the architecture and receive the compliance status using the large channel, while developers send the deltas and receive live feedback on compliance using the fast communication channel.

The communication platform is extensible towards different kinds of analysis, where live compliance checking is the first instance that uses its features. We discuss in Section 5.3 how SAVE LiFe addresses the essential requirements on live compliance checking listed in Table 4 (see Section 1.3). In addition, we envision extending the platform towards other reverse engineering techniques and their quasi-constructive application [Knodel 2008a], which would enrich the analysis capabilities of SAVE LiFe.

Technically, all three building blocks of SAVE LiFe (see Section 5.2) are implemented in Java and built on top of the Eclipse platform. Eclipse is an open-source platform that provides an Integrated Development Environment (IDE) for developers. The platform is generic but highly extensible, which is exactly what SAVE LiFe does: it specializes the Eclipse platform for its own purpose – live compliance checking with clients and server constituting three different, individually deployable specializations.

SAVE LiFe extends its ancestor SAVE – the snapshot analysis tool – by the live feedback communication platform, the client-server architecture, the delta analysis, and the ability to execute analytical analysis techniques in a quasi-constructive manner. SAVE is a snapshot analysis tool for analyzing and optimizing the architecture of implemented software systems. It extracts information from system artifacts, performs an arbitrary kind of computation, visualizes the results, or generates system artifacts. SAVE is a joint development of Fraunhofer IESE (Institute for Experimental Software Engineering IESE in Kaiserslautern, Germany) and the Fraunhofer Center Maryland (Center for Experimental Software Engineering in College Park, Maryland, USA). The work on SAVE as an architecture analysis tool – with live compliance checking as the driving vision in mind – started in 2004 [Miodonski 2004]. The initial idea of live compliance checking evolved towards SAVE LiFe – a scalable and mature research prototype providing a live analysis platform on a central server supporting developers and architect as clients.

5.1 Solution Overview

SAVE LiFe consists of three distinct, logical building blocks, which are depicted in Figure 42. The architecture manager is responsible for realizing the process part as defined for the architect (see Section 4.1.1), and the development monitor for the coding process part (see Section 4.1.2). Both communicate with the compliance checker (see Section 4.1.3), which realizes the remaining process part. Accordingly, the roles are represented by the actors architect and developer.

The next subsections introduce the conceptual view and its instantiations for the different building blocks. We then continue with a discussion of the distributed communication platform and finally show the development environment integration of SAVE LiFe.

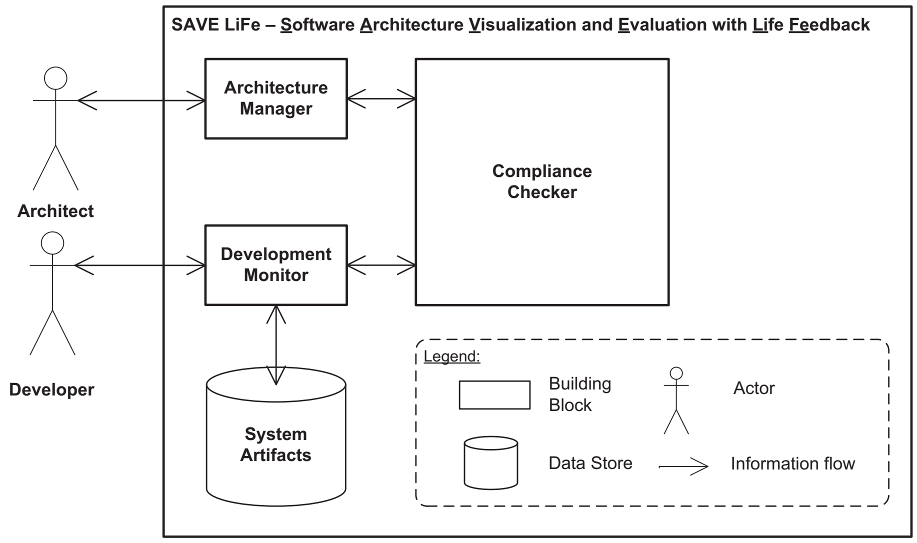


Figure 42 SAVE LiFe: Conceptual Building Blocks

5.1.1 Conceptual view

The conceptual view is the most abstract architectural view used for capturing the application domain by mapping the functionality of the system to conceptual components and showing data stores, external interfaces, and hardware devices. It also depicts the relationships among the conceptual elements.

Figure 43 depicts the conceptual view on the SAVE product line architecture, which is instantiated (partially or completely) for every SAVE analyzer, while Table 14 and Table 15 describe the conceptual components and data stores.

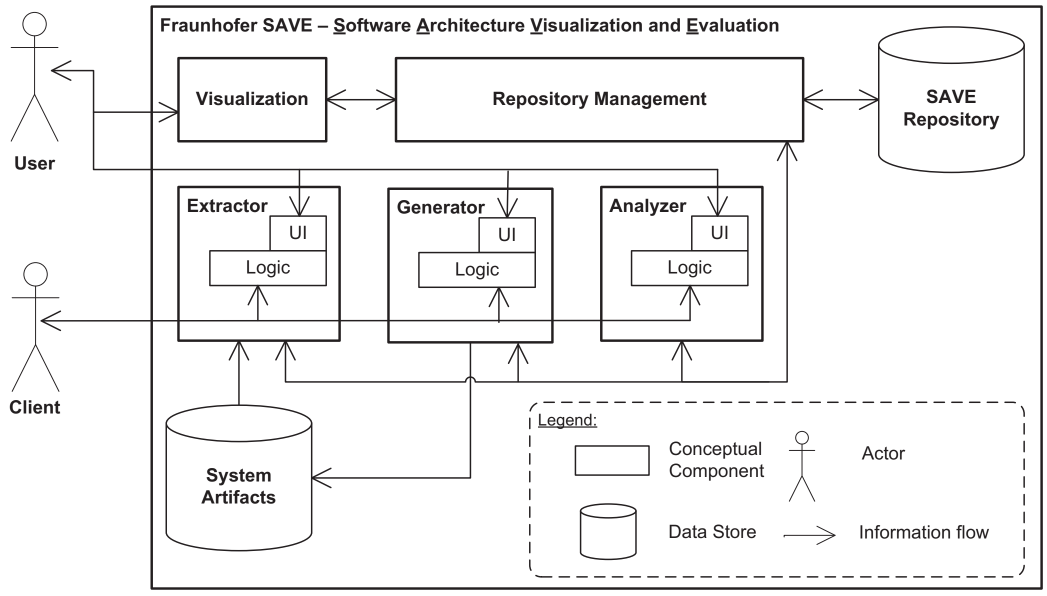


Figure 43 SAVE LiFe and SAVE: Conceptual View

The conceptual view separates the abstract, logical concepts realized by SAVE LiFe. In fact, this conceptual view is also shared by SAVE, the ancestor for analyzing system snapshots.

The actor user interacts with SAVE LiFe using the visualization or the user interfaces of conceptual components. Clients (remotely or locally) directly access the logic of the conceptual components. The conceptual components are independent from each other. This characteristic allows having many different extractors, analyzers, or generators in parallel within one SAVE LiFe or SAVE configuration. Integral for the different interactions is the repository management, which enables coupling on the data level (i.e., one analyzer can operate on data provided by one extractor). Table 14 details the description of the conceptual components, while Table 15 explains the roles of the data stores.

Conceptual Component	**Responsibility**
Visualization	The conceptual component Visualization is responsible for the visualization of information stored in the SAVE Repository. The information visualized in models comprising entities and relations is produced by the conceptual components Extractor or Analyzer. The visualized information may be displayed in graphical, charting, tabular, or textual form. The actor User is able to interactively navigate, browse, filter, and manipulate the information presented in order to gain knowledge from the information.
Extractor	The conceptual component Extractor analyzes existing system artifacts by applying fact extraction functionality (e.g., parsing, pattern matching, filtering, or data importing) to gather information about the System Artifacts processed.
Generator	The conceptual component Generation is responsible for producing System Artifacts that are generated based on the information stored in the SAVE Repository. System Artifacts can be created newly or existing ones can be modified.
Analyzer	The conceptual component Analyzer provides analysis functionality for abstracting, aggregating, comparing, transforming, or enriching a SAVE model. All computation functionality processes information from at least one existing model in the SAVE Repository and either modifies it, or creates a new model(s), which is (are) stored in the SAVE Repository, too. The information extracted by the conceptual component Extractor is typically processed by the Analyzer to mine it for relevant and crucial information in the SAVE Repository. Analyzers are either initiated by the actor User or executed (semi-) automatically by an external Client. If necessary, the actor User or the actor Client provides input or decides about the parameters and configuration for one specific execution of an Analyzer.

Conceptual Component	Responsibility
Repository Management	The conceptual component Repository Management is responsible for creating, accessing, managing, loading and storing the SAVE Repository, the internal data model of SAVE. All accesses to a SAVE Repository have to use Repository Management; no direct access is allowed.

Table 14 SAVE LiFe: Conceptual Components

Data Stores	Responsibilities
System Artifacts	The data store System Artifacts comprises all artifacts of the system. The most commonly analyzed artifact is the source code; however Extractors are not limited to it. System Artifacts comprise data from configuration management systems (e.g., CVS, SVN), instrumented run-time traces, intermediate representations (e.g., GXL, RSF, CSV), CASE tools (e.g., Rational Modeler), defect databases (e.g., BugZilla, JIRA), third-party metrics tools (e.g., Understand, JHawk), build scripts (e.g., Makefiles, Antfiles), and other available artifacts.
SAVE Repository	The SAVE Repository is the central data store of SAVE. By its nature, it stores all data produced by one of the Extractors, or Analyzers, and is the basis for the output produced by the Generators.

Table 15 SAVE LiFe: Data Stores

5.1.2 Client-Server-Client Deployment

The three building blocks – architect manager, compliance checker as application on top of the extensible analysis and communication platform, and development monitor – are depicted in Figure 44, Figure 45, and Figure 46, respectively.

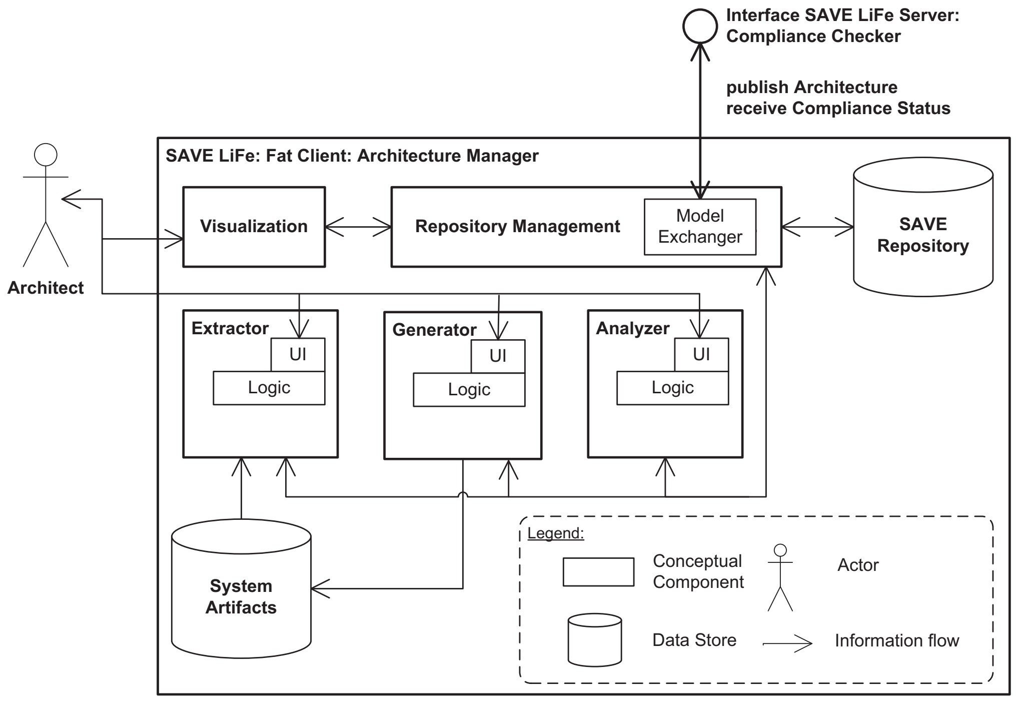


Figure 44 SAVE LiFe: Fat Client: Architecture Manager

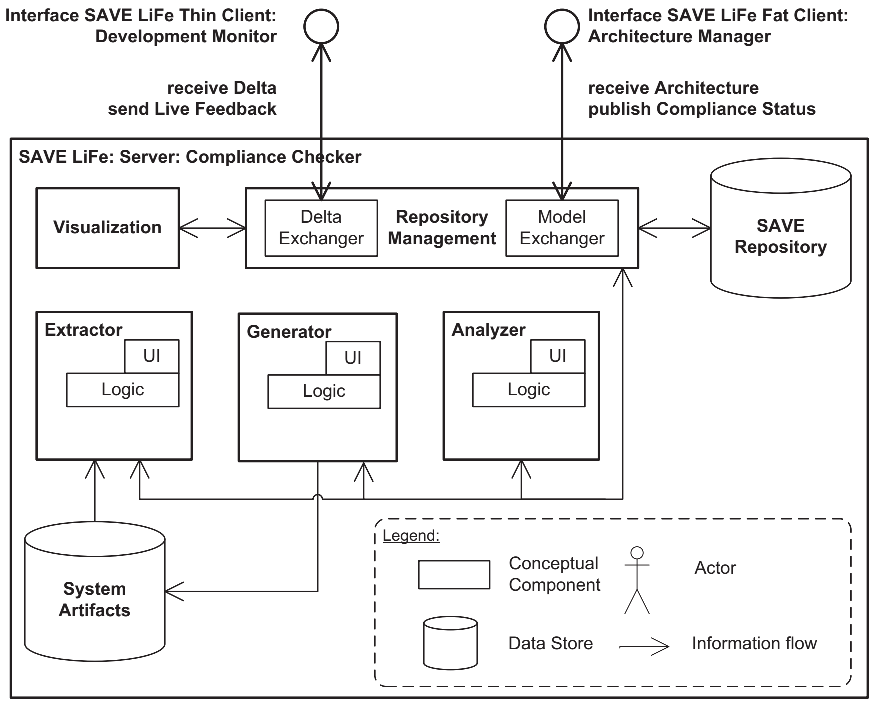


Figure 45 SAVE LiFe: Server: Compliance Checker (Extensible Analysis and Communication Platform)

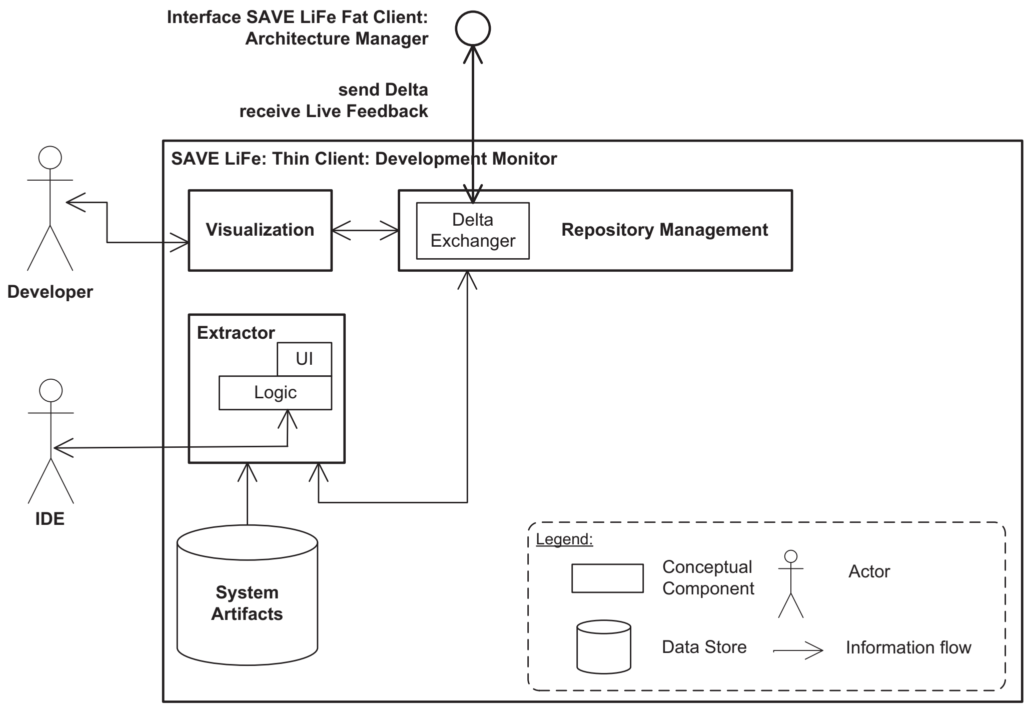


Figure 46 SAVE LiFe: Thin Client: Development monitor

By preserving the layout and positioning for the same elements throughout the figures (i.e., Figure 44 to Figure 46), we can illustrate the differences between thin client, fat client, and server.

The fat client instantiates the conceptual view completely. We highlighted the expanded repository management, which contains the model exchanger. The model exchanger is responsible for publishing the architecture and receiving the latest compliance status. The compliance status allows browsing, navigating, and delving into details. The fat client is very similar to the stand-alone version of SAVE. It allows analyzing one snapshot of the system, which, in this case, is always the latest state of the development. Using the history feature of SAVE, the architect can navigate back in time using a flip-book kind of browsing, which visualizes the changes that have occurred at distinct points in time.

The server realizes the basic communication and analysis platform of SAVE LiFe. It communicates with two clients, the architect manager and the development monitor. The server is running independent of any external actor. It is triggered by incoming data on one of the two communication channels and responds in the intended way by either publishing the compliance status or sending live feedback on compliance. All three figures show the relevant ends of the communication channels.

The thin client comprises only visualization for displaying the live feedback and the extractor for determining the local modification scope, which is then sent to the compliance checker. The development environment triggers the functionality automatically when predefined events happen (e.g., saving a compilation unit). Typically, there are many developers using many thin clients communicating with one central server.

5.1.3 Distributed Communication Platform

The mechanism for enabling communication between the server and the two clients is realized using Java remote method invocations (see [RMI 2009]). RMI as a multi-threaded technology allows writing distributed applications, which allows the server to process client requests in parallel.

RMI establishes a logical communication channel between clients and servers. The data structures distributed objects can be created and called from instances, running on different Java Virtual Machines (JVM), as if they were local. It is irrelevant whether they are located on one machine or on different machines connected via a network. The communication interfaces are defined in interfaces shared by clients and server. The methods of the server interface are implemented by classes in the server, the ones of the client by classes located in the client plug-in.

Clients can connect to the server by registering themselves to the server. The server listens to a defined Internet protocol (IP) address on a distinct port. The clients can initiate communication by connecting to the specified IP address and port. Clients identify themselves by their own IP address, a callback port, and a name representing the current name of the client.

The basic communication platform of SAVE LiFe establishes two logical, bi-directional channels for data exchange between the server and the clients. The remote data exchange allows either to transfer large amount of data on a communication channel called model exchanger or smaller bits of information using the communication channel called delta exchanger.

5.1.4 Development Environment Integration

We decided to integrate SAVE LiFe into the Eclipse IDE (integrated development environment) [Eclipse 2009]. Figure 47 sketches this integration and lists the third-party plug-ins that are reused. SAVE LiFe extends the regular Fraunhofer SAVE tool. The Eclipse Modeling Framework (see [EMF 2009]), the Graphical Editing Framework (see [GEF 2009]), and the Graphical Modeling Framework (see [GMF 2009]) are reused for data model management and visualization purposes. The

Fraunhofer PuLSE Common Architecture acts as an abstraction layer on top of these plug-ins.

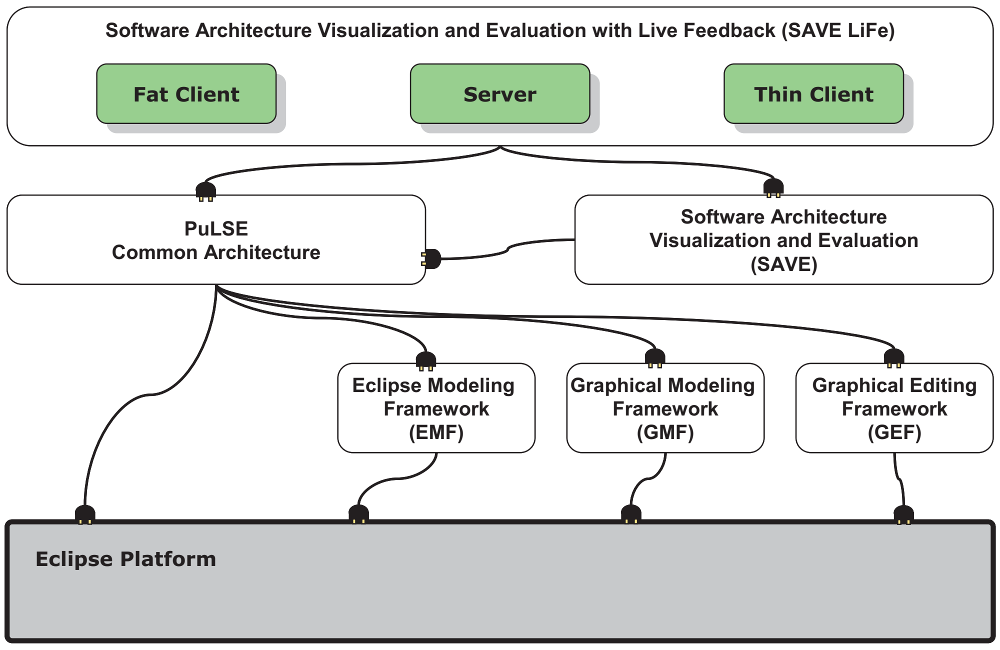

Software Architecture Visualization and Evaluation with Live Feedback (SAVE LiFe)
Fat Client
Server
Thin Client
PuLSE Common Architecture
Software Architecture Visualization and Evaluation (SAVE)
Eclipse Modeling Framework (EMF)
Graphical Modeling Framework (GMF)
Graphical Editing Framework (GEF)
Eclipse Platform

Figure 47 SAVE LiFe: Eclipse Integration and Reused Plug-ins

5.2 SAVE LiFe Building Blocks

The building blocks of SAVE LiFe are presented in this section. Features are distributed to either the server or one of the respective clients or to a combination of both. We implemented the distribution of features as presented here (however, it is possible to imagine selecting a different distribution strategy).

5.2.1 Fat Client: Architecture Manager

The architecture manager is the tool for the architect to interact with the compliance checker. It is realized as a fat client and comprises a lot of functionality.

5.2.1.1 Feature: Formalize Model and Define Mapping

The architects use the fat client of SAVE LiFe. The responsibilities of the architect are derived from the steps of the Reflexion model technique (see Section 3):

- **Structural Model Definition:** The architect specifies the structural model against which the implementations of the developers are compared. There is exactly one structural model for which the compliance checking is executed.

- **Source Code System Selection:** The architect defines the source code system to be monitored with SAVE LiFe and specifies the respective locations in the configuration management system.
- **Define Mapping Instructions:** The structural model and the source code model require mapping instructions that relate the architectural elements to the source code elements. These mapping instruction are based on the structural model and the source code model. In case there is no model yet, the mapping remains void. The mapping can be updated and changed at any time.
- **Developer Assignment:** The architect specifies the developers who work on the realization of the software system.

5.2.1.2 Feature: Publish Architecture

Figure 48 depicts the data flow in the definition of the respective models in a pipe-and-filter notation. The extraction of the structural model is done either manually or automatically, depending on the architecture documentation. The computation step prepares the mapping of the structural model to the source code model. The repository management then aggregates the models together with the user management and stores all information in the SAVE repository. Then the architect manually initiates the transfer of the configured SAVE repository from the client to the server. The transfer sends the repository data via the repository management interface as described above. This interface is bi-directional, which means the architect can get the latest version from the server at any time (or other architects can modify the SAVE repository managed by the server to their fat clients and refine the structural model, the mapping, the source code model, or the users' information).

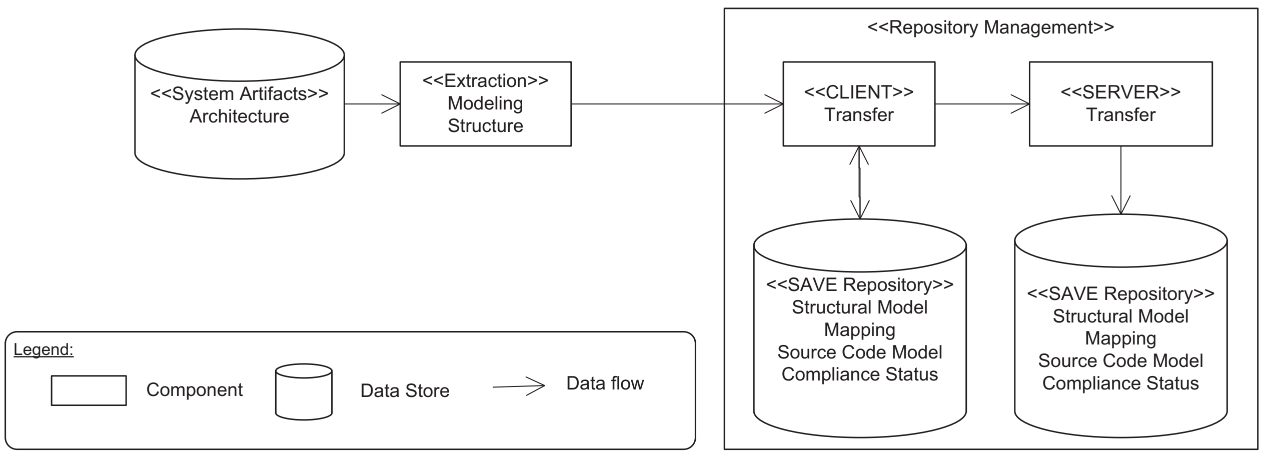

<<Repository Management>>
<<System Artifacts>> Architecture
<<Extraction>> Modeling Structure
<<CLIENT>> Transfer
<<SERVER>> Transfer
<<SAVE Repository>> Structural Model Mapping Source Code Model Compliance Status
<<SAVE Repository>> Structural Model Mapping Source Code Model Compliance Status
Legend:
Component
Data Store
Data flow

Figure 48 Architect Manager: Pipe-and-Filter View for Model Definition

5.2.1.3 Feature: Analyze Snapshot

The Eclipse platform allows defining visual containers that combine a set of views and editors within a predefined window – an Eclipse perspective. SAVE and its sibling – the architecture manager – share the same perspective in Eclipse, which means that the SAVE-related views are arranged in a predefined way. Figure 49 presents a screenshot of the SAVE perspective in Eclipse and highlights the different building blocks (each block is denoted with a capital letter in Figure 49):

- **A – SAVE Model Browser:** organizes the data of SAVE models in projects and offers a hierarchical representation. All related artifacts are presented (projects, models, views, etc.) and can be selected as the target of actions in a context menu.
- **B – Combined Visualization / Editor:** The main part of the perspective is the integrated visualization and editor. It has an engine for the visualization of software architectures offering a large number of graphical elements. One main feature is its configurability (i.e., enabling and/or disabling certain graphical elements), which allows users to adapt the visualization of results to their needs.
- **C – Legend:** This view explains the meaning of the graphical elements that are available in the current configuration. Due to the configurability, it is important to denote the current meaning of the graphical elements.
- **D – Outline:** A bird view on the whole model currently displayed in the visualization / editor. While the visualization / editor allows zooming and then only displays a small excerpt, the outline always shows the whole model and additionally highlights the excerpt currently in the editor visible with a transparent rectangle.
- **E – Filter Management, Decoration Management:** These views can be used to filter the information displayed in the visualization. Components and relations can be filtered according to several properties (e.g., their type). Additionally, the decorations that are shown to represent certain properties of components or relations can be filtered out in order to simplify the visualization and reduce the information to the amount needed in a specific situation.
- **F – Properties:** For the component or relation currently selected in the visualization, the external properties are displayed and can be modified (e.g., name).
- **G – Detail View:** For the component or relation currently selected in the visualization, additional information can be provided (e.g., about the internal hierarchical structure).
- **Hidden – Information View:** For any component or relation currently selected in area B, additional information on aggregated relations or the relation itself is provided (e.g., origin or destination code elements).

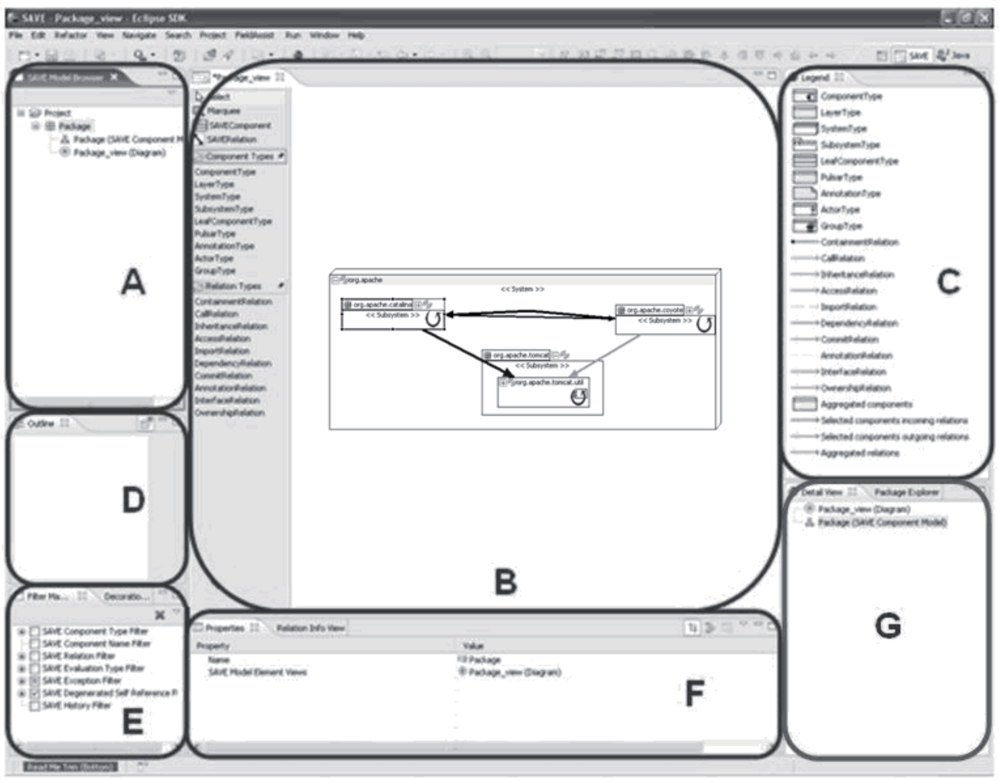


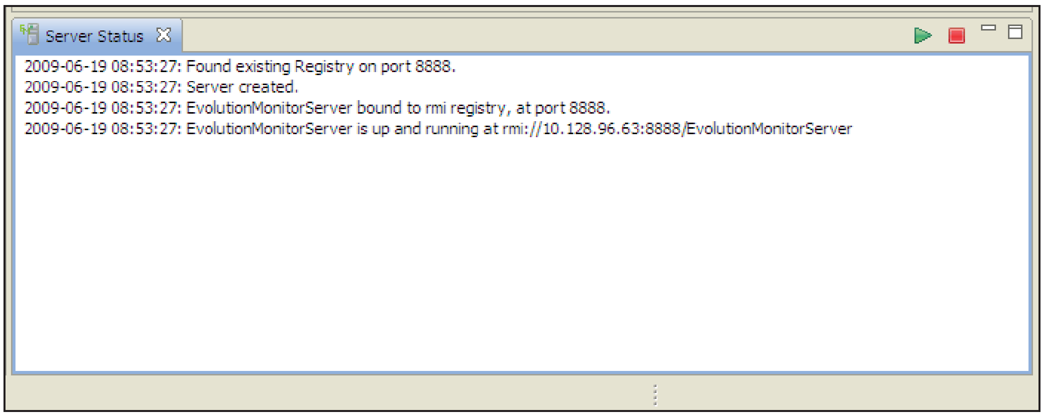
Figure 49 Architect Manager: Screenshot of User Interface

5.2.2 Server: Compliance Checker (Extensible Analysis and Communication Platform)

The compliance checker is the central server that realizes the extensible analysis and communication platform. It interacts with both the architect and the developers.

5.2.2.1 Feature: Start Server

Figure 50 depicts a screenshot of the console that is part of the compliance checker. The console outputs the events triggered for the compliance checker (either by the architect or the developers) and corresponding status including log messages on the arriving events and the reaction by the compliance checker. The console allows starting and stopping the server, the only two non-automated tasks.

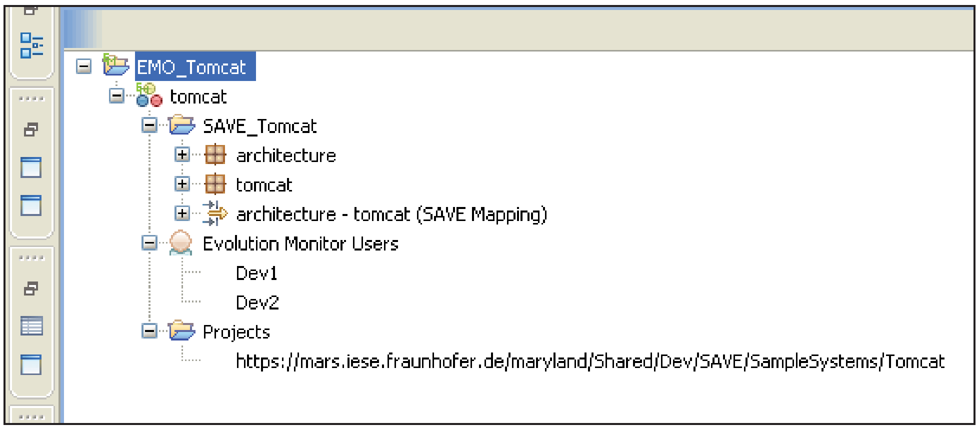


```
2009-06-19 08:53:27: Found existing Registry on port 8888.
2009-06-19 08:53:27: Server created.
2009-06-19 08:53:27: EvolutionMonitorServer bound to rmi registry, at port 8888.
2009-06-19 08:53:27: EvolutionMonitorServer is up and running at rmi://10.128.96.63:8888/EvolutionMonitorServer
```

Figure 50 Compliance Checker: Screenshot of SAVE Server Console

5.2.2.2 Feature: Browse SAVE Repository

Figure 51 depicts an example of the definition of the SAVE LiFe models. The repository defined by the architect monitors the Apache Tomcat web server. It comprises a structural model called *architecture*, a source code model called tomcat, and a mapping *architecture – tomcat*. Furthermore, two developers (*Dev1* and *Dev2*) have been specified to be monitored by SAVE LiFe and the name of the respective source code folder in the configuration management system (in this case Subversion) is specified under the Projects node in the tree in Figure 51. Once this information has been transferred to the SAVE LiFe server, live compliance checking starts automatically and proceeds continuously.



```
EMO_Tomcat
  tomcat
    SAVE_Tomcat
      architecture
      tomcat
      architecture - tomcat (SAVE Mapping)
    Evolution Monitor Users
      Dev1
      Dev2
    Projects
      https://mars.iese.fraunhofer.de/maryland/Shared/Dev/SAVE/SampleSystems/Tomcat
```

Figure 51 Compliance Checker: Screenshot of SAVE Repository Browser

5.2.2.3 Compliance Checking

Compliance checking is executed on the server for every change the developer clients made (see Figure 52 for the data flow depicted in a pipe-and-filter view). The multi-threaded execution of the compliance checking using the structural model defined by the architect, the temporary source code models produced by the fact extraction, and the mapping provided by the architects.

Compliance checking has a limited scope. The computed results are only transferred for the elements modified by the developers. The high frequency for every change and the limitation in the scope (only the delta of the source code has to be parsed) enable fast computation of the compliance checking results, which are then presented to the developers.

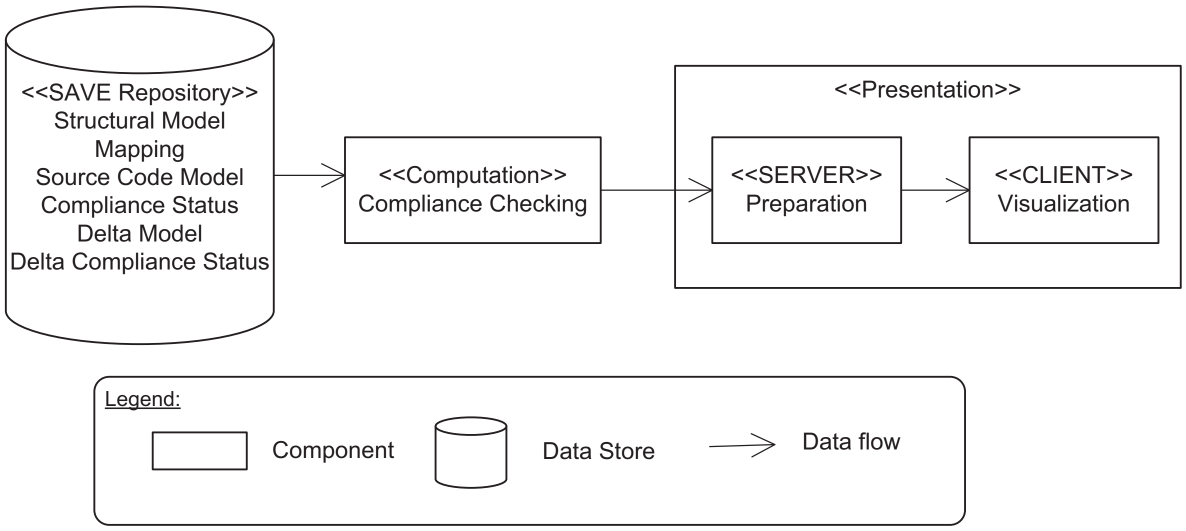


Figure 52 Compliance Checker: Pipe-and-Filter View of Compliance Checking

5.2.3 Thin Client: Development Monitor

The development monitor is the tool for the developers to interact with the compliance checker. It is realized as a thin client and comprises a minimal set of functionality, but supports many users in parallel at the same time.

5.2.3.1 Feature: Extract Delta Facts

The fact extraction in SAVE LiFe is depicted in Figure 53. The process is initiated by each modification to the source code made by a client. Hence, fact extraction is performed for every change made in the thin client; in other words, any changes developers make to their source code initiate the fact extraction, which eventually results in the presentation of the compliance checking results.

The SAVE LiFe platform hooks into the build process of the Eclipse platform for the respective source code language whenever the source code is built. When the platform starts the builder, it iterates over the elements in the developer's workspace (i.e., the workspace of the client) and checks whether or not they have been modified.

Determining whether or not a source code element has been modified is dependent on its status extracted from the source code configuration management system. The modified files are then transferred to the SAVE LiFe server. The SAVE LiFe server then parses the modified source code files and creates a temporary source code model. The temporary source code model contains the delta to the original source code model (see Section 2). The temporary copy of the source code model is based on an identical copy of the source code model as stored on the server. This server's source code model is based on the latest commit to the configuration management system. The temporary model integrates the deltas based on the parsing of the source code elements (i.e., the compilation units modified). The source code relationships of the modified elements replace the relationships of the pre-existing elements.

This integration of the modified source code elements allows initiating the execution of compliance checking on the temporary model. The results of compliance checking take into account the locally modified source code elements of each developer. Hence, the developers will receive feedback on their local modifications.

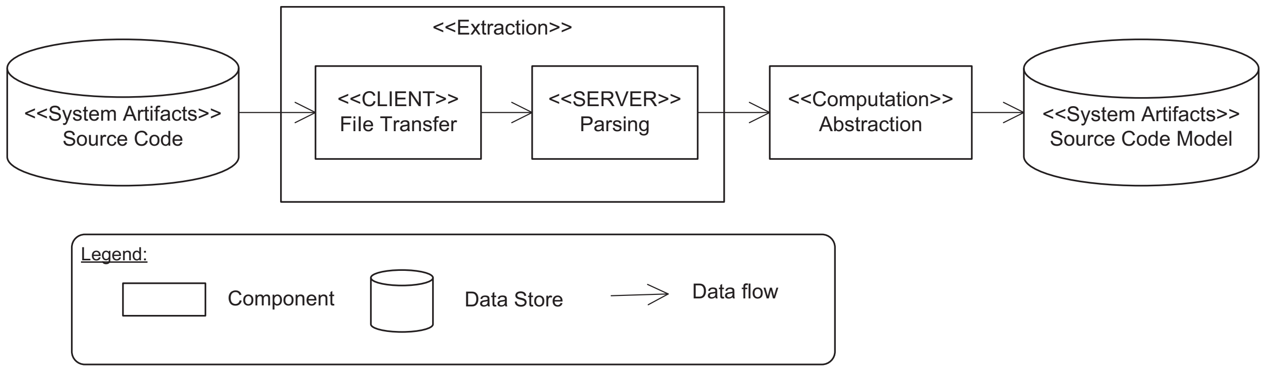
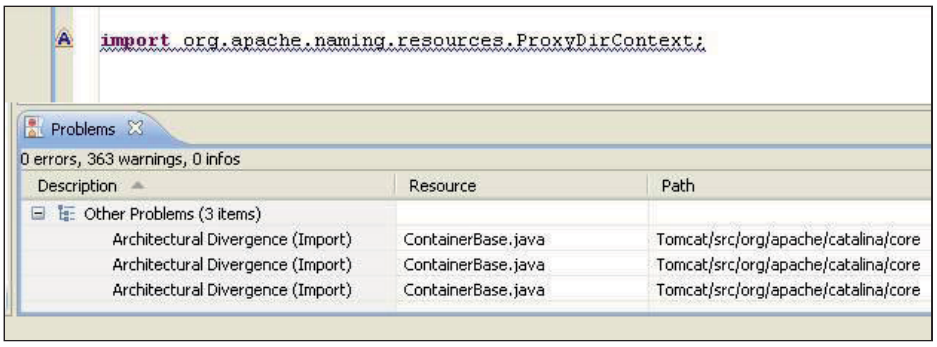

<<Extraction>>
<<System Artifacts>> Source Code
<<CLIENT>> File Transfer
<<SERVER>> Parsing
<<Computation>> Abstraction
<<System Artifacts>> Source Code Model
Legend:
Component
Data Store
Data flow

Figure 53 Development Monitor: Pipe-and-Filter View of Fact Extraction

5.2.3.2 Feature: Display Results (Live Feedback)

The presentation of the results to the developers is integrated into the development environment (see Figure 54 for two screenshots). On the one hand, the results of compliance checking are available in a tabular format listing the spots in the source code that cause violations (see lower part of Figure 54). On the other hand, the presentation uses Eclipse problem markers to indicate source code lines that cause a structural violation (see the upper part of Figure 54). The "A" in front of the source code statement indicates the structural violation. When the mouse is moved over the architecture violation icon, additional information about the structural decomposition is displayed (i.e., the containing architectural elements of source and target of the violations are listed). The usage of problem markers for highlighting structural violations enables the Eclipse quick-fix functionality of the editor (e.g., removing the violation with a mouse click, or triggering a move of the enclosing source code element refactoring). The visualization of compliance checking results notifies the developers about structural violations in the source code they are currently editing. The presentation of the results is calculated in the background in a non-intrusive way. As soon as the results are available, the editor places the overlay icons for the problem markers. Furthermore, the problem markers are placed in the source code explorer of Eclipse, which allows top-down navigation to the elements that cause violations.



```
import org.apache.naming.resources.ProxyDirContext;
```

0 errors, 363 warnings, 0 infos

Description	Resource	Path
Other Problems (3 items)		
Architectural Divergence (Import)	ContainerBase.java	Tomcat/src/org/apache/catalina/core
Architectural Divergence (Import)	ContainerBase.java	Tomcat/src/org/apache/catalina/core
Architectural Divergence (Import)	ContainerBase.java	Tomcat/src/org/apache/catalina/core

Figure 54 Development Monitor: Screenshot Display of Compliance Checking Results

5.2.4 SAVE

In addition to live compliance checking, the Fraunhofer SAVE tool realizes a set of compliance checking algorithms for structural or behavioral views and for implementation variants. It operates on implementation snapshots and mines the source code for relevant data constituting the resulting source code model. Most relevant fact extraction handles software systems implemented in Java (compatibility Java 1.6 and earlier version), C/C++, Delphi, J2EE-specific Java extensions (including JSP), several intermediate representations, and importers for architectural models with XML-based file format. For detailed information on SAVE, please refer to [Knodel 2009a].

5.3 Realization of Live Compliance Checking Requirements

This section traces the principles of live compliance checking (see Section 1.3) and the essential requirements on the realization (see Table 4, as well Section 1.3) to the solutions provided by SAVE LiFe.

Requirement	Solution
Live Feedback	The development monitor (SAVE LiFe thin client) receives feedback from the server. The feedback is live because fact extraction, lifting, and compliance checking are limited to the local scope (i.e., the deltas) only.
Ease of Use	The architecture violations are visualized with a special icon using Eclipse problem markers. The violation marker is annotated with context information on the violation. Perceiving the violation and interpreting it is a simple, trivial task. However, resolving the violation still might be complicated.
High Execution Frequency	The constant and continuous sending of all deltas by each developer results in high execution frequency. The central server computes the compliance checking results and enables sending live feedback to the developers.
Delta Analysis	The development monitor sends deltas and receives live feedback on their compliance. The feedback is always directed at the originator.
Distributed Team Support	The client-server-client architecture allows scaling for arbitrary team sizes. The communication channels operate logically over the network, so the encapsulated technical protocols managing the connections between clients and server can be adapted on demand.
Smooth Integration into Environment	SAVE LiFe is fully integrated into the Eclipse IDE. It takes advantage of other third-party plug-ins and applies several Eclipse best practices (e.g., the problem marker concept for displaying violations). The display of compliance results allows developers to perceive the feedback non-intrusively without distracting them from their current implementation task.
Robustness	Fact extraction in SAVE LiFe allows parsing of non-compiling and incomplete source code, with the parser mining relevant data to the largest extent possible but ignoring non-compiling statements.
Commit Control	SAVE LiFe allows committing source code still comprising violations. We opted for loose control in order to have the possibility of committing explicit violating exceptions to the configuration management.
Separation of Roles: Architect and Developer	The two roles are clearly separated by the existence of two distinct clients: the architecture manager for architects and the development monitor for developers. Both communicate with the central server.

Table 16 Realization of Essential Requirements on Live Compliance Checking

5.4 Technical Solution

The technical solution comprises two clients – architecture manager and development monitor – and one central server – the compliance checker. The client-server-client communication transfers data from clients to server and vice versa. To communicate SAVE LiFe requires distributed data structures, which can be transferred from clients to server and vice versa. In particular, SAVE LiFe uses the following distinct data models:

- **Structural Model:** The structural model captures the intended decomposition specified by the architect (see Section 2.1).
- **Source Code Model:** The source code model captures the static structure of a system at development time (see Section 2.2).
- **Mapping Model:** The mapping model (see Section 2.3) defines the relation of architectural elements to source code elements and vice versa.
- **Delta Source Code Model:** The delta source code model is an instance of the source code model, which is created for the locally modified delta of the developer. Thus, the meta-model of the source code (see Section 2.2) also serves as meta-model for the delta model.
- **Compliance Status Model:** The compliance status model is an annotated structural model, which discloses the architecture violations of the system under evaluation. Thus, the meta-model of the structural model (see Section 2.1) also serves as meta-model for the compliance status. The compliance status annotates convergences, divergences, and absences, as computed by the compliance function (see Section 2.4).
- **Delta Compliance Status Model:** The delta compliance status model is an instance of the source code model, which is created for the locally modified delta of the developer annotated by the compliance status drilled down to the source code model using the mapping (see Section 2.3). Thus, the meta-model of the source code (see Section 2.1) also serves as meta-model for the delta compliance status. The same annotations – convergence, divergence, and absence (as defined in Section 2.4) – hold for the delta compliance status.

The data models are used in the mode of operation of the client-server-client system, in particular the compliance checker. The steps correspond to distinct methods described in Appendix D using pseudo code:

- **Algorithms SAVE LiFe Fat Client – Architecture Manager:** The architect executes the methods *publishArchitecture()* and *requestComplianceStatus()* on demand. Both methods are accessible from the user interface of the architecture manager (see Appendix D, Section D.1).

- **Algorithms SAVE LiFe Thin Client – Development Monitor:** The development monitor tracks the work of each developer in the integrated development environment individually. The development monitor hooks into the incremental project builder of the development environment (e.g., for Eclipse *org.eclipse.core.internal.events*). The builder is executed whenever physical resources (i.e., files or folders) are changed and saved. When executed, the method *monitorCodeandSendDelta()* of the development monitor is invoked automatically. Hence, the method is executed for any change made to the source code. The server – after having computed the results – invokes *receiveLiveFeedback()* remotely for the respective developer to transfer the results, which potentially include violations (Appendix D, Section D.2).
- **Algorithms SAVE LiFe Server – Compliance Checker:** The methods of the compliance checker are remotely triggered by the respective client. The architecture manager invokes either *receivePublishedArchitecture()* or *publishComplianceStatus()*, while the development monitor invokes *sendDelta()*. After compliance checking has been executed, the server invokes *receiveLiveFeedback()* in the development monitor so developers become aware of the violations – if present – promptly (Appendix D, Section D.3).

The algorithms as described in Appendix D realize the requirements for live compliance checking as stated in Table 16.

5.5 Summary

SAVE LiFe realizes the idea of providing developers with live feedback on architecture compliance – live compliance checking. It is a client-server-client tool, which extends the snapshot analysis tool Fraunhofer SAVE.

On the one hand, SAVE LiFe provides a basic communication platform, which allows running arbitrary analyses with live feedback. On the other hand, it presents the first implementation of live compliance checking. This live compliance checking implements an adapted version of the Reflexion model technique.

SAVE LiFe extends the standalone SAVE tool towards a client-server-client system consisting of the client architecture manager, the client developer monitor, and the server compliance checker. Live compliance checking has been enabled as an analyzer using the client-server principle to support multiple developers working in distributed teams. Architects can update the structural model at any time, which then immediately serves as new input to live compliance checking.

Enabling other compliance checking techniques (e.g., dependency rules, which are already a feature of SAVE) based on SAVE LiFe is easily possible due to extension mechanisms of the platform. The principles of live feedback would then apply as well, as just the server-side computation algorithm of the results would change. The SAVE LiFe platform supports distributed development teams with multiple developers. It is hence a tool that supports a whole development organization in achieving compliance by construction.

6 Validation

Live compliance checking as proposed in the previous sections of this thesis aims at the direct goal of reducing structural violations in software implementations. This reduction causes two effects: first it leads to further effort savings in compliance achievement due to the learning effect and the prompt removal effect, and second, it leads to improved productivity in the lifecycle of the software system due to reliable evolution management based on the architecture, where compliance ensures the traceability between the two abstraction levels.

We validate on the one hand the positive effects of live compliance checking (see Section 6.1). An experiment shows that live feedback indeed reduces the number of structural violations in the implementations. In this case, we compared six teams comprising a total of 19 developers realizing components of similar size and spending roughly the same average effort per developer. Three teams received support by SAVE LiFe, while the other three teams – the control group – applied an ordinary development approach (i.e., without SAVE LiFe). Both groups developed a software system over a period of 35 days encompassing the lifecycle phases implementation, integration, and testing. The experiment showed that the group supported by SAVE LiFe had 60% less architectural violations throughout than the control group. Hence, the experiment provides evidence of the positive effects of SAVE LiFe on compliance. The developers of the groups supported by the live compliance checking feature spent roughly the same effort in all phases as the developers in the control group.

If we imagine repairing the structural violations contained in the above-mentioned components, we can see the advantages live compliance checking has over its regular analytical sibling:

- The number of violations is reduced by 60%, which eventually would mean 60% less items to repair. This results in a substantially lower compliance achievement effort.
- Further, there is no need for workshops or meetings where the architect explains the repair tasks to the developers. All developers already know about the violations because they see the violations plus context information highlighted in their own source code editor.
- The compliance feedback is directed and tailored to the originator or local expert. Developers responsible for a set of compilation units receive only feedback on their local modification scope. There is no

need to first dismantle the bulk of violations to identify who would be the responsible developer for carrying out the repair.

On the other hand, we have to show that compliance positively impacts the evolution of software systems. Therefore, Section 6.2 revisits the industrial case A (see Section 1.2.1), where the development organization – after observing a downward trend in the degree of compliance – decided to institutionalize compliance checking as a part of their quality engineering strategy using Fraunhofer SAVE. The organization invested a large amount of effort to restructure the implementation in order to achieve compliance. It now applies analytical compliance checking at relatively regular intervals. This restructuring feedback (although not live) could raise the degree of compliance degree by up to 98% over time.

In a second case, Fraunhofer IESE supported the architectural re-design of an industrial system. The re-design imposed a completely different structural decomposition and component hierarchy for the software implementation. After adapting the source code towards this new decomposition, there were initially about 46% violating dependencies. Again, we gave regular compliance feedback (and again it was not live) using Fraunhofer SAVE and could observe a compliance increase to 95% over time.

We confirmed the sustained compliance in these two cases and could also observe an increased productivity. The development organization could produce and, evolve more systems at the same time with the same effort as before. The industrial stakeholders confirmed in interviews that compliance is at least one crucial factor (though not the only one) for this productivity gain.

6.1 Feedback by Live Compliance Checking

Our research goal was to analyze the effects of live compliance checking on the compliance of component implementations. Therefore, we conducted an experiment monitoring the implementations of student development teams during a practical course lasting five weeks (i.e., 35 days in total). The experiment aimed at verifying the implementation (i.e., the system was built right, as intended) but not at validating the architecture (i.e., the right system was built and could achieve all requirements). In the experiment, we hence assumed that the architecture was well-designed. A detailed description of the experiment including the material can be found in [Knodel 2008d] and [Rost 2007]; the experiment material is listed in Appendix E.

6.1.1 Setup Experiment GSE2007

6.1.1.1 Hypothesis

Our main assumption is that live compliance checking allows the prompt removal of structural violations. The constant and live feedback raises the developers' awareness regarding the structural violations. It is raised immediately, which enables the developers to remove the violations promptly. Hence, our hypothesis is:

- **$H_{C0.1}$** – The null hypothesis is that live compliance checking with direct feedback has no impact on the number of violations.
- **H_{F1}** – Live feedback on compliance reduces structural violations in implementations. Live compliance checking causes components to have less structural violations than if they had been developed in regular development.

6.1.1.2 Operationalization H1

We measured a-posteriori the number of architectural violations for the implemented system for each component development team. We classified the teams into being part of either the support group or the control group. We quantified the architectural violations by the number of divergences in the source code of each component analyzed, applying the Reflexion model technique to compute the divergences.

6.1.1.3 Subjects

The subjects of the experiment were Bachelor and Master students of the Technical University of Kaiserslautern. A total of 19 students participated in the SAVE LiFe experiment. The participants' level of experience in software engineering and development projects varied from having been involved in one or two development projects to more than five. Their level of experience in the domain of the practical course also varied, from low to very high, but was balanced throughout the groups.

6.1.1.4 Context

The experiment was conducted in the context of a practical course at the Technical University of Kaiserslautern. The course is attended by Bachelor and Master students. The GSE 2007 course comprised, among other phases, four weeks of architecture and component design, three weeks of implementation and two weeks of test and integration. The domain of the system developed in GSE 2007 was ambient intelligence. After the completion of the course, the software system developed by the students, consisted of 4377 lines of code, 90 classes, and 15 interfaces in 25 packages.

6.1.1.5 Experimental Materials

The students worked in a laboratory at the university, each on their own computer. On every computer, two configurations of Eclipse were installed, one with enabled support of the SAVE LiFe, the other one without support. A SAVE LiFe server was installed on a separate machine, continuously running and reachable over the local network.

6.1.1.6 Experimental Design

The students worked distributed in six component development teams of the software system. Table 1 lists the components by name, the development team size, and the teams that had support by live compliance checking.

Component Name	**Number of Developers**	**Live Compliance Checking Support**
amiCAInteraction	3	yes
Synchronization	3	yes
Controller	5	yes
Persistence	2	no
LocationManager	2	no
UserInterface	4	no

Table 17 Live Feedback Experiment: Component Teams

6.1.1.7 Experimental Tasks

Within the GSE practical course, the students developed a system in the ambient intelligence domain. Therefore, they executed a development process, starting from requirements engineering via architecture and component design and implementation to test and integration. The experiment was started with the implementation phase and monitored all subsequent phases (i.e., implementation, refactorings, testing, and integration). Hence, the experimental task for each team was to implement the respective component. The duration of the development monitored was a total of five weeks (i.e., 35 days) and ended with the deployment of the system.

6.1.1.8 Data Collection

The source code of all component teams was managed by a central configuration management system – Subversion. The Subversion server stored the history of the project development in the form of revisions. This means that whenever a new change was committed to the server, this change was assigned a revision number and allowed restoring the state at a later point in time. Additionally, we enabled logging in the SAVE LiFe server to log each live compliance checking execution. Besides these, no other means of data collection were necessary during the project. To investigate how the students perceived the support by SAVE LiFe, we asked them to fill out a questionnaire.

6.1.1.9 Data Analysis

We measured the number of architectural violations a-posteriori each day for the last commit to the Subversion server. For this purpose, we restored the respective system state for all six components. The specified architecture was created during the design phase by the students themselves.

6.1.2 Results

This section describes the results obtained from analyzing the collected data and the questionnaire.

6.1.2.1 Structural Violations

To observe the evolution of the software architecture, one state per development day was analyzed and the number of architecture violations has been computed. Figure 2, Figure 3, and Figure 4 show the resulting numbers of violations captured. In all three figures, the abscissa shows the period of the project development in days (in total 35 days). The ordinate shows the number of divergences – the architecture violations.

Figure 55 depicts the number of violations per component. The components *amiCAInteraction*, *LocationManager*, and *Persistence* are mainly used by the other components. In the architecture, they are rather utility or library components, which are self-contained and have almost no outgoing relations. This is reflected in Figure 55 by showing 0 violations (except for the component *amiCAInteraction* during the last days). The components *Controller*, *UserInterface*, and *Synchronization* constitute the major part of the relations in the system. They also cause almost all violations, as shown Figure 55. The divergences of the control group are all produced by the component *UserInterface* component, those of the supported group by the components *Controller* and *Synchronization*.

Figure 56 aggregates the number of violations for both the control group and the group supported by live compliance checking. In the beginning of the implementation phase, the number of architecture violations produced by the supported group as well as by the control group increased. However, after 13 days of development, the number of architecture violations was reduced by more than 50%, whereas the value of the control group remained constant. Later there was only a slight reduction for the control group; the value of the supported group is constantly about 60% lower than that of the control group. In the end, there was a peak of the divergences of the supported group. We assume that this peak results from the integration phase before the final deployment of the system.

Next to the violations for both groups, Figure 57 depicts the total number of dependencies in the system in order to visualize the growth of the system. The major part of the dependencies were implemented in the first 13 days and then stayed more or less on a constant level with a peak at the end (the integration phase). We observed that when the number of violations in the supported group decreased, the number of overall relations also decreased (but less than the number of divergences). As the number of divergences stayed almost constant for the control group, we consider this to be an indicator that the supported group refactored parts of the system, which made the components' implementation more compliant.

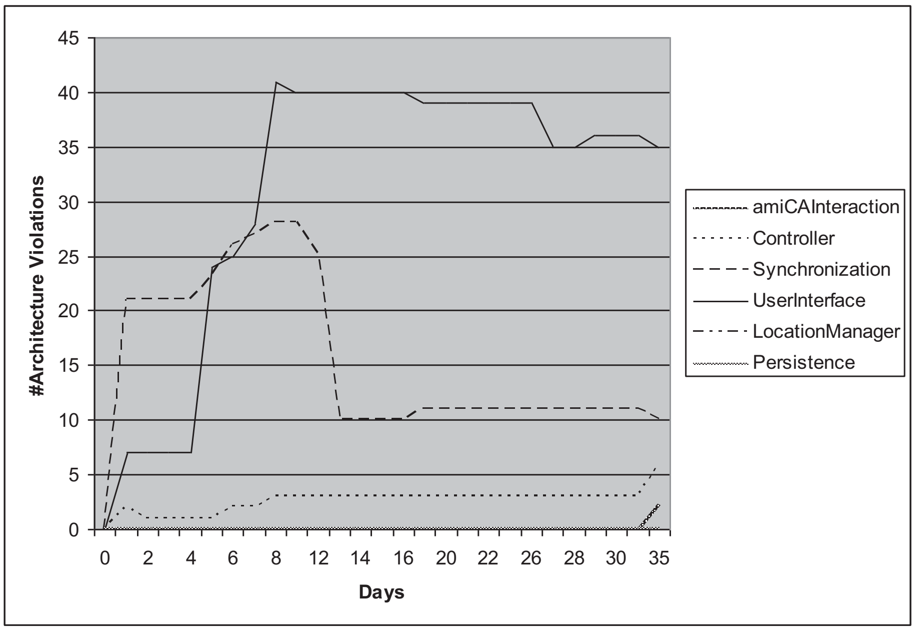


Figure 55 Live Feedback Experiment: Architectural Violations per Component

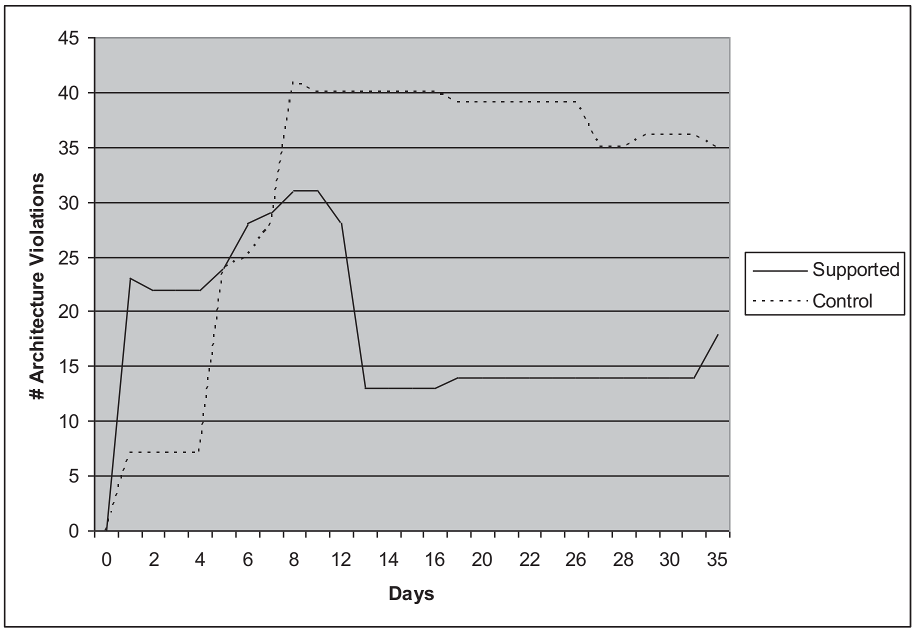


Figure 56 Live Feedback Experiment: Architectural Violations Aggregated by Supported and Control Group

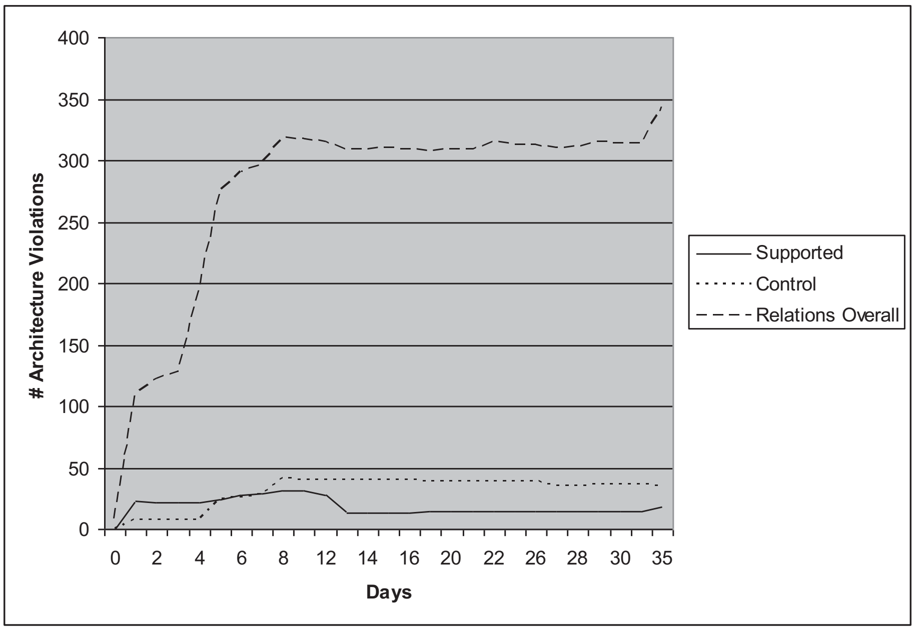


Figure 57 Live Feedback Experiment: Architectural Violations and Total Relations

6.1.2.2 Debriefing Questionnaire

To measure transfer success (i.e., the acceptance of a newly introduced technology or software), a survey was conducted, asking the participants of the supported group 21 questions about their experiences of working with SAVE LiFe. The questionnaire was designed to address the idea of innovation transfer success factors as introduced by Green and Hevner [Green 2000]. These factors are shown in Figure 58. Several sub-factors are grouped into major categories (boxes), with influences on other major categories, indicated by the arrows (e.g., "Target Environment" influences "Perceived Control"). All five categories directly or indirectly influence the transfer success of an innovation – which in our case was SAVE LiFe.

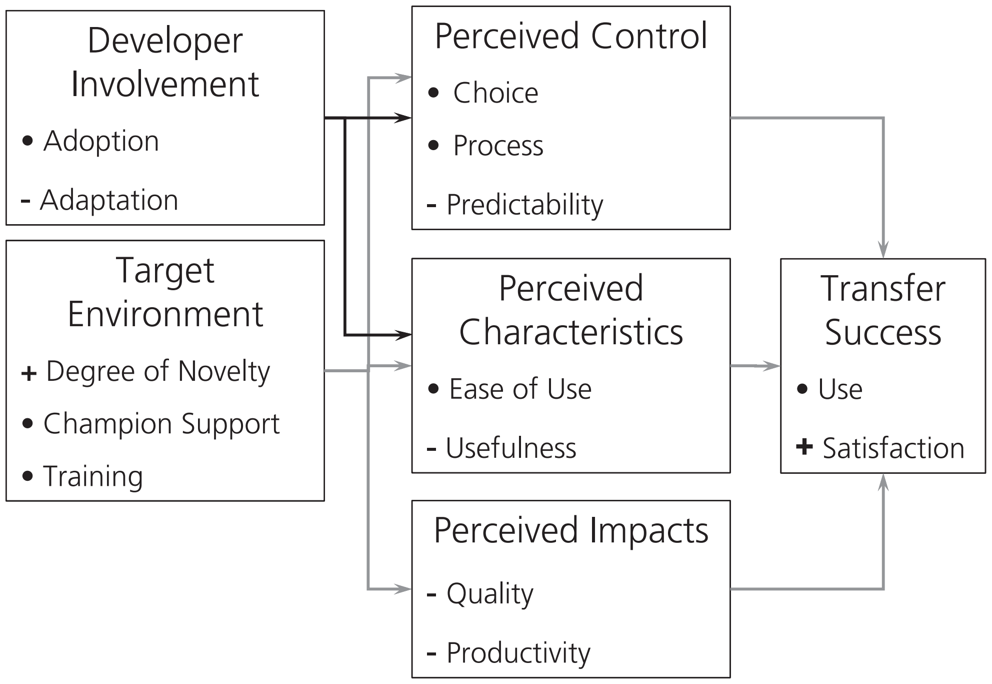


Figure 58 Live Feedback Experiment: Transfer Success Factors

The possible answers were organized in the form of a four point Likert scale, with the alternatives "strong disagree", "disagree", "agree", and "strong agree". To be able to calculate the trends, a scale from -2 to 2 was assigned to each alternative. The number of answers for each alternative was then multiplied by the corresponding range number and the average was calculated. We consider a value of lower than -0.5 as a negative, and higher than 0.5 as a positive trend. The analysis of the answers given by the students showed positive and negative trends in different factors. Figure 58 depicts positive trends with a "+" (degree of novelty, satisfaction) and negative trends with a "–" (adaptation, predictability, usefulness, quality, productivity). The other factors (adoption, champion support, choice, process, ease of use, and use) were rather balanced. The following list presents a discussion for these trends:

- **Adaptation:** Adaptation is defined as the development and installation of the IT innovation [Green 1999], which means the familiarization of the user with the new technology. So the question in this context was whether or not the developers used the computation results provided by SAVE LiFe to improve compliance. The answers given indicate that this was not the case, which might possibly result from the reasons given for the factor predictability, which may also have caused this negative trend.

- **Degree of Novelty:** Degree of Novelty refers to the extent to which the learning and use of the IT innovation represents a new experience to the user [Green 1999]. The answers show a clear positive trend, indicating that the developers were satisfied with the usage and user interface of the SAVE LiFe.
- **Predictability:** The factor of Predictability is defined in [Green 1999] as the predictability dimension of control. It refers to knowing what event(s) will occur and when, and not necessarily to controlling the event itself, meaning whether the students could anticipate the results computed by SAVE LiFe. The analysis of the questionnaire showed that the results were not predictable for the participants. Since all groups were doing all the tasks of the development process, but for their own components, including architecture design, the students might not have had an idea of the target architecture as a whole, making the evaluation results hardly predictable. This might also have caused a feeling of dissatisfaction, possibly also causing the negative trend in the adaptation trend.
- **Usefulness:** Perceived usefulness refers to how reasonable the application of the new technology is seen, based on the provided support. The participants thought that the compliance checking did not help to improve the implementation. Hence, the compliance checking results were considered as rather useless.
- **Quality and Productivity:** These factors refer to the perceived impact of the introduction of the new technology. The questions were whether SAVE LiFe helped to write architecture-compliant source code and therefore led to an improved architecture, and whether the results helped to save time in later refactorings. In these factors, there a negative trend was also identified, which is probably also based on the students not understanding the benefits of SAVE LiFe. However, no negative impact for quality and productivity was indicated, since there was no negative one in the adoption of SAVE LiFe.
- **Satisfaction:** Although the overall trend was rather negative, the transfer success shows a positive trend for the factor satisfaction. The reason for this might be that the participants agreed with the intended purpose and capabilities of SAVE LiFe, although they might not have used the support for repairing all violations.

6.1.3 Threats to Validity

The threats to validity cover three categories: internal, construct, and external, as proposed in [Wohlin 2000].

6.1.3.1 Internal Validity

Internal validity is the degree to which independent variables have an impact on dependent variables. The following threats to internal validity were identified:

- Due to the design of the GSE practical course, there was no explicit architect for designing the architecture and managing the realization. The students themselves took the roles of the architect and the developers as they designed the architecture and realized it. Therefore it cannot be guaranteed that the target architecture was correct with respect to the intended purpose and structure of the system and, since nobody controlled the realization, it is not sure whether the students perceived an incentive in being compliant to the architecture.
- The number of participants in each component development team was rather small, as was the developed system, which makes it difficult to generalize the observed results for mid- or large-scale projects with more developers involved.
- Due to the rules of the GSE practical course, the students were not forced to work in the laboratory and were therefore not forced to use SAVE LiFe. Although the server log files provided information about the usage, the fact the use was not enforced might affect the generalizability and thus, the validity of the analyzed results.
- The participants may have had different levels of experience of using the Eclipse IDE for developing software projects, which might have made it difficult for students having less experience with Eclipse to use the live feedback results. This is also true for different levels of experience and for understanding the idea of software architecture. To reduce the effects of these threats, we explained the usage of SAVE LiFe to the participants in form of a presentation and of a tutorial.

6.1.3.2 Construct Validity

Construct validity is the degree to which the settings of the experiment in terms of the dependent and independent variables reflect the goal of the experiment. The following possible threats were identified:

- As the distribution of the development groups into experiment groups was random, this might have resulted in an uneven distribution of experience levels and might therefore have caused a bias of the analysis results because high architecture quality might result from the participants' experience level rather than from the SAVE LiFe support. To mitigate this threat, the questionnaire asked about the experiences of the students. The analysis of this data did not reveal an experience advantage.

- The participants knew about the fact that there were two different groups using different versions of Eclipse. This might have created a bias for the students. Due to the design of the GSE course and the environmental constraints, there was no way to separate the members of the two groups for the duration of the experiment.
- It is not guaranteed that the questions asked in the debriefing questionnaire are the right ones to make a statement about the transfer success of SAVE LiFe. To mitigate this threat, the questionnaire was designed according to the idea of transfer success factors.

6.1.3.3 External Validity

External validity is the degree to which the results of the experiment can be transferred to other people and to changed environmental settings.

- The participants might not be representative. Since all participants were students, the results might not be representative for industrial practice. To make a statement about the usage of SAVE LiFe in an industrial context, the experiment might have to be replicated with professional software developers as participants.
- The task might not be representative. On the one hand, the system developed by the students belongs to a relatively new area of domain, namely ambient intelligence, and therefore does not reflect typical commercial software projects. On the other hand, the students' interest to in achieving an implementation in compliance with the architecture might possibly have been low, as they did not continue working on this project after the GSE course had finished. To overcome these threats, we envision a replication of the experiment in a real project.
- Green-field development happens rather seldom in industrial practice; typically, systems are not developed from scratch but evolved from existing reusable components, existing source code, or from migration. This was not the case in the GSE practical course: a completely new system was developed. To overcome this threat, a replication of the experiment should be conducted in the context of an evolving software project that is not being developed from scratch.

6.1.4 Conclusion

The experiment investigated the impact of live compliance checking on the implementation of component teams. The analysis results showed that the number of architecture violations were, after an initial peak, almost 60% lower throughout in the group that was supported by SAVE LiFe, compared to the control group. We observed a decrease in the number of architecture violations. The violations in two complex components of the supported teams were significantly lower than in those of the control teams. We consider this as an indicator for the positive impact of SAVE LiFe. The debriefing questionnaire showed negative trends in several success factors, mostly caused by the weak adaptation of the developers. This is most presumably grounded in the nature of the project, which was part of a practical course, and we can therefore assume that the main goal for the students was to get the system running, and only minor interest existed in producing a high-quality implementation that was compliant to the architecture. Additionally, we think that the participants' focus was not on the target architecture, or that they did not have the basic concepts of their target architecture in mind, or that there was a deficit in understanding architecture violations, when they implemented the system, which was shown in the questionnaire results where the students indicated the results were not predictable.

Based on the aspects mentioned above and on the threats to validity, we plan to replicate the experiment. We aim at getting deeper empirical insights into the effects of live compliance checking. Ideally, such a replication would include several modifications:

- The project should not be short term only. The developers should have an interest in the future of the system (because they will have to maintain it).
- Ideally, several component teams would implement the same component as specified in the architecture.
- The number of component teams should be increased to obtain statistically significant results.
- The participants should preferably be professional developers, with almost the same level of experience with software architecture, or at least the distribution into experiment groups should ensure that the number of professionals and inexperienced participants is the same for both experiment groups.
- The two experiment groups should not know of each other and should develop the same product according to the same architecture. Also, the development of the whole system should be monitored, not only the components.
- A software architect (and not the developers themselves) should be responsible for defining the architecture and track its realization.

- SAVE LiFe support should be mandatory during the whole experiment.

Although there is a clear need for replication, the experiment provided initial evidence about the benefits of live compliance checking. The results showed a reduction in the number of violations for the supported development teams. All teams invested approximately the same development effort for developing top-level components. The supported teams caused less architectural violations, and, hence, when refactoring the implementation of the system to achieve architecture compliance, the supported teams would have about 60% fewer items to refactor. The fewer refactoring items would eventually result in effort savings for refactorings.

We claim that live architecture compliance checking has a positive effect on the implementation of software systems. We think that prompt removal of violations (as opposed to late removal when applying analytical compliance checking) reduces the overhead effort. The results of this experiment are a first data point to corroborate our claim.

6.2 Benefits of Regular Compliance Feedback

In two case studies, we gave regularly repeated (though not live) feedback on the compliance of the software implementation and observed the compliance status over time. The research question for the two industrial case studies presented in this section was whether or not a high degree of compliance feedback has a positive impact on the overall evolution of the software systems. Hence, our hypothesis was:

- **$H_{F0.2}$** – The null hypothesis is that compliance has no impact on the productivity of an organization.
- **H_{F2}** – Compliance increases the productivity of the development organization.

6.2.1 Product Line of Climate and Flue Gas Measurement Devices

6.2.1.1 Context

In [Knodel 2008b], we reported on the continuation of case A as presented in Section 1.2.1 with the product line of climate and flue gas measurement devices developed by Testo. The development organization decided to integrate analytical compliance checking into their quality engineering strategy. The results were presented in joint workshops at major project checkpoints. In the beginning, compliance checking was offered by Fraunhofer IESE as a service and was conducted offline. Currently, the Testo architects are conducting compliance checking independently and on demand by using the Fraunhofer SAVE tool [Knodel 2009a].

Here we summarize the experiences with architecture compliance checking at Testo. In total, 15 different instances of the product line (i.e., distinct Testo products delivered to the market) have been checked regularly over a period of more than two years (as described in Section 5).

6.2.1.2 Compliance Status

The Testo reference architecture, like every product line, consists of two parts: the application-specific implementation and the family-specific implementation with generic components, in Testo's case called "framework (fw)". Figure 59 depicts this principle, which holds for every instance of the Testo product line of climate and flue gas measurement devices. The arrow in Figure 59 indicates that the product-specific implementation parts are allowed to use the framework. On average, the Testo products achieve a reuse degree of about 40% (i.e., the framework comprises approximately 40% of each product line instance). The values have been measured with various size metrics like lines of code (LoC), number of framework files used, number of framework functions used. The size of the Testo measurement devices ranges from 10 KLoC to 600 KLoC; all products have been implemented in the C programming language.

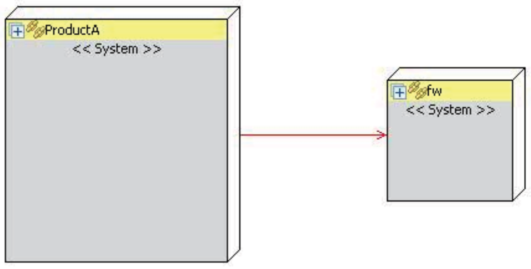


Figure 59 Structural Model: Framework Usage

Orthogonally to the framework, layering was established; however, no strict layering was enforced. The layers crosscut both the framework and the product-specific implementations, with one exception: the layer "ui" is purely application-specific. Figure 60 shows the layering of the Testo product line of climate and flue gas devices as initially specified. Figure 60 depicts the adaptations of the layered architecture. Subsystems and components contained in layers have been filtered out with one exception: For the component "Display" in the layer "hc", it was decided to make it visible in the layering, since all elements of the "ui" layer are allowed to access this component, but no other elements in the "hc" layer are. To compute the compliance checking results, we used the Fraunhofer SAVE tool. Figure 61 depicts sample compliance checking results and shows the results using overlay icons (convergences as check marks, divergences as exclamation marks, and absences as "X" icons).

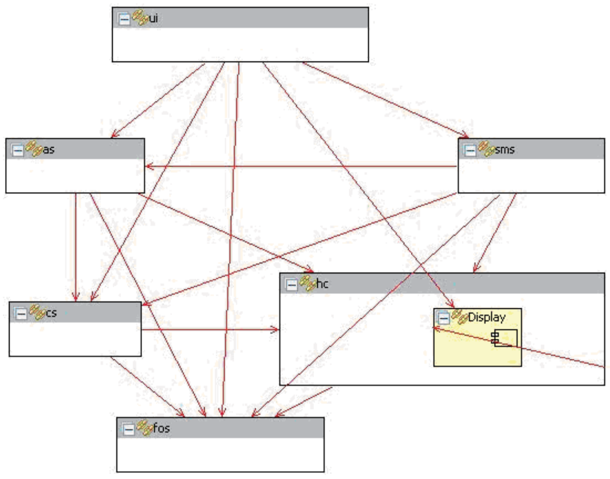


Figure 60 Structural Model: Layering

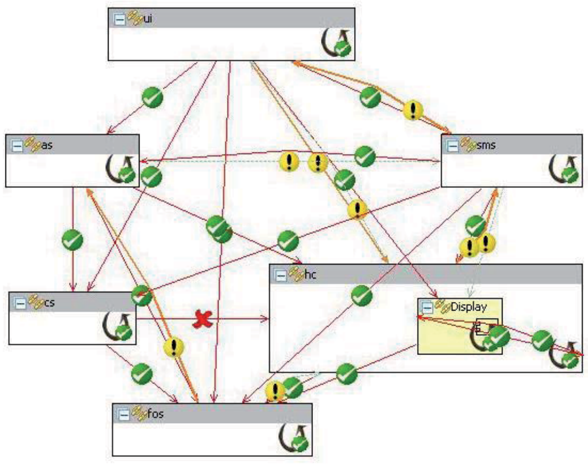


Figure 61 Compliance Status: Visualization of Convergences, Divergences, and Absences

Table 18 shows the total number of convergences (# of conv.), divergences, and the degree of compliance (% compliant) for the products. Since absences mainly express dependencies not instantiated for a variant of the product line, we excluded them from the table. Table 18 presents the detailed analysis results and lists the numbers of convergences and divergences to the layering as depicted in Figure 60. The number of the products roughly indicates the order of development despite the fact that some products were developed concurrently. The first column in Table 18 shows the evaluation date when compliance checking was conducted. For each product evaluated at an evaluation date, there are two rows: The first row lists the number of convergences and the number of divergences (the sum is the total number of dependencies within a product). The second row computes the degree of violation: The number of divergent relations is divided by the total number of relations. Although the products are different regarding their features, their size, and their developers, the ratio of divergences within each product is comparable.

Compared to the first three products described in Section 1.2.1, we can see that, except for product P1, all compliance degrees are higher (i.e., compliance was 95.7% for P1, 89.8% for P2, and 72.7% for P3). At the evaluation date (2006-08), Fraunhofer IESE produced the compliance checking results, this time for products P4 to P10. All these products comprised a percentage of divergences of less than 5% (P9 is the only exception with less than 10%). It can be observed that the total number of divergences significantly decreased afterwards. For the evaluation date 2006-10, compliance checking was repeated for products P4 to P10 and was newly conducted for P11. The percentage of divergences decreased for all products evaluated, with the exception of P8 and P10, where a slight increase was found. Especially the compliance of product P11 is remarkable, since mainly new employees were involved in the realization of this product. The last two evaluation dates (2007-03 and 2007-10) were conducted independently by the Testo architects and discussed internally. However, the compliance checking results were shared with Fraunhofer IESE. Updated evaluations of products P4, P5 and P11 have been made, and products P12 to P15 were newly evaluated. It can be observed that the percentage of divergences did not exceed 5%. The exception of product P4 (5.8% at evaluation date 2007-03) was counteracted, as the results for the date 2007-10 show.

Date	**Status**	**Products**											
		P4	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**	**P14**	**P15**
2006-08	# of Conv.	**3824**	**1729**	**293**	**337**	**854**	**205**	**1990**	n/a	n/a	n/a	n/a	n/a
	# of Div.	**83**	**33**	**6**	**18**	**21**	**16**	**92**					
	% compliant	**97.9**	**98.1**	**98.0**	**94.9**	**97.6**	**92.8**	**94.6**	n/a	n/a	n/a	n/a	n/a
2006-10	# of Conv.	**4571**	**1044**	**313**	**398**	**1105**	**285**	**1810**	**3968**	n/a	n/a	n/a	n/a
	# of Div.	**76**	**16**	**6**	**11**	**45**	**20**	**88**	**107**				
	% compliant	**98.4**	**98.5**	**98.1**	**97.3**	**96.1**	**93.4**	**95.4**	**97.4**	n/a	n/a	n/a	n/a
2007-03	# of Conv.	**2307**	**505**	n/a	n/a	n/a	n/a	n/a	**5822**	**6120**	**6164**	**2103**	**2092**
	# of Div.	**21**	**31**						**161**	**44**	**45**	**13**	**13**
	% compliant	**99.1**	**94.2**	n/a	n/a	n/a	n/a	n/a	**97.3**	**99.0**	**99.3**	**99.4**	**99.4**
2007-10	# of Conv.	n/a	**2843**	n/a	n/a	n/a	n/a	n/a	n/a	**3976**	n/a	**2975**	**2828**
	# of Div.		**50**							**39**		**33**	**33**
	% compliant	n/a	**98.3**	n/a	n/a	n/a	n/a	n/a	n/a	**99.1**	n/a	**98.9**	**98.8**

Table 18 Compliance Status: Checking Results Grouped per Product and Evaluation Date

For all evaluation dates, it can be said that the result enabled controversial discussions among architects and developers. Together with the rationales of the stakeholders involved, the compliance checking results served as input to the decision-making process on structural repairs.

6.2.1.3 Summary

The main effect, as confirmed subjectively by the Testo architects and objectively after evaluating 15 products, is that the development organization obtains a great benefit from achieving compliance to a large extent and from avoiding (and counteracting) architectural degeneration. Although we cannot quantify the benefits of architecture compliance checking as a distinct quality engineering instrument, we observed certain factors that indicate its usefulness:

- **Shortened compliance checking cycle:** Fraunhofer IESE started providing compliance checking activities as a service to Testo. The point in time when the compliance checking was conducted was usually late in product development. However, we were able to observe that the cycles between two compliance checking workshops became shorter over time (one evaluation in 2005, two evaluations in 2006, and two evaluations in 2007). At the moment, Testo plans to further reduce the compliance checking cycle time and apply it early in product development, even if only partial implementations are available. This motivates the idea of live compliance checking, which shortens compliance checking to the least minimum possible by providing live feedback.
- **Fewer violations over time:** The results of the compliance checking activities indicate that architectural knowledge in the minds of the developers has been established successfully. For instance, when comparing products P3, P4, P11, and P13 (all are mid-sized systems), a significant and sustainable decline in the number of divergences can be observed. We believe that the number of violations was reduced due to the learning effect, which was achieved by developer education in workshops and meetings.
- **Refinement of the analysis scope:** The initial applications of compliance checking evaluated the framework usage and revealed violations of the layered architecture. The latest analysis results indicate by the low number of violations that these architectural constraints are now in place to a large degree. The architects at Testo plan to refine the analysis scope of compliance checking by evaluating the detailed dependencies on the subsystem and/or component level, too.
- **Cope with evolution:** Compliance checking has been able to cope with the evolution of the Testo architecture and the implemented products. Currently, the third generation of the reuse infrastructure (i.e., the framework) is being developed and, as Table 18 shows, more than a dozen products have been verified. The compliance checking results provide input to the continuous refinement and improvement of the reference architecture and thus, the resulting products.

- **Input to decision-making:** Compliance checking, as mentioned above, provides input to the decision-making process regarding modifications to the architecture and/or source code. Besides, due to the product line context, it is one prerequisite for strategic discussions (e.g., investment into reusable components, test strategy, planning and design for reuse) affecting all members of the product line. In architecture-centric development, it is fundamental that the product implementations adhere to the architecture, since most decisions are based upon the architecture.
- **Raising architectural awareness:** Architecture awareness among developers requires that each developer has knowledge about the architecture, especially in those parts that are related to the current task assignments. The discussion on compliance has fostered the awareness of the developers.

The key lessons learned are that the development organization at Testo evolves its product line with a lot more instances at the same time now than in the past. Ensured compliance is one of the factors that allows managing and maintaining the family of systems at Testo. The importance of compliance was confirmed by the architects. We consider this productivity gain as a partial consequence of having compliance achieved.

6.2.2 Remote Measurement Devices

6.2.2.1 Context

In [Beyer 2008], we report on our experiences in establishing an architecture-centric approach at a small development organization – Wikon GmbH. We applied product line engineering concepts to achieve reuse on a higher level of abstraction than source code. Iteratively, we evolved a development organization towards systematic reuse by introducing an architecture-centric strategy for product development. The Wikon measurement devices – called the XENON8 family – are embedded systems that monitor technical facilities remotely. The development organization for these devices comprises three people, two developers and one person mainly responsible for quality assurance. The systems are implemented in the C programming language.

Crucial for the success of architecture-centric development was the fact that we ensured compliance of the software implementations. We defined a target for the structural decomposition and started the restructuring. In doing so, we monitored the gap between the intended state, which was only 54% compliant with the new structural decomposition, and the work in progress, which incrementally was refactored to establish the new structure. Over a period of roughly two years the implementation was reshaped towards the intended structure.

6.2.2.2 Compliance Status

The first architecting activity we conducted was obviously to define the target architecture of the XENON8 platform. To learn about the variability, we analyzed two variants of the ancestor platform. For the architecture definition, we applied the architecting module of the Fraunhofer PuLSE methodology (Product Line Software and System Engineering, please refer to [Bayer 1999] for details), which resulted in four architectural views, namely conceptual view, structural view, behavioral view, and implementation view. This initial documentation of the intended architecture was partially reconstructed based on the analysis of the ancestor platform. It enabled efficient discussions and reasoning on the abstraction level of the software architecture. Hence, the architectural views were used as a communication vehicle and served as a foundation for making decisions on how to achieve the business and development goals.

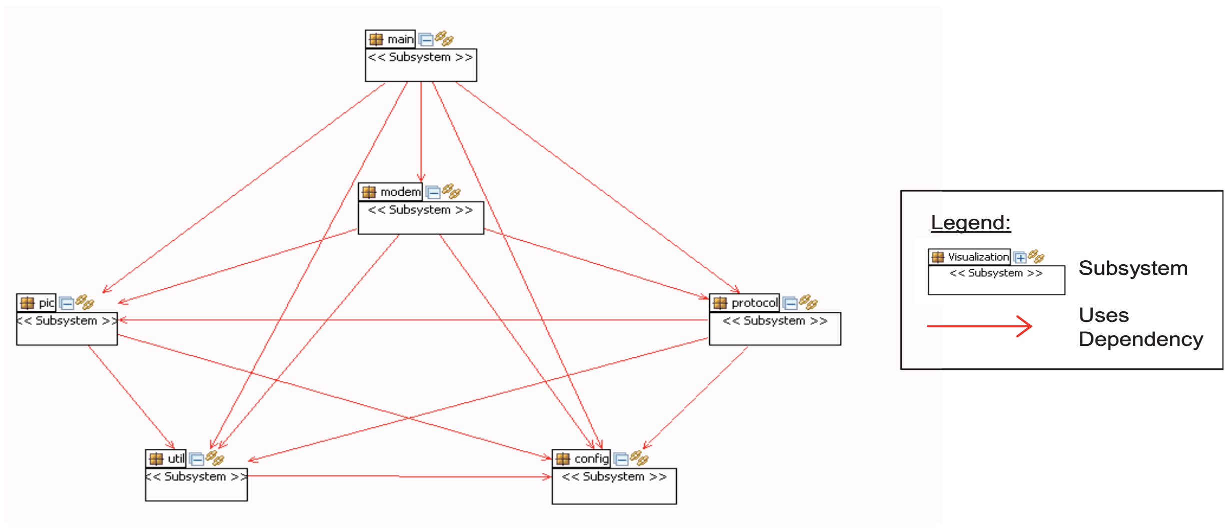


Figure 62 Structural Model: Subsystems and Dependencies

The structural view (see Figure 62) describes the functional decomposition of the system and captures the static structure of a system. It is relatively close to the later implementation because at Wikon, the subsystem structure is reflected by directories in the file system. It is therefore especially interesting for technical stakeholders like developers.

We conducted static architecture compliance checking using Fraunhofer SAVE to regularly measure the distance between the intended target architecture and the implementation in progress. At the beginning, the implementation had a compliance degree of 54% of all the dependencies (i.e., includes, function calls, or variable accesses). During the evolution, the implementations were continuously refactored to match the intended architecture. The compliance status was reviewed regularly and, based on the compliance feedback, we could educate the developers in advance on the consequences of the architectural decisions made. Although architecting and architecture-centric development were new disciplines at Wikon, the developers could achieve a compliance degree of 95% over time. On their own, the Wikon engineer stated the goal of achieving 98% in the near future.

6.2.2.3 Summary

Architecture-centric development can only be successful if the counterparts in the implementation are realized as intended. Then, and only then, can the architecture serve as a vehicle for decision-making on evolution and maintenance and as an instrument to guide the development. At Wikon, we were able to observe the positive impact of architecture-compliant software implementations.

Compared to the previous product generation, the architecture-centric development saved 12 person-months of development time (from 32 to 20 person-months) and 3 person-months for quality assurance (mainly testing, from 8 to 5 person-months) for the first product generation developed following the new strategy. Moreover, the number of variants derived almost doubled during the same period of time, while quality was kept on the same level (there were even slightly less issues due to software-related problems in the field).

The regular feedback on compliance was one important factor to guide the restructuring activities at Wikon into the right (i.e., the intended) direction. The importance of compliance was confirmed by the Wikon engineers. We claim that compliance played a major role in achieving the productivity gain (doubled number of products, one third savings in development effort, with no drop in quality).

6.3 Conclusion

We investigated the role of live feedback in one experiment and compliance in industrial systems in two case studies. In summary, we learned the following lessons:

- **Live compliance checking can reduce the number of structural violations.** As we observed in the experiment with 60% fewer violations, the number was significantly lower for the supported group than for the control group.
- **SAVE LiFe successfully realizes the idea of live compliance checking.** The basic communication and analysis platform performed well in the experiment. Although we could collect a couple of minor suggestions for improving the implementation, we could show that the general principle of live feedback and quasi-constructive reverse

engineering works. Live compliance checking as the first instance provided conveying results.

- **SAVE LiFe scales to development organizations.** We were able to run SAVE LiFe for the experiment with 19 developers for a period of 35 days. Although this experiment took place in an academic setting, we are confident regarding potential roll-outs in industrial settings. Other test applications have already shown the scalability towards larger software-systems and larger distances between developers.
- **It is feasible to achieve high compliance degrees in industrial settings.** We could observe in the two industrial case studies that in the end, they had a compliance degree up to 99% for particular products. This value shows that it is feasible to achieve compliant implementation in industrial practice.
- **Educating developers on the architecture can be achieved.** Giving developers advice on compliance, informing them about violations and their architectural context (origin and target architectural elements, type of violation, etc.) educates the developers over time. We assume that the frequency of feedback influences the time for learning. The faster the feedback is received, the faster the learning is achieved. Therefore, live feedback as realized by SAVE LiFe is appealing because it provides the results with the least delay possible.
- **Compliance as one key enabler for architecture-centric development.** We observed in both industrial studies that the benefits of architecture could be harvested because of compliant implementations. Productivity increased by ensured compliance of implementations with the architecture.

In short, the experiment as well as the two industrial case studies revealed the importance of compliance feedback. Sustaining compliance by pointing out the structural violations in the implementation was successful. The results are indications that compliance feedback educates developers because the compliance could be sustained over a long period of time. Furthermore, education leads to better performance during the development. We therefore tend to accept the two hypotheses H_{F1} and H_{F2} and believe that feedback successfully establishes architecture knowledge in the minds of the developers. We further claim a positive impact of live compliance checking.

7 Analysis and Outlook

This thesis introduced an approach for live compliance checking, which sustains structure in software implementations right from the beginning of the development. In other words, it achieves architecture compliance by construction.

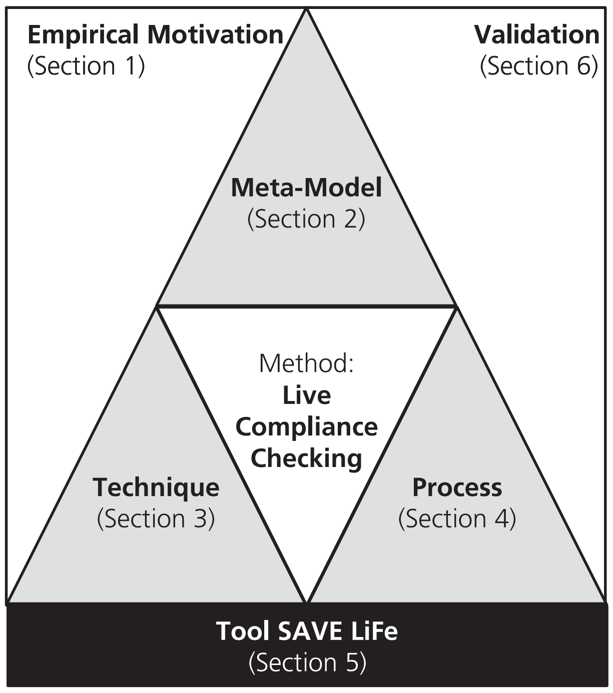


Figure 63 Method Overview: Live Compliance Checking

Figure 63 depicts the core parts of this thesis – the building blocks of the live compliance checking approach:

- Section 2 introduces the underlying *meta-models* for the structural view of the architecture, the source code model, and the mapping between them.
- Section 3 continues by discussing the state of the art of compliance checking techniques and presents base technologies from the area of reverse engineering.
- The process for live compliance checking as part of the development is explained in Section 4. Further, we elaborate on the impact of high-frequency live compliance feedback on developers and derive potential savings for the compliance achievement effort.
- The technical realization of SAVE LiFe (Software Architecture Visualization and Evaluation with Live Feedback) is presented in Section 5. SAVE LiFe is a client-server-client system providing a fat

client (the architecture manager), a central server (the compliance checker), and thin clients (the development monitor).
- The overall empirical motivation for the idea of live compliance checking was already given in Section 1. We motivated the idea by a survey on industrial cases, where the evolution and maintenance of software systems was effort-intensive and time consuming. In all cases, we could further observe that none of the examined systems was compliant with its architecture. We confirmed in three replications of a controlled experiment that lack of compliance is one factor that affects the evolution negatively.
- Having developed the solution and realized tool support via SAVE LiFe and its ancestor SAVE allowed us to investigate the effects of compliance checking with live feedback and feedback at regular intervals. Section 6 presents the results of this validation. Support by SAVE LiFe with live feedback on compliance resulted in 60% fewer violations. Further, we showed the positive effects of compliance in industrial case studies: Increased productivity. Moreover, we could observer that a degree of 99% compliance is feasible in an industrial context.

The results and contributions of this thesis are summarized in Section 7.1 followed by an outlook on future work in Section 7.2. Section 7.3 concludes this thesis with final remarks.

7.1 Results and Contribution

There are three strategies for improving software productivity: working faster, working smarter, or avoiding unnecessary work [Boehm 1999]. Live compliance checking achieves the latter, which promises the highest payoff. Supported by SAVE LiFe, the effort for sustaining structure in software implementations is significantly reduced. Further, developers are educated on the architecture, which enables them to understand the role of their local task in the overall system perspective and to actively participate in architecting.

The pro-active prevention of structural decay is the underlying idea of turning compliance checking from an analytical into a quasi-constructive technique. The results and contributions are in detail:

- We defined a metric for architecture compliance, which can be used to assess the compliance status of software implementations (see Section 2.4).
- The metric takes advantage of the formal definition of the meta-models for the structural view of the architecture and the source code model (see Sections 2.1 to 2.3). Meta-models are required for the execution of analytical compliance checking and live compliance checking.

- We characterized typical evolution scenarios for implemented systems by using the compliance metric (see Section 2.5).
- We exemplified how the reverse engineering archetype is instantiated by compliance checking techniques (see Section 3.3).
- We provided a review of the state of the art in compliance checking (see Section 3.3). We showed the equivalence in expressiveness of Reflexion models and dependency rules (see Section 3.3).
- We showed that Reflexion models provide better applicability for live compliance checking than dependency rules (see Section 3.4).
- We explained the new paradigm of quasi-constructive reverse engineering with live feedback, which generally aims at the constructive use of analytical techniques (see Section 3.5).
- We introduced the process for live compliance checking, which involves the roles of architect and developer (see Section 4.1). Their process parts extend their regular development process through interaction with compliance checking process parts. All three parts together constitute the approach for live compliance checking.
- We explain how the high execution frequency leads to two effects in the implementation of software systems: the learning effect for the developers and the prompt removal effect for violations (see Section 4.2).
- We present a theoretical model on the savings for structural repairs effort (i.e., the compliance achievement effort, see Section 4.3), which predicts for weak effects savings of 67% compared to regular development.
- Finally, Section 5 presents the tool support for live compliance checking. The client-server-client system consists of the architecture manager, the compliance checker on top of the analysis platform, and the development monitor.
- The empirical contributions comprise a survey on industrial systems distilling the problem of lack of compliance (see Sections 1.2.1), three replications of a controlled experiment showing compliance benefits (less than 50% effort for an evolutionary task (see Section 1.2.2), an experiment showing the effect of live compliance checking supported by SAVE LiFe (see Section 6.1), and two industrial case studies reporting on the positive impact of compliance on the overall productivity of a development organization (see Section 6.2). Figure 7 summarizes the empirical contributions (see Section 1.4).

In short, this thesis contributes to the field of software architecture by providing a method that achieves compliance by construction. The positive effects have been empirically validated. In addition to this contribution, this thesis revealed open issues that provide many entry points for future research.

7.2 Future Work

This section delivers a sketch of potential future activities based on the results provided by this thesis. The outlook is grouped into four areas: experimentation (addressing the need for further validation), compliance checking (raising issues in its application), quasi-constructive reverse engineering (the new paradigm introduced by this thesis), and live feedback platform (open issues with the underlying technical platform).

7.2.1 Experimentation

The empirical studies we provided in this thesis require further experimentation and replications, if possible, with varying factors. We can identify four cases that are particularly interesting:

- First, the examined cases in the state-of-the-practice survey (see Sections 1.2.1) might not be representative. We therefore would like to extend the survey and encourage further investigation on this topic by other researchers. Although we have a strong belief that lack of compliance is a recurring, practical problem, more data is required to extend the ground of this assumption.
- Second, the three replications of the compliance experiment (see Section 1.2.2) varied the groups of persons executing the evolutionary task, but this was the sole factor that was different. We challenge other researchers to vary the evolutionary task and the system under analysis. We look forward to receiving more data on this topic.
- Third, the experiment on live compliance checking (see Section 6.2) provided a first data point on the impact of live feedback. However, due to the environmental settings, we plan to replicate this experiment. Ideally, we would have two groups of teams: all implementing the same functionality based on the same architecture over a long period of time (i.e., several weeks or month), one group with SAVE LiFe support and the other one without. Only such a long-term study can deliver well-grounded empirical facts on the long-term applicability and effects of live compliance checking. We are searching for such opportunities but know that this proposed experimentation scenario is rather utopistic.
- Fourth, another open issue is to isolate the role of compliance for the productivity gain as observed in the two case studies reported on in Section 6.2. In both cases, the architects perceived compliance as an important factor; however, we were not able to quantify it. It would be interesting to have data that allows determining the exact impact of compliance.

In short, experimentation in future work requires more data points on the effects of compliance, live feedback, and the role of compliance throughout the entire lifecycle of the software system.

7.2.2 Compliance Checking

We gathered a lot of experiences on applying compliance checking (analytical and quasi-constructive with live feedback) in the course of this thesis. These experiences yield open issues for future research related to the result presentation and the capability to analyze variants:

- The architects of a system have the responsibility to monitor and track the compliance of the implementation. Due to the inherent complexity of modern software systems, adequate means have to be chosen to support the architects and this task. Visualization offers the potential to easily see complex correlations in large data sets, which are not obvious when just looking at the pure data in a textual or tabular form. However, the visualization of compliance checking results for either architects or developers has a strong impact on the perception of the results. For instance, for the fat client of the architecture, we could show a 63% gain in effectiveness for architectural analysis tasks simply by changing the configuration of the graphical elements of the visualization [Knodel 2008c]. Future work should aim at finding an optimal configuration for the architects and investigate different visualization options for the developers, too.
- In compliance checking we are able to analyze exactly one variant (snapshot-based or live during development) at a time. However, many systems today rather exist in families, which means they comprise a number of similar variants or are managed as a product line (see [Weiss 1999] and [Clements 2001] with explicit variation points. Compliance checking so far does not support the analysis of several variants at the same time (i.e., in one run). The extension of compliance checking might lead to effort savings when analyzing large product families.

Compliance checking as such is rather mature, but we identified open issues in the visualization of the results and the missing ability to analyze several variants in one single run.

7.2.3 Quasi-Constructive Reverse Engineering

We introduced the paradigm shift towards quasi-constructive use of reverse engineering techniques (see Section 3.5). An open research question is how this paradigm will influence software development in the long run:

- First, there is a research need to analyze, which reverse engineering techniques are adequate and appropriate candidates to be transformed into a quasi-constructive technique. It is unknown at the moment, if all or a limited set of techniques (and if so, which ones) can produce useful results as a quasi-constructive technique.
- Second, the return on investment (ROI) has to be analyzed for the techniques. Is the effort to integrate the technique into the development process worthwhile? For live compliance checking, we can say that there is clear pay-off, which was confirmed by the theoretical model and the empirical data. But this question is unanswered for other reverse engineering techniques.
- Third, the technical infrastructure and its capabilities to execute other reverse engineering techniques in a quasi-constructive manner has to be reviewed with respect to its performance, resource consumption, and other issues.

To summarize, we think that the idea of quasi-constructive reverse engineering [Knodel 2008a] is beneficial, at least for some techniques like compliance checking. But there is a clear need for future research before we can generalize our conclusion.

7.2.4 Live Feedback Platform

The technical live feedback platform (see Section 5) establishes a logical communication channel between the server and either the thin clients or the fat client. It further provides an extensible platform, which could provide more analysis features:

- We can imagine realizing the live analysis for features like source code metrics, bad smells, anti-patterns, etc. All analyses are based on the meta-model of the source code. Hence, they potentially could be integrated as another kind of analysis provided by SAVE LiFe.
- In addition to the light-weight live feedback for developers, we see as future work the transfer of graphical results so that the developers can use advanced visualization means for perceiving the results.
- Another major benefit would be an extension for the fat client of the architect. Instead of just requesting the compliance status, it might realize a live monitor that visualizes the whole development work of the team live or in a flipbook-based manner. This feature would empower the architect to have full control and total transparency of the development organization currently ongoing work.

The live feedback platform is the first step towards supporting development organizations in managing the source code in an architecture-centric manner. We envision future work to extend this platform towards live visualization of the whole ongoing development with additional analysis features realized on the basis of the platform.

These extensions could culminate in a software evolution environment where everything is visualized live and every modification is immediately processed, analyzed, and tracked. This software evolution environment would enable new development practices and establish quasi-constructive reverse engineering with live feedback as a paradigm, and, last but least, fully integrate reverse engineering and forward engineering. Due to the new approach to quality engineering, rework and unnecessary work could be potentially saved.

7.3 Final Remarks

This thesis provides a classical example of applied research. We observed a practical problem in industry. The assumption on the underlying root cause was empirically validated – we found evidence that compliance has a significant impact on the effort required for evolution. This motivation then guided the development of the solution introduced by this thesis – live compliance checking. To enable this basic idea, we defined the method's meta-models, the core technique, and the surrounding process. Further, we developed the SAVE LiFe tool to support and automate the process. Validating the solution then assured that the solution developed actually tackles the problem observed. Our studies to date confirm that this is the case. Feedback educates developers and eventually leads to a higher degree of compliance.

Our approach achieves architecture compliance by construction. It sustains structure in implementations and assures traceability between architecture and source. However, only the future will allow judging the real value of live compliance checking. The method and the tool are ready to be rolled out for applications in industrial practice. And, of course, we feel confident regarding the future and optimistically look forward to live compliance checking applications.

8 References

[Aldrich 2002]
Aldrich, J., Chambers, C., & Notkin, D. (2002). *ArchJava: connecting software architecture to implementation.* Proceedings of the 22rd International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA.

[Allen 1997]
Allen, R., & Garlan, D. (1997). *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology, 6(3), 213-249.

[Areces 1998]
Areces, C., Felder, M., Hirsch, D., & Yankelevich, D. (1998). *Modal logic as a design notation.* 9th International Workshop on Software Specification and Design (IWSSD-9), Kyoto, Japan.

[Babar 2004]
Babar, M. A., & Gorton, I. (2004). *Comparison of Scenario-Based Software Architecture Evaluation Methods*. Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04) - Volume 00.

[Basili 1993]
Basili, V. (1993). *The Experimental Paradigm in Software Engineering*. Experimental Software Engineering Issues: Critical Assessment and Future Directions, Lecture Notes in Computer Science #706.

[Basili 1994]
Basili, V., Caldiera, G., & Rombach, H. D. (1994). *The Goal/Question/Metric Paradigm*. Encyclopedia of Sofware Engineering (Ed.: John Marciniak), John Wiley & Sons, vol. 1, 528-532.

[Batory 1997]
Batory, D., & Geraci, B. J. (1997). *Composition Validation and Subjectivity in GenVoca Generators*. IEEE Transactions on Software Engineering, 23(2), 67-82.

[Bauhaus 2008]
Bauhaus. (2008). *Bauhaus*. from http://www.axivion.com

[Bayer 1999]
Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., et al. (1999). *PuLSE: A Methodology to Develop Software Product Lines.* Proceedings of the Fifth Symposium on Software Reusability (SSR 1999), Los Angeles, USA.

[Bayer 2004]
Bayer, J., Forster, T., Ganesan, D., Girard, J.-F., John, I., Knodel, J., et al. (2004). *Definition of Reference Architectures based on Existing Systems*. Fraunhofer IESE, Kaiserslautern, Germany. IESE-Report 034.04/E.

[Beyer 2008]
Beyer, H. J., Hein, D., Schitter, C., Knodel, J., Muthig, D., & Naab, M. (2008). *Introducing Architecture-Centric Reuse into a Small Organization.* 10th International Conference on Software Reuse (ICSR 2008).

[Boehm 1981]
Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ : Prentice-Hall.

[Boehm 1995]
Boehm, B. W. (1995). *Engineering Context for Software Architecture.* First International Workshop on Architecture for Software Systems, Seattle, Washington, USA.

[Boehm 1999]
Boehm, B. W. (1999). *Managing software productivity and reuse*. IEEE Computer, 32(9), 111–113.

[Bosch 2000]
Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co.

[Bourquin 2007]
Bourquin, F., & Keller, R. K. (2007). *High-impact Refactoring Based on Architecture Violations.* 11th European Conference on Software Maintenance and Reengineering, 2007. CSMR '07.

[Broehl 1995]
Broehl, A.-P., & Droeschel, W. (1995). *Das V-Modell. Der Standard fuer die Softwareentwicklung mit Praxisleitfaden* Oldenbourg Verlag, (2nd edition).

[Carmichael 1995]
Carmichael, I., Tzerpos, V., & Holt, R. C. (1995). *Design maintenance: unexpected architectural interactions.* Proceedings of the International Conference on Software Maintenance (ICSM 1995), Opio (Nice), France.

[Chikofsky 1990]
Chikofsky, E. J., & Cross II, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy (Vol. 7, pp. 13-17): IEEE Computer Society Press.

[Christl 2005]
Christl, A., Koschke, R., & Storey, M.-A. D. (2005). *Equipping the Reflexion Method with Automated Clustering.* Working Conference on Reverse Engineering (WCRE), Pittsburgh, USA.

[Clements 2001]
Clements, P., & Northrop, L. M. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[Clements 2002a]
Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., et al. (2002a). *Documenting Software Architectures: Views and Beyond*. Pearson Education.

[Clements 2002b]
Clements, P., Kazman, R., & Klein, M. (2002b). *Evaluating software architectures: methods and case studies*. Addison-Wesley Longman Publishing Co., Inc.

[Clements 2003]
Clements, P., & Kazman, R. (2003). *Software Architecture in Practices*. Addison-Wesley Longman Publishing Co., Inc.

[Dashofy 2002]
Dashofy, E. M., van der Hoek, A., & Taylor, R. N. (2002). *An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages.* Proceedings of the 24th International Conference on Software Engineering (ICSE 2002).

[Davis 1997]
Davis, M. J., & Williams, R. B. (1997). *Software architecture characterization*. ACM SIGSOFT Software Engineering Notes, 22(3), 30-38.

[Deming 1986]
Deming, W. E. (1986). *Out of the Crisis*. MIT CAES Center for Advanced Engineering Study. Cambridge, MA, USA.

[Dennis 2003]
Dennis, G. (2003). *TSAFE: Building a Trusted Computing Base for Air Traffic Control Software.* Masters Thesis, Massachusetts Institute of Technology (MIT), USA.

[Deursen 2004]
Deursen, A. v., Hofmeister, C., Koschke, R., Moonen, L., & Riva, C. (2004). *Symphony: View-Driven Software Architecture Reconstruction*. Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04) - Volume 00.

[Dijkstra 1982]
Dijkstra, E. W. (1982). *Selected writings on computing: a personal perspective*. Springer-Verlag New York, Inc.

[Dobrica 2002]
Dobrica, L., & Niemelä, E. (2002). *A survey on software architecture analysis methods*. IEEE Transactions on Software Engineering, 28 638-653.

[Dueñas 2005]
Dueñas, J. C., & Capilla, R. (2005). *The Decision View of Software Architecture*. 2nd European Workshop on Software Architecture (EWSA 2005). Pisa, Italy.

[Ebert 2002]
Ebert, J., Kullbach, B., Riediger, V., & Winter, A. (2002). *Gupro - generic understanding of programs*. Electronic Notes in Theoretical Computer Science, 72(2), 59–68.

[Eclipse 2009]
Eclipse. (2009). *Eclipse Platform*. from http://www.eclipse.org

[Eichberg 2008]
Eichberg, M., Kloppenburg, S., Klose, K., & Mezini, M. (2008). *Defining and continuous checking of structural program dependencies.* Proceedings of 30th International Conference on SoftwareEngineering (ICSE 2008), Leipzig, Germany.

[Eick 2001]
Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). *Does Code Decay? Assessing the Evidence from Change Management Data*.

[EMF 2009]
EMF. (2009). *Eclipse Modeling Framework (EMF)*. from http://eclipse.org/modeling/emf/

[Endres 2003]
Endres, A., & Rombach, H. D. (2003). *A Handbook of Software and Systems Engineering - Empirical Observations, Laws and Theories*. Addison-Wesley.

[Erzberger 2001]
Erzberger, H. (2001). *The automated airspace concept.* 4th USA/Europe Air Traffic Management R&D Seminar, Santa Fe, New Mexico, USA.

[Feijs 1998]
Feijs, L., Krikhaar, R., & Van Ommering, R. (1998). *A Relational Approach to Support Software Architecture Analysis*. Software Practice and Experience, 28(4), 371-400.

[Fischer 2003]
Fischer, M., Pinzger, M., & Gall, H. (2003). *Analyzing and Relating Bug Report Data for Feature Tracking.* Proceedings of the 10th Working Conference on Reverse Engineering, Victoria, Canada.

[Fjelstad 1983]
Fjelstad, R. K., & Hamlen, W. T. (1983). *Application program maintenance study: report to our respondents*. G. Parikh and N. Zvegintzov, eds. Tutorial on Software Maintenance. Los Angeles, CA: IEEE Computer Society Press, 11–27.

[Frakes 1992]
Frakes, W. B., & Baeza-Yates, R. (1992). *Information Retrieval, Data Structures and Algorithms*. Prentice Hall.

[Frenzel 2007]
Frenzel, P., Koschke, R., Breu, A., & Angstmann, K. (2007). *Extending the Reflexion Method for Consolidating Software Variants into Product Lines.* 14th Conference on Reverse Engineering (WCRE 2007), Vancouver, Canada.

[Garlan 1995]
Garlan, D. A., & Ockerbloom, J. R. (1995). *Architectural mismatch: Why reuse is so hard*. IEEE Software, 12(6), 17–26.

[GEF 2009]
GEF. (2009). *Graphical Editing Framework (GEF)*. from http://eclipse.org/gef/

[GMF 2009]
GMF. (2009). *Graphical Modeling Framework (GMF)*. from http://eclipse.org/modeling/gmf/

[Godfrey 2000]
Godfrey, M. W., & Lee, E. H. S. (2000). *Secrets from the monster: Extracting Mozilla's software architecture*. In Proc. of 2000 Intl. Symposium on Constructing software engineering tools (CoSET 2000).

[Graphviz 2008]
Graphviz. (2008). *Graphviz - Graph Visualization Software*. from http://www.graphviz.org/

[Green 1999]
Green, G., & Hevner, A. (1999). *Perceived Control of Software Developers and Its Impact on the Successful Diffusion of Information Technology* Special Report CMU/SEI-98-SR-013, Software Engineering Institute, Carnegie Mellon University, USA.

[Green 2000]
Green, G., & Hevner, A. (2000). *Guidance for the Successful Diffusion of Information Technology Innovations in Software Development Organizations*. IEEE Software, 17(6), 96-103.

[Gurp 2002]
Gurp, J. v., & Bosch, J. (2002). *Design erosion: problems and causes*.

[Harris 1995]
Harris, D. R., Reubenstein, H. B., & Yeh, A. S. (1995). *Reverse engineering to the architectural level.* Proceedings of the 17th International Conference on Software engineering (ICSE 1995), Seattle, USA.

[Henry 1981]
Henry, S., & Kafura, D. (1981). *Software Structure Metrics Based on Inforamtion Flow*. IEEE Transactions on Software Engineering, SE-7(5).

[Herzum 2000]
Herzum, P., & Sims, O. (2000). *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, Inc.

[Hochstein 2005]
Hochstein, L., & Lindvall, M. (2005). *Combating architectural degeneration: a survey*. Information and Software Technology, In Press, Corrected Proof.

[Hofmeister 2000]
Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc.

[Holt 1996]

Holt, R. C. (1996). *Binary relational algebra applied to software architecture*. CSRI Technical Report 345, University of Toronto, Canada.

[Holt 1998]

Holt, R. C. (1998). *Structural manipulations of software architecture using Tarski relation algebra* Proceedings of 5th Working Conference on Reverse Engineering (WCRE 1998), Honolulu, Hawaii, USA.

[Hou 2006]

Hou, D., & Hoover, H. J. (2006). *Using SCL to specify and check design intent in source code*. IEEE Transactions on Software Engineering, 32(6), 404-423.

[IEEE-Std-610.12 1990]

IEEE-Std-610.12. (1990). *IEEE standard glossary of software engineering terminology*. IEEE, New York, 1990.

[IEEE-Std.1471 2000]

IEEE-Std.1471. (2000). *ANSI/IEEE Std 1471-2000 - Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE, New York, October 2000.

[Ionita 2002]

Ionita, M. T., Obbink, H., & Hammer, D. (2002). *Scenario-Based Architecture Evaluation Methods: An Overview*. Software Architecture Review and Assessment Workshop Proceedings (SARA), at International Conference on Software Engineering (ICSE'02).

[Jazayeri 2000]

Jazayeri, M., Ran, A., & Linden, F. v. d. (2000). *Software architecture for product families: principles and practice*. Addison-Wesley Longman Publishing Co., Inc.

[jDepend 2008]

jDepend. (2008). *jDepend*. from http://www.clarkware.com/software/JDepend.html

[John 2003]

John, I., & Dörr, J. (2003). *Elicitation of Requirements from User Documentation.* Ninth International Workshop on Requirements Engineering: Foundation for Software Quality (Refsq '03), Klagenfurt/Velden, Austria.

[jRMTool 2008]

jRMTool. (2008). *jRMTool*. from http://www.cs.ubc.ca/~murphy/jRMTool

[Klocwork 2008]
Klocwork. (2008). from http://www.klocwork.com/

[Knodel 2002]
Knodel, J. (2002). *Process models for the reconstruction of software architecture views.* Diploma Thesis no. 1987, University of Stuttgart, Germany.

[Knodel 2003]
Knodel, J., & Pinzger, M. (2003). Improving Fact Extraction of Framework-Based Software Systems. In *10th Working Conference on Reverse Engineering. WCRE'2003 - Proceedings* (pp. 186-195): IEEE Computer Society, Los Alamitos.

[Knodel 2004]
Knodel, J., & Girard, J.-F. (2004). *Request-driven Reverse Engineering for Product Lines.* Reengineering Prozesse (RePro 2004) Workshop . Fallstudien, Methoden, Vorgehen, Werkzeuge, Koblenz, Germany.

[Knodel 2005a]
Knodel, J., John, I., Ganesan, D., Pinzger, M., Usero, F., Arciniegas, J. L., et al. (2005a). *Asset Recovery and Their Incorporation into Product Lines.* 12th Working Conference on Reverse Engineering (WCRE 2005), Pittsburgh, USA.

[Knodel 2005b]
Knodel, J., Lindvall, M., & Muthig, D. (2005b). *Static Evaluation of Software Architectures - A Short Summary.* Fifth Working IEEE / IFIP Conference on Software Architecture (WICSA 2005), Pittsburgh, USA.

[Knodel 2005c]
Knodel, J., & Muthig, D. (2005c). *Analyzing the Product Line Adequacy of Existing Components.* First International Workshop on Reengineering towards Product Lines (R2PL 2005), Pittsburgh, USA.

[Knodel 2006a]
Knodel, J., Kolb, R., Muthig, D., Leszak, M., Rauch, P., Meier, G., et al. (2006a). *Software Architecture Innovation Cycle - Development, Documentation, and Compliance Checking*. Fraunhofer IESE, Kaiserslautern, Germany. IESE-Report 178.06/E.

[Knodel 2006b]
Knodel, J., Koschke, R., & Mende, T. (2006b). *Reverse Engineering in a Reuse Context*. Technical Report. Fraunhofer IESE, Kaiserslautern, Germany. IESE-Report 177.06/E.

[Knodel 2006c]
Knodel, J., Muthig, D., Naab, M., & Lindvall, M. (2006c). *Static Evaluation of Software Architectures*. Proceedings of the Conference on Software Maintenance and Reengineering.

[Knodel 2006d]
Knodel, J., Muthig, D., Naab, M., & Zeckzer, D. (2006d). *Towards Empirically Validated Software Architecture Visualization.* ACM Symposium on Software Visualization, SOFTVIS 06, Brighton, United Kingdom.

[Knodel 2007]
Knodel, J., & Popescu, D. (2007). *A Comparison of Static Architecture Compliance Checking Approaches.* Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), Mumbai, India.

[Knodel 2008a]
Knodel, J., & Muthig, D. (2008a). *A Decade of Reverse Engineering at Fraunhofer IESE - The Changing Role of Reverse Engineering in Applied Research.* 10th Workshop Software Reengineering (WSR 2008), Bad Honnef, Germany.

[Knodel 2008b]
Knodel, J., Muthig, D., Haury, U., & Meier, G. (2008b). *Architecture Compliance Checking - Experiences from Successful Technology Transfer to Industry.* 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), Athens, Greece.

[Knodel 2008c]
Knodel, J., Muthig, D., & Naab, M. (2008c). *An Experiment on the Role of Graphical Elements in Architecture Visualization*. Empirical Software Engineering Journal (EMSE) 13 (6), 693-726.

[Knodel 2008d]
Knodel, J., D. Muthig, & Rost, D. (2008d). Constructive Architecture Compliance Checking — An Experiment on Support by Live Feedback. International Conference on Software Maintenance (ICSM), Beijing, China.

[Knodel 2009a]
Knodel, J., Duszynski, S., & Lindvall, M. (2009a). *SAVE: Software Architecture Visualization and Evaluation.* 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), Kaiserslautern, Germany.

[Knodel 2009b]
Knodel, J., & Lindvall, M. (2009b). *TSAFE Architecture Analyses - Evaluation of the Quality Attribute Compliance*. Technical Report. IESE-Report 051.09/E, Kaiserslautern, Germany.

[Kolb 2006]
Kolb, R., John, I., Knodel, J., Muthig, D., Haury, U., & Meier, G. (2006). Experiences with Product Line Development of Embedded Systems at Testo AG. In *10th International Software Product Line Conference, SPLC 2006 - Proceedings* (pp. 172-181): IEEE Computer Society, Los Alamitos.

[Koschke 2003]
Koschke, R., & Simon, D. (2003). *Hierarchical Reflexion Models.* 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria, Canada.

[Koschke 2005]
Koschke, R. (2005). *Rekonstruktion von Software-Architekturen*. Informatik - Forschung und Entwicklung, 19(3), 127-140.

[Koschke 2008]
Koschke, R. (2008). *Zehn Jahre WSR - Zwölf Jahre Bauhaus.* Proceedings of 10th Workshop Software Reengineering (WSR 2008), Bad Honnef, Germany.

[Krikhaar 1999]
Krikhaar, R. (1999). *Software Architecture Reconstruction.* Ph.D. Thesis, University of Amsterdam, The Netherlands.

[Kruchten 1995]
Kruchten, P. (1995). *The 4+1 View Model of Architecture*. IEEE Software, 12(6), 42-50.

[Lam 2003]
Lam, P., & Rinard, M. (2003). *A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information.* Proceedings of the 17th European Conference on Object-Oriented Programming.

[Lattix 2008]
Lattix. (2008). *Lattix Dependency Manager (LDM).* from http://www.lattix.com

[Lehman 1985]
Lehman, M. M., & Belady, L. A. (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc.

[Lindvall 2002]
Lindvall, M., Tvedt, R. T., & Costa, P. (2002). *Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture.* 8th IEEE International Software Metrics Symposium (METRICS 2002), Ottawa, Canada.

[Lindvall 2003]
Lindvall, M., Tvedt, R. T., & Costa, P. (2003). *An Empirically-Based Process for Software Architecture Evaluation*. Empirical Software Engineering, 8(1), 83-108.

[Lindvall 2005]
Lindvall, M., Rus, I., Shull, F., Zelkowitz, M. V., Donzelli, P., Memon, A., et al. (2005). *An Evolutionary Testbed for Software Technology Evaluation*. Innovations in Systems and Software Engineering - a NASA Journal,, 1(1), 3-11.

[McCabe 1976]
McCabe, T. J. (1976). *A complexity measure*. IEEE Transaction on Software Engineering, 2(4).

[Medvidovic 1999]
Medvidovic, N., Rosenblum, D. S., & Taylor, R. N. (1999). *A Language and Environment for Architecture-Based Software Development and Evolution.* Proceedings of the International Conference on Software Engineering (ICSE 1999).

[Medvidovic 2000]
Medvidovic, N., & Taylor, R. N. (2000). *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, 26(1), 70-93.

[Mendoca 1996]
Mendoca, N. C., & Kramer, J. (1996). *Requirements for an Effective Architecture Recovery Framework.* SIGSOFT 96 Workshop, San Francisco, USA.

[Merriam-Webster 2009]
Merriam-Webster. (2009). *Compliance -- Merriam-Webster Dictionary*. from http://www.merriam-webster.com/dictionary/compliance

[Miller 2002]
Miller, F. J. (2002). *I = 0 (Information has no intrinsic meaning)*. Electronic Journal on Information Research, 8(1), paper no. 140.

[Miodonski 2004]
Miodonski, P. (2004). *Evaluation of Software Architectures with Eclipse.* Rombach, H. Dieter (Supervisor); Muthig, Dirk (Supervisor); Lindvall, Mikael (Supervisor); Knodel, Jens (Supervisor); Forster, Thomas (Supervisor). Diploma Thesis, TU Kaiserslautern, Germany.

[Moonen 2001]
Moonen, L. (2001). *Generating robust parsers using island grammars.* Working Conference on Reverse Engineering (WCRE), Stuttgart, Germany.

[Moonen 2002]
Moonen, L. (2002). *Exploring Software Systems.* PhD thesis, Faculty of Natural, Sciences, Mathematics, and Computer Science, University of Amsterdam.

[Müller 1994]
Müller, H. A., Wong, K., & Tilley, S. R. (1994). *Understanding software systems using reverse engineering technology.* The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS).

[Murphy 1995]
Murphy, G. C., Notkin, D., & Sullivan, K. J. (1995). *Software Reflexion Models: Bridging the Gap between Source and High-Level Models.* Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering Washington D.C., USA

[Murphy 1996]
Murphy, G. C., & Notkin, D. (1996). *Lightweight lexical source model extraction*. ACM Transactions on Software Engineering and Methodology, 5(3), 262-292.

[Murphy 1997]
Murphy, G. C., & Notkin, D. (1997). *Reengineering with Reflexion Models: A Case Study*. IEEE Computer, 30(8), 29-36.

[Murphy 2001]
Murphy, G. C., Notkin, D., & Sullivan, K. J. (2001). *Software reflexion models: bridging the gap between design and implementation*. IEEE Transactions on Software Engineering, 27(4), 364-380.

[Naur 1968]
Naur, P., & Randell, B. (1968, 7-11 Oct). *SOFTWARE ENGINEERING: Report of a conference sponsored by the NATO Science Committee*, Garmisch, Germany.

[OSGi 2009]
OSGi. (2009). *OSGi - Open Services Gateway initiative*. from http://www.osgi.org/

[Parnas 1972]
Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules (Vol. 15, pp. 1053-1058): ACM.

[Parnas 1994]
Parnas, D. L. (1994). *Software aging*. Proceedings of the 16th international conference on Software engineering.

[Perry 1992]
Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture (Vol. 17, pp. 40-52): ACM.

[Pinzger 2002]
Pinzger, M., Fischer, M., Gall, H., & Jazayeri, M. (2002). *Revealer: A Lexical Pattern Matcher for Architecture Recovery.* Working Conference on Reverse Engineering, Richmond, USA.

[Pinzger 2003]
Pinzger, M., Gall, H., Girard, J.-F., Knodel, J., Riva, C., Pasman, W., et al. (2003). *Architecture Recovery for Product Families.* Fifth International Workshop on Product Family Engineering (PFE-5), Siena, Italy.

[Pollet 2007]
Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cimpan, S., & Verjus, H. (2007). *Towards A Process-Oriented Software Architecture Reconstruction Taxonomy.* 11th European Conference on Software Maintenance and Reengineering (CSMR'07)

[Postma 2003]
Postma, A. (2003). *A method for module architecture verification and its application on a large component-based system*. Information & Software Technology, 45(4), 171-194.

[Pressman 2004]
Pressman, R. S. (2004). *Software Engineering: A Practitioner's Approach*. McGraw Hill, New York.

[Probst 1999]
Probst, G. J. B., Raub, S., & Romhardt, K. (1999). *Wissen managen: Wie Unternehmen ihre wertvollste Ressource optimal nutzen*. Wiesbaden. Dr. Th. Gabler Verlag.

[Raza 2006]

Raza, A., Vogel, G., & Plödereder, E. (2006). *Bauhaus - a tool suite for program analysis and reverse engineering.* 11th International Conference on Reliable Software Technologies (Ada-Europe).

[Riva 2004]

Riva, C. (2004). *View-based Software Architecture Reconstruction.* Ph.D. Thesis, Vienna University of Technology, Austria.

[RMI 2009]

RMI. (2009). *Java Remote Method Invocation (Java RMI)*. from http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[Rosik 2008]

Rosik, J., Gear, A. L., Buckley, J., & Babar, M. A. (2008). *An industrial case study of architecture conformance*. Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement.

[Rost 2007]

Rost, D. (2007). *Real-Time Tracking of Evolving Software Architectures.* Knauber, Peter (Supervisor). Knodel, Jens (Supervisor). Diploma Thesis, Hochschule Mannheim, Germany.

[Rozanski 2005]

Rozanski, N., & Woods, E. (2005). *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional.

[Rus 2002]

Rus, I., & Lindvall, M. (2002). Guest Editors' Introduction: Knowledge Management in Software Engineering (Vol. 19, pp. 26-38): IEEE Computer Society Press.

[Sartipi 2006]

Sartipi, K., Dezhkam, N., & Safyallah, H. (2006). *An Orchestrated Multi-view Software Architecture Reconstruction Environment.* Proceedings of the IEEE International Working Conference on Reverse Engineering (WCRE 2006), Benevento, Italy.

[Schmid 2005]

Schmid, K., John, I., Kolb, R., & Meier, G. (2005). *Introducing the PuLSE Approach to an Embedded System Population at Testo AG*. Proceedings of the International Conference on Software Engineering (ICSE’05).

[Semmle 2008]
Semmle. (2008). *Semmle .QL - Source Code Query Language*. from http://www.semmle.com

[Shaw 1996]
Shaw, M., & Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc.

[Sommerville 2001]
Sommerville, I. (2001). *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc.

[SonarJ 2008]
SonarJ. (2008). *SonarJ*. from http://www.hello2morrow.com/

[Sotograph 2008]
Sotograph. (2008). *Software Tomography*. from http://www.software-tomography.com/

[SPEM 2008]
SPEM. (2008). *Software Process Engineering Metamodel (SPEM)*. from http://www.omg.org/technology/documents/formal/spem.htm

[Structure101 2008]
Structure101. (2008). *Structure 101*.

[Sveiby 1997]
Sveiby, K.-E. (1997). *The New Organizational Wealth: Managing and Measuring Knowledge-Based Assets*. San Francisco, CA, USA. Berrett-Koehler Publishers.

[Tu 2001]
Tu, Q., & Godfrey, M. W. (2001). *The Build-Time Software Architecture View*. Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01).

[Tyree 2005]
Tyree, J., & Akerman, A. (2005). Architecture Decisions: Demystifying Architecture (Vol. 22, pp. 19-27): IEEE Computer Society Press.

[UML 2008]
UML. (2008). *Unified Modeling Language*. from http://www.omg.org/technology/documents/formal/spem.htm

[van Ommering 2000]
van Ommering, R., van der Linden, F., Kramer, J., & Magee, J. (2000). *The Koala Component Model for Consumer Electronics Software*. IEEE Computer, 33(3), 78-85.

[Wallnau 1996]
Wallnau, K., Clements, P., Morris, E., & Krut, R. (1996). *The Gadfly: An Approach to Architectural-Level System Comprehension*. Proceedings of the 4th International Workshop on Program Comprehension (IWPC '96).

[Waters 1999]
Waters, R., & Abowd, G. D. (1999). *Architectural Synthesis: Integrating Multiple Architectural Perspectives.* Sixth Working Conference on Reverse Engineering (WCRE 1999) Atlanta, Georgia, USA.

[Weggeman 1999]
Weggeman, M. (1999). *Wissensmanagement - Der richtige Umgang mit der wichtigsten Unternehmens-Ressource*. Bonn. mitp-Verlag.

[Weiss 1999]
Weiss, D. M., & Lai, C. T. R. (1999). *Software Product-Line Engineering. A Family-Based Software Development Process*. Addison-Wesley.

[Wikipedia 2008]
Wikipedia. (2008). *Knowledge --- Wikipedia, The Free Encyclopedia*. from http://en.wikipedia.org/w/index.php?title=Knowledge&oldid=186235923

[Wohlin 2000]
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslen, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers.

[Wright 1936]
Wright, T. P. (1936). *Learning Curve*. Journal of the Aeronautical Sciences.

[Zachman 1987]
Zachman, J. A. (1987). *A framework for information systems architecture*. IBM Systems Journal, 26(3), 276-292.

[Zimmermann 2005]
Zimmermann, T., Weissgerber, P., Diehl , S., & Zeller, A. (2005). *Mining Version Histories to Guide Software Changes*. IEEE Transactions on Software Engineering, 31(6), 429-445.

9 Appendix

Appendix A Architectural Views and Compliance Checking

A.1 Architectural Views and Viewpoint

The documentation of software architecture typically comprises a set of architectural views that have an explicit meta-model – the architectural viewpoint.

The architecture community has adopted the idea of view-based architecture documentation. To apply it in practice, the set of views to be used must be selected so that it is neither too large (resulting in overhead) nor too small (missing important information). The selection of relevant views is a non-trivial task because of the variety of available views and stakeholder concerns to be addressed. Hence, it is common practice to constitute the architecture documentation on top of a standard set of architectural views and to tailor it towards system-project-, or organization-specific needs and, if necessary, extend the set with customized views.

A number of architectural view sets has been proposed by different researchers. The most commonly used ones are presented in chronological order:

- **Kruchten:** As one of the most important contributions to view-based documentation of software architectures, [Kruchten 1995] proposed a system of four interrelated views (logical, process, development, and physical view) augmented with a fifth redundant view (scenarios) that abstracts from requirements and shows how the architectural views work together to satisfy the requirements.
- **Davis:** [Davis 1997] proposed a set of four views (domain, component, platform, and interface view) augmented with a fifth view, the context view, to capture the dynamic behavior and quality characteristics of the software system.
- **Hofmeister:** Based on the analysis of the software architectures of large industrial systems, [Hofmeister 2000] proposed a set of four distinct views (conceptual, module, code, and execution view) each describing particular aspects of the system.
- **Herzum:** [Herzum 2000] introduced another set of four views (technical, application, project management, and functional view).
- **Clements:** [Clements 2002a] describes a set of three so-called view types and extends it by the description of commonly occurring forms

and variations called styles (module view type: decomposition, uses, generalization, and layer style; component-and-connector view type: pipe-and-filter, shared-data, publish-subscribe, client-server, peer-to-peer, and communicating-process style; and allocation view type: deployment, implementation, and work assignment style).

- **Rozanksi:** [Rozanski 2005] presents a catalog of six core views (functional, information, concurrency, development, deployment, and operational view) and applies architectural perspectives for considerations crosscutting the architectural views.
- **Fraunhofer IESE:** The architectural view set defined at Fraunhofer IESE comprises a standard set of four architectural views (conceptual, structural, behavioral, and implementation view), which is optionally extended with four additional views (data, hardware, execution, and organizational view). [Bayer 2004] and [Knodel 2006a] present a detailed description of the Fraunhofer IESE view set, which is used throughout this thesis unless otherwise mentioned.

Table 19 present a comparison of the different view sets as presented above. The views in Table 19 differ in terms of names and definition, and although the concerns they address are sometimes not clearly separated, the views can roughly be mapped onto each other. In the event there is no direct correspondence, Table 19 depicts a "–" in the respective table cell. Sometimes two views correspond to one of the other view sets.

Table 19 is not complete since other view models have been proposed, and single views have been motivated in literature, for instance, the built-time view [Tu 2001] or the decision view [Dueñas 2005]. However, one observation can be made in Table 19: All of the view sets compared propose the structural and the implementation view, and, with one exception, the behavioral view (or something named alike).

- Structural view: The structural view describes the functional decomposition of the system and captures the static structure of a system in terms of layers, subsystems, and components, the interfaces provided by them, and the relationships between the various elements.
- Implementation view: The implementation view describes how the software implementing a system is organized in the development environment. It captures how architectural elements defined in the structural view are organized in the development, integration, or configuration management environments.
- Behavioral view: The behavioral view illustrates how the architectural elements defined in the structural view interact with each other for a number of typical usage scenarios. The behavioral view shows which elements of the architecture interact, which operations are invoked by an element, and which messages and events are passed between elements.

Kruchten	Davis	Hof-meister	Herzum	Clements	Rozanski	Fraun-hofer IESE
Use Cases	Context View	–	–	Scenarios	Scenarios Perspectives	Scenarios
Logical View	Domain View	Conceptual View	Application	–	–	Conceptual View
Development View	Interface View	Module View	Technical	Module	Functional	Structural View
Process View	Context View	Execution View	–	Component and Connector	Concurrency	Behavioral View
Development View	Component and Interface View	Code View	Technical	Allocation (Implementation)	Development	Implementation View
–	–	–	–	–	Information	Data View
Physical View	Platform View	Execution View	–	Allocation (Deployment)	Deployment	Hardware View
Process and Physical View	Platform View	Execution View	–	Allocation (Deployment)	Operational	Execution View
–	–	–	Project Management View	Allocation (Work Assignment)	–	Organizational View

Table 19 Comparison of Architectural View Sets

A.2 Architecture Compliance Checking and Violations

Table 20 gives an overview of architecture compliance checking by relating the input view and the software system counterpart to each other. Hence, it illustrates which architectural views can be compared against which system artifacts. Please note that only the most popular architectural views (structural, behavioral, and implementation view; see Table 19) have been listed in Table 20 and that the conceptual view has not been included in the table since it is the most abstract view providing only a brief overview of a system.

Table 20 further classifies the types of violations that can be detected and presents the application phase (covering the categories design, implementation, integration, and execution) in which the compliance checking activity is typically executed.

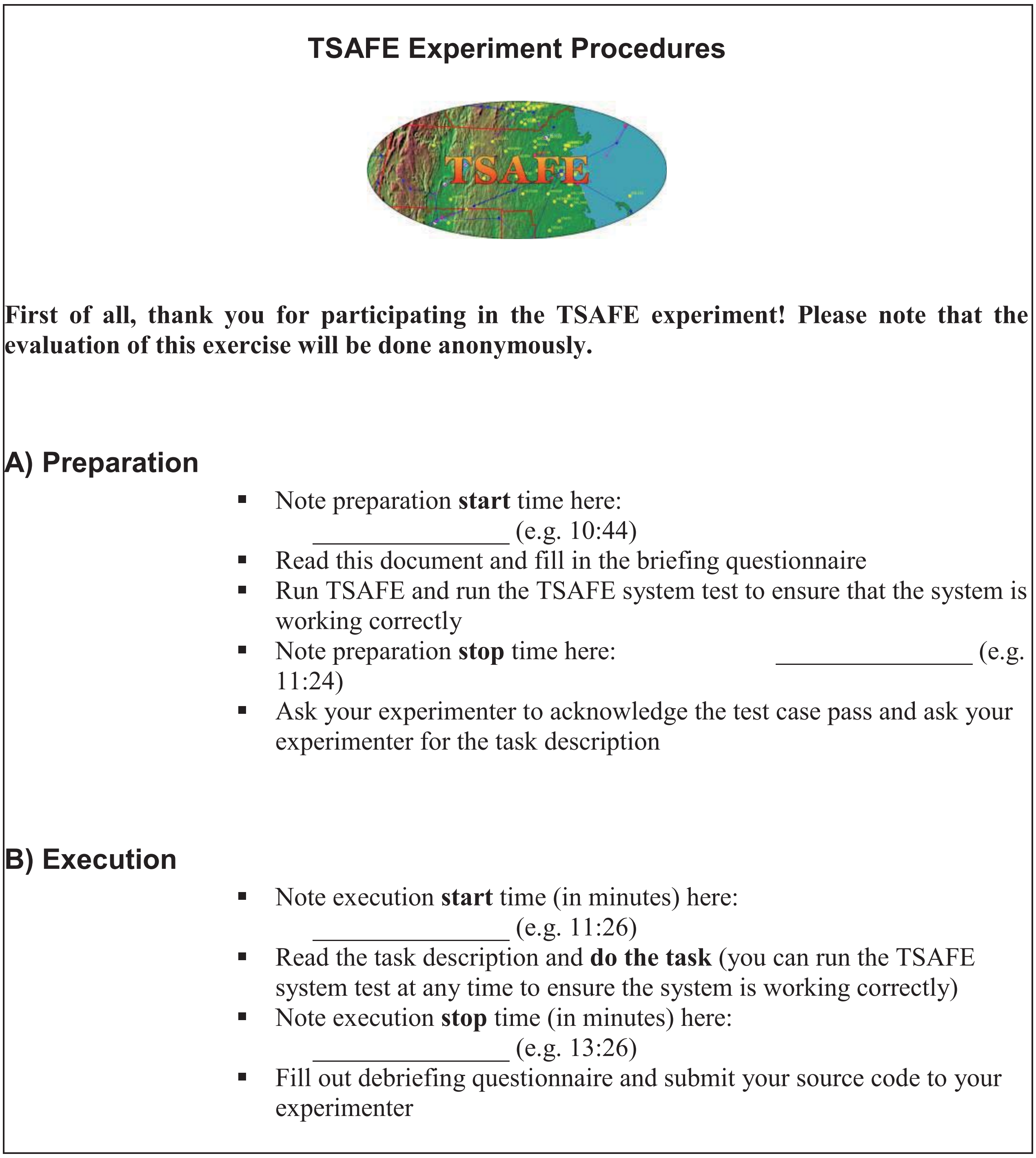
Input View	System Artifact	Violations	Application Phase
Structural View	Component design models	**Design violations**: unspecified architectural elements and unspecified **static** inter-element relationships	Design
	Source code	**Structural violations**: unspecified architectural elements and unspecified **static** inter-element relationships implemented in the source code	Integration
	Run-time traces	**Structural violations**: unspecified architectural elements and unspecified **dynamic** inter-element relationships implemented in the source code	Execution
Behavioral View	Component design models	**Design violations**: unspecified architectural elements and unspecified **dynamic** inter-element relationships	Integration
	Run-time traces	**Protocol violations**: unspecified **dynamic** inter-element relationships	Execution
Implementation View	Source code	**Decomposition violations**: unspecified decomposition of architectural elements in the file system	Integration
	Logging of configuration management transactions	**Ownership violations**: unspecified code ownership (unauthorized access or modifications) of architectural elements	Integration
	Regression test suite	**Test violations**: unspecified omission of architectural elements in regression test	Execution

Table 20 Overview of Architecture Compliance Checking

Appendix B Experiment Compliance

This section presents the material used for the experiment on the impact of compliance as described in Section 1.2.2.

B.1 Experiment Procedures



TSAFE Experiment Procedures

TSAFE

First of all, thank you for participating in the TSAFE experiment! Please note that the evaluation of this exercise will be done anonymously.

A) Preparation

- Note preparation **start** time here:
  _______________ (e.g. 10:44)
- Read this document and fill in the briefing questionnaire
- Run TSAFE and run the TSAFE system test to ensure that the system is working correctly
- Note preparation **stop** time here: _______________ (e.g. 11:24)
- Ask your experimenter to acknowledge the test case pass and ask your experimenter for the task description

B) Execution

- Note execution **start** time (in minutes) here:
  _______________ (e.g. 11:26)
- Read the task description and **do the task** (you can run the TSAFE system test at any time to ensure the system is working correctly)
- Note execution **stop** time (in minutes) here:
  _______________ (e.g. 13:26)
- Fill out debriefing questionnaire and submit your source code to your experimenter

Figure 64: Experiment Compliance: Experiment Procedures

B.2 Experiment Object Description

Tactical Separation Assisted Flight Environment (TSAFE)

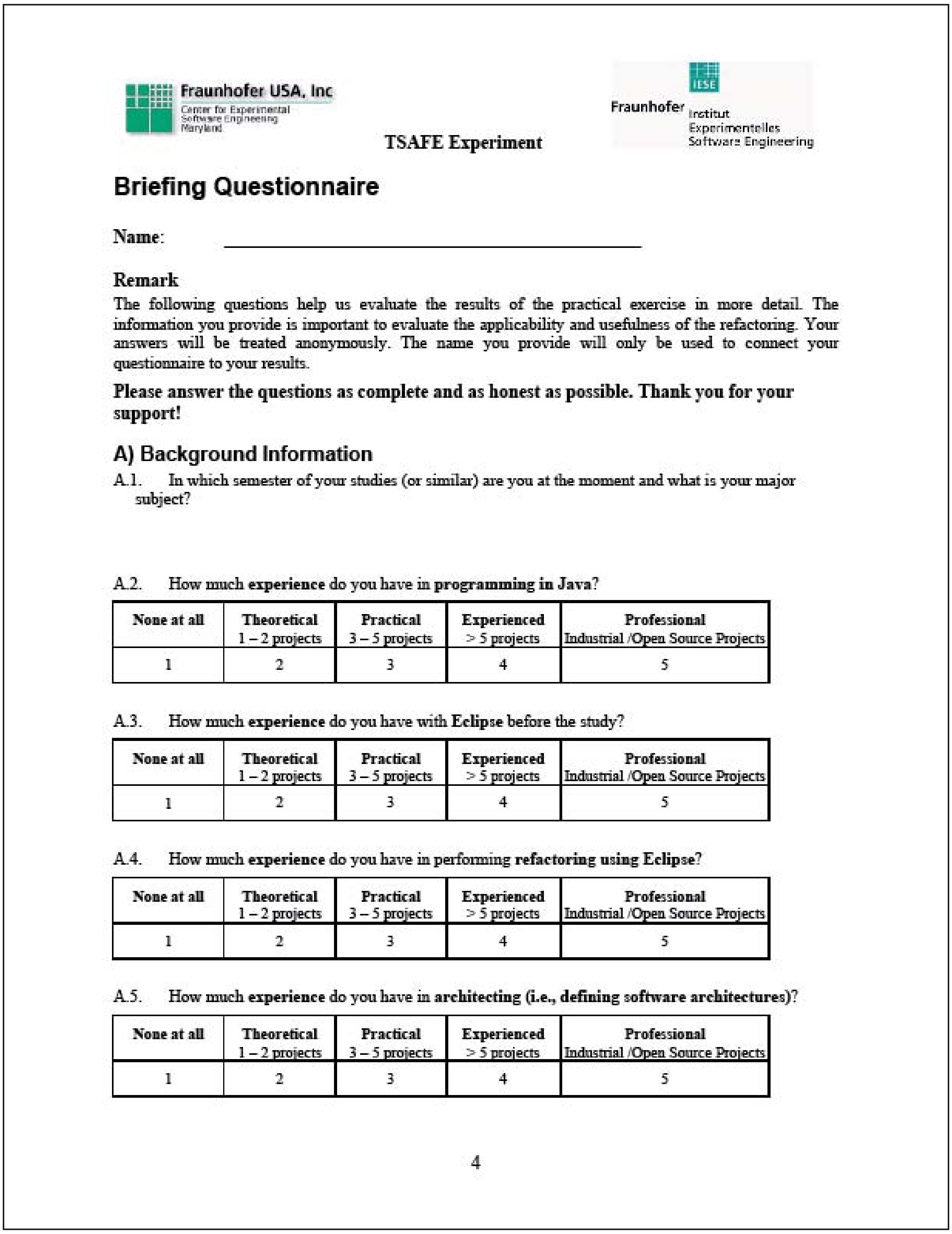
The experiment investigates how the **quality of an architecture** is affected by refactorings. Refactoring here means changing the decomposition of the source code (creating, moving, merging or splitting of methods, classes, packages or other source code elements) without altering the external behavior of the system. Hence, refactoring aims at improving the internal structure of the source code.

We will ask you to perform a refactoring of the TSAFE system. We will evaluate the quality of the restructured source code. High quality is achieved when the architecture is decomposed into components that have **high internal cohesion and low coupling to external components**. After the experiment, we will analyze how well your restructured source code matches the reference solution created by several TSAFE experts.

Please read this document carefully and contact your experimenter in case you have any problems in running TSAFE. Typically, the preparation takes about 15 – 30 minutes.

Please try to perform the task as fast as possible but aim at achieving a high quality. The maximal allotted time for the executing task is 2 hours. Please note that it took the experts between 0.25 hours to 2 hours to complete the refactoring task.

B.3 Briefing Questionnaire



Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Fraunhofer Institut Experimentelles Software Engineering

TSAFE Experiment

Briefing Questionnaire

Name: ____

Remark

The following questions help us evaluate the results of the practical exercise in more detail. The information you provide is important to evaluate the applicability and usefulness of the refactoring. Your answers will be treated anonymously. The name you provide will only be used to connect your questionnaire to your results.

Please answer the questions as complete and as honest as possible. Thank you for your support!

A) Background Information

A.1. In which semester of your studies (or similar) are you at the moment and what is your major subject?

A.2. How much **experience** do you have in **programming in Java**?

None at all	**Theoretical** 1 – 2 projects	**Practical** 3 – 5 projects	**Experienced** > 5 projects	**Professional** Industrial /Open Source Projects
1	2	3	4	5

A.3. How much **experience** do you have with **Eclipse** before the study?

None at all	**Theoretical** 1 – 2 projects	**Practical** 3 – 5 projects	**Experienced** > 5 projects	**Professional** Industrial /Open Source Projects
1	2	3	4	5

A.4. How much **experience** do you have in performing **refactoring using Eclipse**?

None at all	**Theoretical** 1 – 2 projects	**Practical** 3 – 5 projects	**Experienced** > 5 projects	**Professional** Industrial /Open Source Projects
1	2	3	4	5

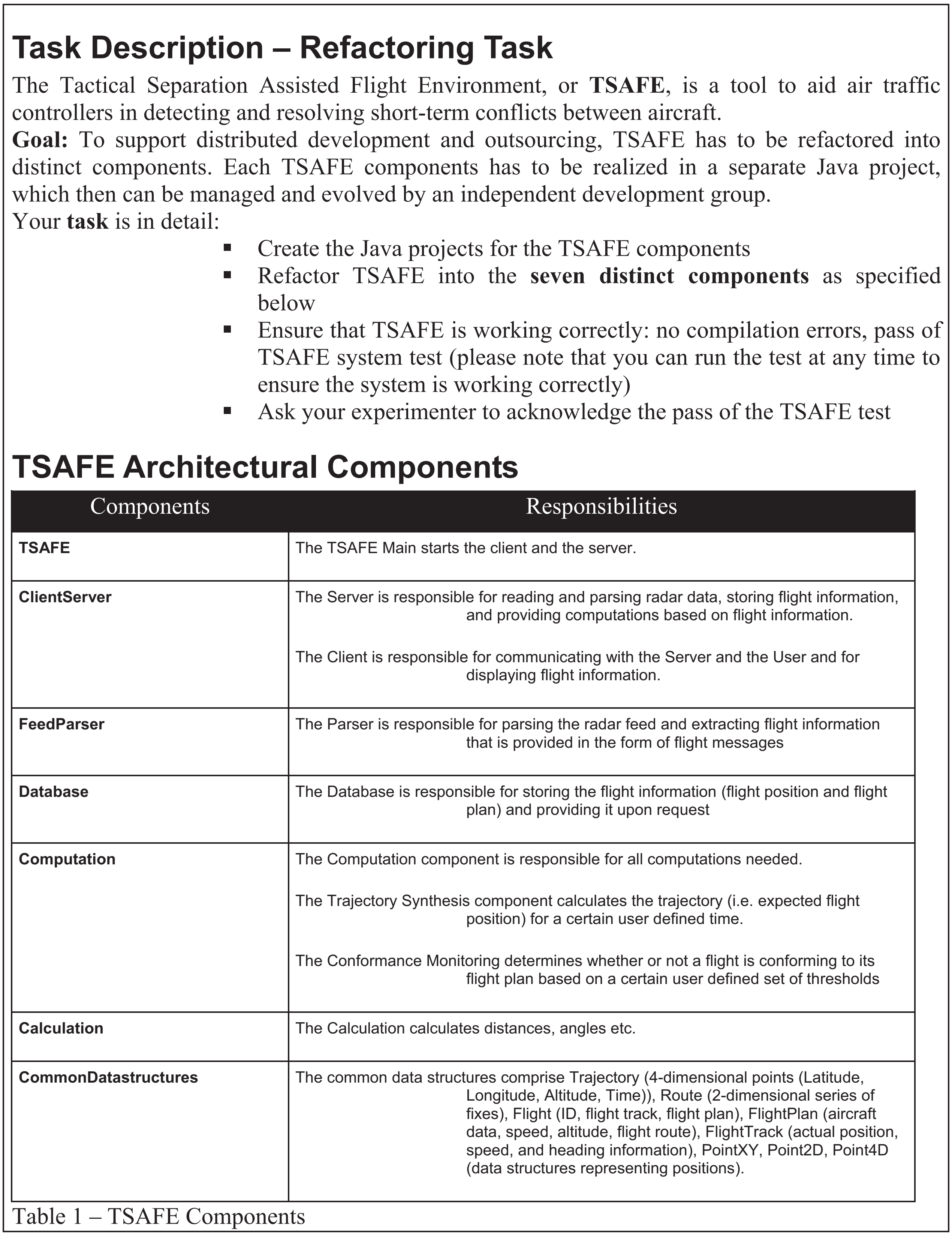
A.5. How much **experience** do you have in **architecting (i.e., defining software architectures)**?

None at all	**Theoretical** 1 – 2 projects	**Practical** 3 – 5 projects	**Experienced** > 5 projects	**Professional** Industrial /Open Source Projects
1	2	3	4	5

4

Figure 65: Experiment Compliance: Briefing Questionnaire

B.4 Task Description



Task Description – Refactoring Task

The Tactical Separation Assisted Flight Environment, or **TSAFE**, is a tool to aid air traffic controllers in detecting and resolving short-term conflicts between aircraft.
Goal: To support distributed development and outsourcing, TSAFE has to be refactored into distinct components. Each TSAFE components has to be realized in a separate Java project, which then can be managed and evolved by an independent development group.
Your **task** is in detail:

- Create the Java projects for the TSAFE components
- Refactor TSAFE into the **seven distinct components** as specified below
- Ensure that TSAFE is working correctly: no compilation errors, pass of TSAFE system test (please note that you can run the test at any time to ensure the system is working correctly)
- Ask your experimenter to acknowledge the pass of the TSAFE test

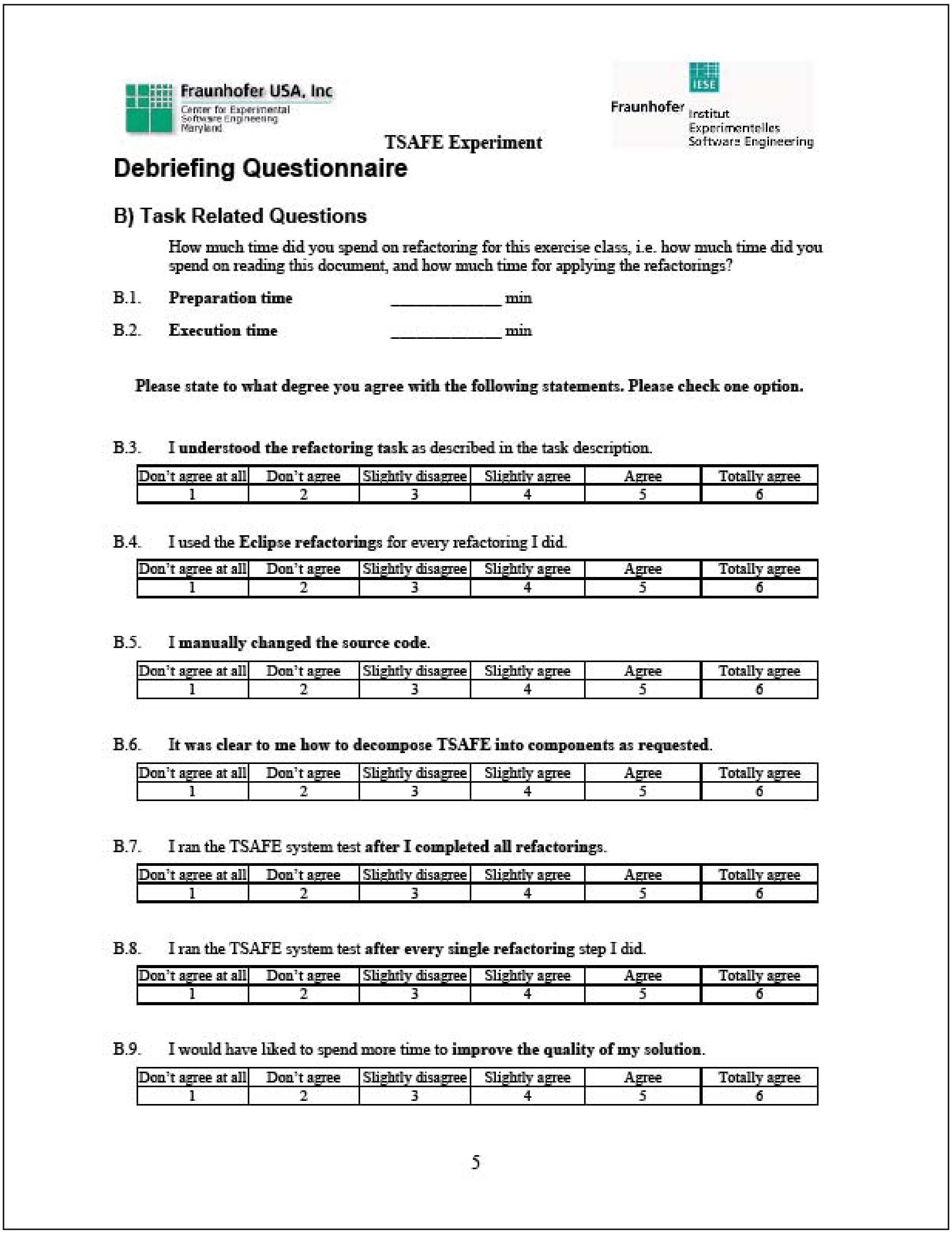
TSAFE Architectural Components

Components	Responsibilities
TSAFE	The TSAFE Main starts the client and the server.
ClientServer	The Server is responsible for reading and parsing radar data, storing flight information, and providing computations based on flight information. The Client is responsible for communicating with the Server and the User and for displaying flight information.
FeedParser	The Parser is responsible for parsing the radar feed and extracting flight information that is provided in the form of flight messages
Database	The Database is responsible for storing the flight information (flight position and flight plan) and providing it upon request
Computation	The Computation component is responsible for all computations needed. The Trajectory Synthesis component calculates the trajectory (i.e. expected flight position) for a certain user defined time. The Conformance Monitoring determines whether or not a flight is conforming to its flight plan based on a certain user defined set of thresholds
Calculation	The Calculation calculates distances, angles etc.
CommonDatastructures	The common data structures comprise Trajectory (4-dimensional points (Latitude, Longitude, Altitude, Time)), Route (2-dimensional series of fixes), Flight (ID, flight track, flight plan), FlightPlan (aircraft data, speed, altitude, flight route), FlightTrack (actual position, speed, and heading information), PointXY, Point2D, Point4D (data structures representing positions).

Table 1 – TSAFE Components

Figure 66: Experiment Compliance: Task Description

B.5 Debriefing Questionnaire



Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

Fraunhofer IESE Institut Experimentelles Software Engineering

TSAFE Experiment

Debriefing Questionnaire

B) Task Related Questions

How much time did you spend on refactoring for this exercise class, i.e. how much time did you spend on reading this document, and how much time for applying the refactorings?

B.1. **Preparation time** ____________ min

B.2. **Execution time** ____________ min

Please state to what degree you agree with the following statements. Please check one option.

B.3. **I understood the refactoring task** as described in the task description.

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

B.4. I used the **Eclipse refactorings** for every refactoring I did.

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

B.5. **I manually changed the source code.**

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

B.6. **It was clear to me how to decompose TSAFE into components as requested.**

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

B.7. I ran the TSAFE system test **after I completed all refactorings.**

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

B.8. I ran the TSAFE system test **after every single refactoring** step I did.

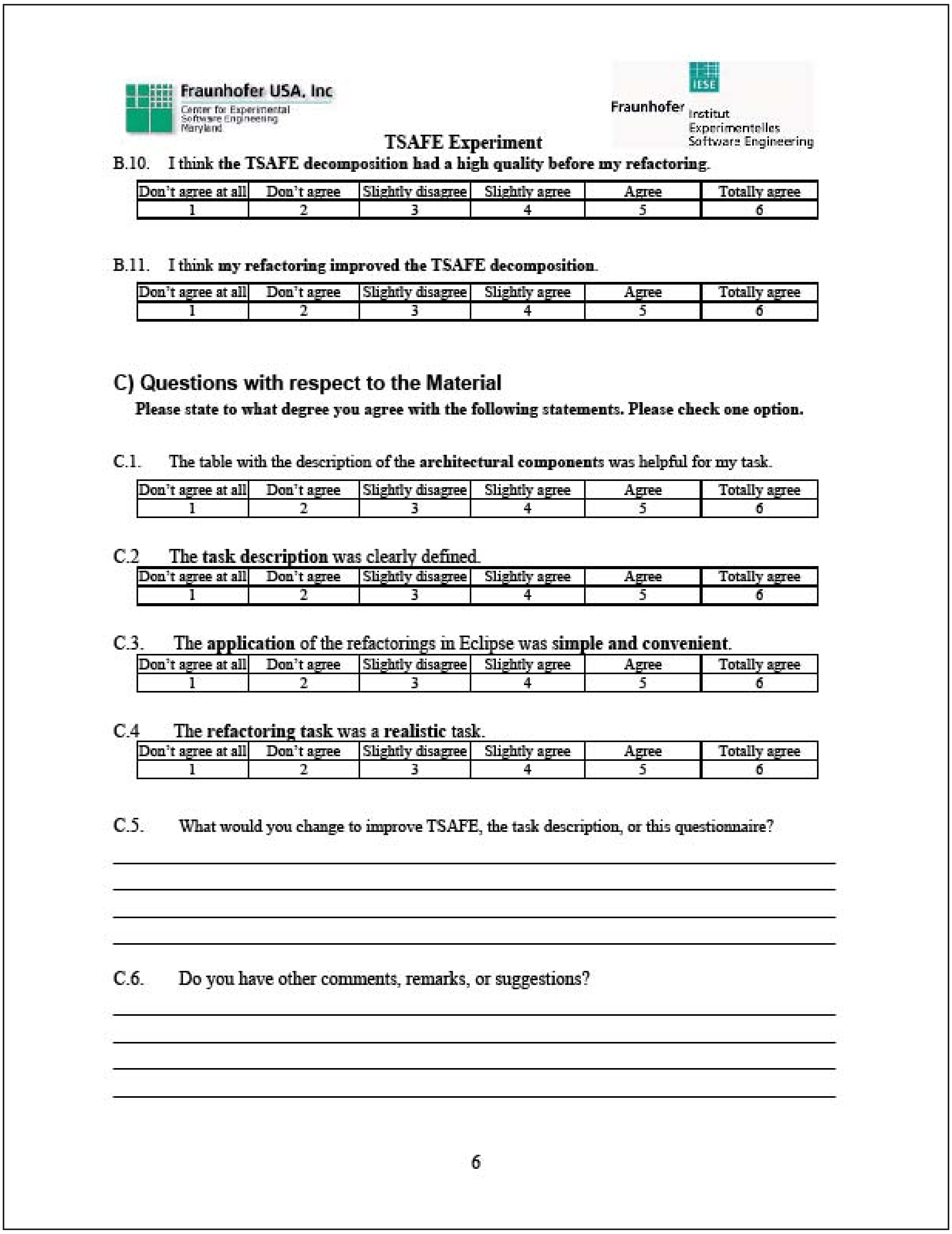
Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

B.9. I would have liked to spend more time to **improve the quality of my solution.**

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

5

Figure 67: Experiment Compliance: Debriefing Questionnaire



Fraunhofer USA, Inc
Center for Experimental Software Engineering Maryland

IESE
Fraunhofer Institut Experimentelles Software Engineering

TSAFE Experiment

B.10. I think **the TSAFE decomposition had a high quality before my refactoring.**

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

B.11. I think **my refactoring improved the TSAFE decomposition.**

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

C) Questions with respect to the Material

Please state to what degree you agree with the following statements. Please check one option.

C.1. The table with the description of the **architectural components** was helpful for my task.

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

C.2 The **task description** was clearly defined.

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

C.3. The **application** of the refactorings in Eclipse was **simple and convenient**.

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

C.4 The **refactoring task** was a **realistic** task.

Don't agree at all	Don't agree	Slightly disagree	Slightly agree	Agree	Totally agree
1	2	3	4	5	6

C.5. What would you change to improve TSAFE, the task description, or this questionnaire?

____

____

____

____

C.6. Do you have other comments, remarks, or suggestions?

____

____

____

____

6

B.6 Experiment Results

ID = Group_Type_Number: A = Group A TSAFE1, B = Group B TSAFE2

P = Pilot Students, F = Maryland Students, I = Industrial (ArQuE), K = Kaiserslautern Students (GSE 2008)

B.6.1 Results Subject Performance

ID	Group	Type	Performance		
			B.1	B.2	
ID			Preparation time	Execution time	Correctness Achieved
A_P_01	A	P	15	54	64
B_P_01	B	P	20	60	100
A_F_01	A	F	36	78	79
A_F_02	A	F	46	150	43
B_F_01	B	F	16	47	100
B_F_02	B	F	25	53	100
A_I_01	A	I	15	45	86
A_I_02	A	I	18	83	64
A_I_03	A	I	20	135	86
A_I_04	A	I	52	150	28
B_I_01	B	I	27	54	100
B_I_02	B	I	23	63	100
B_I_03	B	I	45	115	100
B_I_04	B	I	25	120	100
A_K_01	A	K	23	133	79
A_K_02	A	K	14	145	79
A_K_03	A	K	13	136	35
A_K_04	A	K	29	159	0
A_K_05	A	K	10	105	100
A_K_06	A	K	11	108	86
A_K_07	A	K	11	85	86
A_K_08	A	K	11	57	100
A_K_09	A	K	14	118	93
B_K_01	B	K	29	54	100
B_K_02	B	K	32	29	100
B_K_03	B	K	25	40	93
B_K_04	B	K	29	44	100
B_K_05	B	K	29	36	100
B_K_06	B	K	15	35	100
B_K_07	B	K	30	26	100

Figure 68: Experiment Compliance: Results Subject Performance

B.6.2 Results Briefing Questionnaire: Subject Background

ID	Group	Type	BACKGROUND				
			A.1	A.2	A.3	A.4	A.5
ID			Semester	experience in Java	experience in Eclipse	experience in Refactoring	experience in Architecting
A_P_01	A	P	7	4	4	4	1
B_P_01	B	P	6	4	4	4	4
A_F_01	A	F	9	3	3	1	2
A_F_02	A	F	6	2	2	1	2
B_F_01	B	F	6	3	4	4	4
B_F_02	B	F	6	3	3	1	2
A_I_01	A	I	20	4	4	2	4
A_I_02	A	I	40	3	3	2	2
A_I_03	A	I	15	4	4	2	2
A_I_04	A	I	40	1	2	1	2
B_I_01	B	I	74	2	2	1	4
B_I_02	B	I	70	3	2	1	2
B_I_03	B	I	30	1	3	1	4
B_I_04	B	I	15	1	1	1	2
A_K_01	A	K	6	2	2	1	2
A_K_02	A	K	6	3	2	1	2
A_K_03	A	K	9	2	1	1	2
A_K_04	A	K	7	2	1	1	1
A_K_05	A	K	5	2	2	1	2
A_K_06	A	K	5	3	3	2	2
A_K_07	A	K	5	3	3	1	2
A_K_08	A	K	5	3	3	1	2
A_K_09	A	K	5	2	2	2	2
B_K_01	B	K	7	2	2	1	2
B_K_02	B	K	5	2	2	2	2
B_K_03	B	K	8	2	2	1	1
B_K_04	B	K	6	2	2	1	2
B_K_05	B	K	6	3	2	1	2
B_K_06	B	K	6	2	2	2	1
B_K_07	B	K	6	3	3	1	2

Figure 69: Experiment Compliance: Results Briefing Questionnaire

B.6.3 Results Debriefing Questionnaire: Task Related Questions

ID	Group	Type	Task-								
			B.3	B.4	B.5	B.6	B.7	B.8	B.9	B.10	B.11
ID			understa nding of the task	usage of Eclipse refactori ng	manual changes to the source	decomp osition clear	sytem test after all refactori	system test after every single	more time spend to improve	high decomp osition quallity	imporve ment by refactori ng
A_P_01	A	P	3								
B_P_01	B	P	5								
A_F_01	A	F	3	6	1	4	6	2	5	4	4
A_F_02	A	F	4	4	6	2	1	1	3	5	1
B_F_01	B	F	4	5	2	5	5	4	5	6	5
B_F_02	B	F	5	4	4	2	6	2	3	5	3
A_I_01	A	I	4	6	1	5	6	4	5	4	5
A_I_02	A	I	3	5	1	5	5	3	2	4	2
A_I_03	A	I	5	5	4	4	5	4	4	5	4
A_I_04	A	I	5	5	1	2	1	1	1	3	1
B_I_01	B	I	2	6	1	4	6	1	6	1	6
B_I_02	B	I	4	5	1	3	5	1	4	2	4
B_I_03	B	I	4	4	4	3	1	2	4	4	4
B_I_04	B	I	4	2	4	4	5	2	6	3	3
A_K_01	A	K	5	6	4	4	5	1	4	4	3
A_K_02	A	K	5	5	2	5	6	1	6	2	4
A_K_03	A	K	4	2	2	2	4	3	1	3	3
A_K_04	A	K	5	6	1	6	6	1	6	5	5
A_K_05	A	K	5	6	4	5	6	2	2	5	4
A_K_06	A	K	6	6	1	5	6	1	2	5	5
A_K_07	A	K	5	6	1	4	6	2	2	3	5
A_K_08	A	K	5	6	2	5	6	1	4	2	5
A_K_09	A	K	5	5	2	4	6	1	3	3	4
B_K_01	B	K	5	6	1	4	6	1	4	4	4
B_K_02	B	K	6	6	2	6	6	2	3	6	6
B_K_03	B	K	5	6	1	5	6	1	1	1	6
B_K_04	B	K	5	6	1	5	6	6	4	4	4
B_K_05	B	K	6	6	1	4	6	1	3	2	4
B_K_06	B	K	5	6	1	5	6	2	2	5	4
B_K_07	B	K	6	6	1	6	6	1	2	6	4

Figure 70: Experiment Compliance: Results Debriefing Questionnaire: Task Related Questions

B.6.4 Results Debriefing Questionnaire: Questions with Respect to Material

ID	Group	Type	Material			
			C.1	C.2	C.3	C.4
ID			description of architectural components	task description clear	application of Eclipse refactorings	realistic task
A_P_01	A	P	3	5	6	5
B_P_01	B	P	6	6	6	6
A_F_01	A	F	3	4	4	4
A_F_02	A	F	4	5	4	5
B_F_01	B	F	2	5	5	5
B_F_02	B	F	5	5	4	4
A_I_01	A	I	4	4	5	4
A_I_02	A	I	5	4	4	2
A_I_03	A	I	4	5	5	5
A_I_04	A	I	4	3	2	3
B_I_01	B	I	6	4	5	3
B_I_02	B	I	4	3	3	5
B_I_03	B	I	5	5	5	4
B_I_04	B	I	5	4	4	4
A_K_01	A	K	4	5	5	5
A_K_02	A	K	3	5	2	5
A_K_03	A	K	5	5	2	4
A_K_04	A	K	6	6	6	6
A_K_05	A	K	6	5	4	5
A_K_06	A	K	4	6	5	5
A_K_07	A	K	5	5	4	6
A_K_08	A	K	4	5	6	4
A_K_09	A	K	5	5	4	3
B_K_01	B	K	5	5	5	5
B_K_02	B	K	5	5	5	5
B_K_03	B	K	5	6	5	4
B_K_04	B	K	5	5	5	5
B_K_05	B	K	5	5	6	4
B_K_06	B	K	5	5	6	6
B_K_07	B	K	6	5	5	3

Figure 71: Experiment Compliance: Results Debriefing Questionnaire: Questions with Respect to Material

Appendix C Example Source Code DRVFaçade

This appendix section lists the source of the example used in the introduction (see Section 3.2 and 3.3).

The method "doit()" shows the architecture-compliant implementation of this example, while the method "doitWrong()" causes a structural violation.

C.1 Class BusinessLogic.java

```
package businesslogic;

import driver.*;

public class BusinessLogic {

        public void doit(){
                DriverFacade.activate();
        }

        public void doitWrong(){
                HardwareDriver.activate();
                //EmulationDriver.activate();
        }

        public static void main(String args[]) {
                BusinessLogic myLogic = new BusinessLogic();
                DriverFacade.mode = DriverFacade.HARDWARE;
                myLogic.doit();
                myLogic.doitwrong();
        }
}
```

C.2 Class DriverFacade.java

```java
package driver;

public class DriverFacade {
        public static int mode;
        public static final int EMULATION = 0;
        public static final int HARDWARE = 1;

        public static void activate(){
                switch (mode){
                        case EMULATION:
                                EmulationDriver.activate();
                                 break;
                        case HARDWARE:
                                HardwareDriver.activate();
                                 break;
                }
        }
}
```

C.3 Class HardwareDriver.java

```java
package driver;

public class HardwareDriver {

        public static void activate() {
                doit();
        }

        private static void doit() {
                System.out.println("Hardware executed.");
        }
}
```

C.4 Class EmulationDriver.java

```java
package driver;

public class EmulationDriver {

        public static void activate() {
                doit();
        }

        private static void doit() {
                System.out.println("Emulation executed.");
        }
}
```

Appendix D Algorithms SAVE LiFe in Pseudo Code

D.1 Algorithms Architecture Manager: SAVE LiFe Fat Client

The architect executes the methods *publishArchitecture()* and *requestComplianceStatus()* on demand. Both methods are accessible from the user interface of the Architecture Manager.

D.1.1 Method: publishArchitecture

```
// method publishArchitecture
public int publishArchitecture() {
       // get client connected to server (auto-connect if not yet connected)
       ArchitectureManager client = ArchitectureManagerClient.getClient();
       ComplianceChecker server = ArchitectureManagerClient.getServer();

       // transferData = local data models managed by architect: (1) structural
       // model and (2) mapping, the data models to be transferred are encoded
       // as String arrays, the client ArchitectureManager always manages the
       // latest version of structural model and mapping
       StructuralModel structure = client.getStructuralModel();
       MappingModel mapping = client.getMapping();
       TransferData transferData =
              client.buildTransferData(structure, mapping);

       // remote call to server to transfer structural model and mapping model
       // as part of trasferData
       boolean Ok = server.receivePublishedArchitecture(transferData);

       // structural model and mapping have been transferred successfully to
       // server (i.e., compliance checker)
       if (Ok) { return; }
       // show error information
       else { displayErrorMSG();    }
}
```

D.1.2 Method: requestComplianceStatus

```
// method requestComplianceStatus
public int requestComplianceStatus() {
        // get client connected to server (auto-connect if not yet connected)
        ArchitectureManager client =
                ArchitectureManagerClient.getClient();
        ComplianceChecker server =
                ArchitectureManagerClient.getServer();

        // remote call to server to request the transfer compliance status model
        // from compliance checker, the compliance status model comprises the
        // current overall compliance status of the whole system under
        development
        TransferData transferData = server.publishComplianceStatus();
        ComplianceStatusModel complianceStatus =
              transferData.retrieveComplianceStatusModel();
        client.setComplianceStatus(compliance)

        // visualize the compliance status model using a graphical diagram,
        // which enables the architect to reason on the compliance of the
        // overall system
        client.visualizeComplianceStatus()
}
```

D.2 Algorithms Development Monitor: SAVE LiFe Thin Client

The development monitor tracks the work of each developer in the integrated development environment individually. The development monitor hooks into the incremental project builder of the development environment (e.g., for Eclipse *org.eclipse.core.internal.events*). The builder is executed whenever physical resources (i.e., files or folders) are changed and saved. When executed the method *monitorCodeandSendDelta()* of the development monitor is invoked automatically. Hence, the method is executed for any change made to the source code.

When the server has computed the results, it invokes *receiveLiveFeedback()* remotely to transfer the results, potentially including the violation.

D.2.1 Method: monitorCodeAndSendDelta

```
// method monitorCodeandSendDelta
public void monitorCodeandSendDelta() {
        // get client connected to server (auto-connect if not yet connected)
        DevelopmentMonitor client = DevelopmentMonitorClient.getClient();
        ComplianceChecker server = DevelopmentMonitorClient.getServer();

        // determine modified files in projects currently edited
        Files[] modifiedFiles = determineLocalDelta(ResourcesPlugin.
                        getWorkspace().getRoot().getProjects();

        // remote call to send delta (i.e., modified files) to server
        TransferData transferData = client.buildTransferData(modifiedFiles);
        Boolean Ok = server.receiveDelta(transferData);
        // delta (i.e., modified files) hasstructural model and mapping have
        // been transferred successfully to server (i.e., compliance checker)
        if (Ok) { return; }
        // show error information
        else { displayErrorMSG(); }
}
```

D.2.2 Method: determineLocalDelta

```
// method determineLocalDelta
public Files[] determineLocalDelta (IProject[] projects) {
        foreach project in projects {
                foreach file in project {
                        // check file currently edited for modifications
                        Status status = ConfigurationManagementAdapter
                                        .getFileStatus.getResource(file);

                        if (status == Status.MODIFIED) {
                                //determine local delta of modifications, the
                                resulting delta comprises a model of the file
                                modifiedFiles[].add(file)
                        }
                }
        }
        // the delta comprises the source code files with modification made by
        // the developer, each developers
        return modifiedFiles[];
}
```

D.2.3 Method: receiveLiveFeedback

```
// method receiveLiveFeedback remote called by server
public void receiveLiveFeedback(TransferData transferData) {
        // get client connected to server (auto-connect if not yet connected)
        DevelopmentMonitor client = DevelopmentMonitorClient.getClient();
        ComplianceChecker server = DevelopmentMonitorClient.getServer();

        // remotely called by server to transfer the results of the compliance
        // checking for the last delta sent to server
        DeltaResultModel deltaResults =
                transferData.retrieveComplianceStatusModel();
        client.setDeltaResults(deltaResults)

        // displayDeltaResult
        client.displayDeltaResults();
}
```

D.2.4 Method displayDeltaResult

```
// method determineLocalDelta
public void displayDeltaResutl() {
        DevelopmentMonitor client = DevelopmentMonitorClient.getClient();
        // get list of DeltaResults to display the single deltaResult, which
        // represent the architecture violations computed by the server
        DeltaResultModel deltaResults = client.getDeltaResults();

        foeach deltaResult in deltaResults {
                // resolve corresponding element in editor
                String projectName = deltaResult.getProjectName();
                IPath path = deltaResult.getPath();
                IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
                IProject project = root.getProject(projectName);
                ICompilationUnit unit = project
                                .findElement(deltaResult.getCompilationUnit());

                // clear existing marker, and create new ones for current
                // violations, and show them to developer
                unit.clearMarker();
                unit.createMarkers(deltaResult.getViolation());
                unit.showMarkers();
        }
}
```

D.3 Algorithms Compliance Checker: SAVE LiFe Server

The methods of the Compliance Checker are remotely triggered by the respective client. The ArchitectureManager invokes either *receivePublishedArchitecture()* or *publishComplianceStatus()*, while the DevelopmentMonitor invokes *sendDelta()*.

After compliance checking has been executed, the server invokes *receiveLiveFeedback()* in the DevelopmentMonitor so developers become aware of the violations – if present – promptly.

D.3.1 Method: receivePublishedArchitecture

```
// method receivePublishedArchitecture remote called by client
public void receivePublishedArchitecture(TransferData transferData) {
        // get client connected to server (auto-connect if not yet connected)
        ArchitectureManager client =
                ComplianceCheckerServer.getArchitectureClient();
        ComplianceChecker server = ComplianceCheckerServer.getServer();

        //update the data models of server
        client.updateStructuralModel(transferData);
        client.updateMapping(transferData);
}
```

D.3.2 Method: updateStructuralModel

```
// method updateStructuralModel
public void updateStructuralModel (TransferData transferData) {
        // updating existing structural model with new structural model sent by
        // client
        StructuralModel structure = transferData.retrieveStructure();
        server.setStructure(structure)
}
```

D.3.3 Method: updateMapping

```
// method updateMapping
public void updateMapping(TransferData transferData) {
        // updating existing mapping with new mapping sent by client
        MappingModel mapping = transferData.retrieveStructure();
        server.setMapping(mapping)
}
```

D.3.4 Method: publishComplianceStatus

```
// method publishComplianceStatus, remotely called by client
public void publishComplianceStatus() {
        // get client connected to server (auto-connect if not yet connected)
        ArchitectureManager client =
                        ComplianceCheckerServer. getArchitectureClient();
        ComplianceChecker server = ComplianceCheckerServer.getServer();

        // remote call to server to request the transfer compliance status model
        // from compliance checker, the compliance status model comprises the
        // current overall compliance status of the whole system under
        // development
        TransferData transferData = server.publishComplianceStatus();
        ComplianceStatusModel complianceStatus
        =transferData.retrieveComplianceStatusModel();
                client.setComplianceStatus(compliance)
}
```

D.3.5 Method: receiveDelta

```
// method receiveDelta remotely called by client
public void receiveDelta(TransferData transferData) {
        // get client connected to server (auto-connect if not yet connected)
        DevelopmentMonitor client =
                ComplianceCheckerServer.getDevelopmentClient();
        ComplianceChecker server = ComplianceCheckerServer.getServer();

        // get modified files and create empty delta source code model
        Files[] modifiedFiles = transferData.retrieveModifiedFiles();
        DeltaModel deltaModel = new DeltaModel();

        // process each file locally modified by a developer and extract facts,
        // the delta model represents all locally modified files
        foreach file in modifiedFiles {
                //cast file to compilation unit and extract delta facts
                ICompilationUnit unit = file.getCompilationUnit();
                DeltaModel deltaModel.add(
                        server.extractDeltaFacts(unit, deltaModel));
        }
        // update the source code model and keep information on history
        server.updateSourceCodeModel(deltaModel);

        // check compliance for the delta modified by a developer and distill
        // the violations (note: convergences and absences are filtered), the
        // delta violations are the spots causing violations (i.e., source code
        // comprising violating statements)
        DeltaComplianceStatus deltaCompliance =
                        server.checkCompliance(deltaModel);
        deltaCompliance = server.distillDeltaViolations(deltaCompliance);

        // remote call to client to transfer the delta violations, the model
        // comprising the list of violations, this information raises the
        // awareness of each developer on client-side on violations caused by
        // him or currently present in the files modified, thus, develop can
        // correct the code and remove the code.
        TransferData transferData = client.buildTransferData(deltaCompliance);
        client.sendLiveFeedback(tranferData);
}
```

D.3.6 Method: extractDeltaFacts

```
// method extractDeltaFacts
public DeltaSourceCodeModel extractDeltaFacts(ICompilationUnit unit,
                                              DeltaSourceCodeModel deltaModel) {
       //parse the compilation unit with a programming language-specific parser
       and extract all dependencies caused by the file to other files or
       compilation units
       CompilationUnitHandler handler = new CompilationUnitHandler();
       DeltaSourceCodeModel deltaModel =
                      handler.parseCompilationUnit(unit)
       return deltaModel;
}
```

D.3.7 Method: parseCompilationUnit

```
// method parseCompilationUnit
private DeltaSourceCodeModel parseCompilationUnit(ICompilationUnit unit,
                                             DeltaSourceCodeModel deltaModel) {
       // the ASTParser extract all dependencies caused by the compilation unit
       to other compilation units, hence the results are the delta facts
       ASTParser parser = ASTParser.newParser().setSource(unit);
              CompilationUnit rootCU = (CompilationUnit)
              parser.createAST(null);
       if (rootCU != null) {
              rootCU.accept(new ASTVisitor(unit, deltaModel);
       }
       return deltaModel;
}
```

D.3.8 Method: updateSourceCodeModel

```
// method updateSourceCodeModel
public void updateSourceCodeModel(DeltaSourceCodeModel deltaModel) {
        // get current date to mark point in time of modifications made
        GregorianCalendar date = new GregorianCalendar();

        // get the source code model
        SourceCodeModel sourceModel = server.getSourceCodeModel();

        // iterate over the elements of the deltaModel and update the source
        // code model respectively, because all modification are sent to the
        // server, the source code model managed by the central server is always
        // up-to-date
        foreach modelElement in deltaModel {
                // element exists already, update point in time and investigate
                // dependencies
                if (sourceModel.exists(modelElement) == true) {
                        sourceModel.updateState(modelElement, date);

                        // iterate over the dependencies of the model elements of
                        // the deltaModel and update the source code model
                        foreach modelDependency of modelElement {

                                // dependency exists already, update modification
                                // point in time
                                if (modelDependency.existsInSourceCodeModel()
                                                == true) {
                                        sourceModel.updateState(modelElement,
                                                    modelDependency, date);
                                } else {
                                        // dependency does not exist, create
                                        // dependency in source code model with
                                        modification point in time
                                        sourceModel.add(modelElement,
                                                modelDependency, date);
                                }
                        }
                } else {
                // dependency does not exist, create model element and its
                // dependencies in source code model with modification point in
                // time
                        sourceModel.addModelElementWithDependencies(modelElement,
                                state);
                }

                foreach modelDependency of modelElement {
                        if (modelDependency.getModificationDate < date) {
                                // source code model is cleaned, in case a
                                // dependency no longer was removed
                                sourceModel.deleteDependency(modelDependency,
                                         date);
                        }
                }
        }
}
```

D.3.9 Method checkCompliance

```
// method checkCompliance
public DeltaComplianceStatus checkCompliance(DeltaSourceCodeModel deltaModel) {
        // get computational models required for compliance checking
        StructuralModel structure = server.getStructuralModel();
        MappingModel mapping = server.getMapping();

        // the lifting operator resolves the mapping and results in the
        // liftedCodeModel, which is a representation of the source code model
        // on the abstraction level of the structural model, hence, both models
        // can be compared
        StructuralModel liftedCodeModel = server.lift(mapping, deltaModel);

        // create empty result container to store the compliance checking
        // results
        DeltaComplianceStatus deltaCompliance = new DeltaComplianceStatus();

        // check presence if planned dependencies of structure to identify
        // absences
        foreach plannedDependency in structure {
                plannedSourceElement = plannedDepdency.getSourceElement();
                plannedTargetElement = plannedDepdency.getTargetElement();

                foreach actualDependency in liftedCodeModel {
                        if (actualDependency.qetSourceElement()
                                        .equals(plannedSourceElement)) {
                                // plan matches actual, dependency is CONVERGENCE
                                deltaCompliance.add(plannedDependency,
                                                    CONVERGENCE);
                                break;
                        }
                }

                // plan did not match actual, dependency is ABSENCE
                deltaCompliance.add(plannedDependency, ABSENCE);
        }

        // check if actual dependencies are planned in structure to identify
        // divergences (i.e., violation dependencies)
        foreach actualDependency in liftedCodeModel {
                actualSourceElement = actualDepdency.getSourceElement();
                actualTargetElement = actualDepdency.getTargetElement();

                foreach plannedDependency in structure{
                        if (actualDependency.qetSourceElement()
                                        .equals(actualSourceElement)) {
                                // actual matches plan, dependency is CONVERGENCE
                                deltaCompliance.add(actualDependency,
                                                    CONVERGENCE);
                                break;
                        }
                }

                // actual did not match plan, dependency is DIVERGENCE
                deltaCompliance.add(actualDependency, DIVERGENCE);
        }
        // return compliance checking results (deltaCompliance), the totality of
        // CONVERGENCEs, ABSENCEs, and DIVERGENCEs
        return deltaCompliance;
}
```

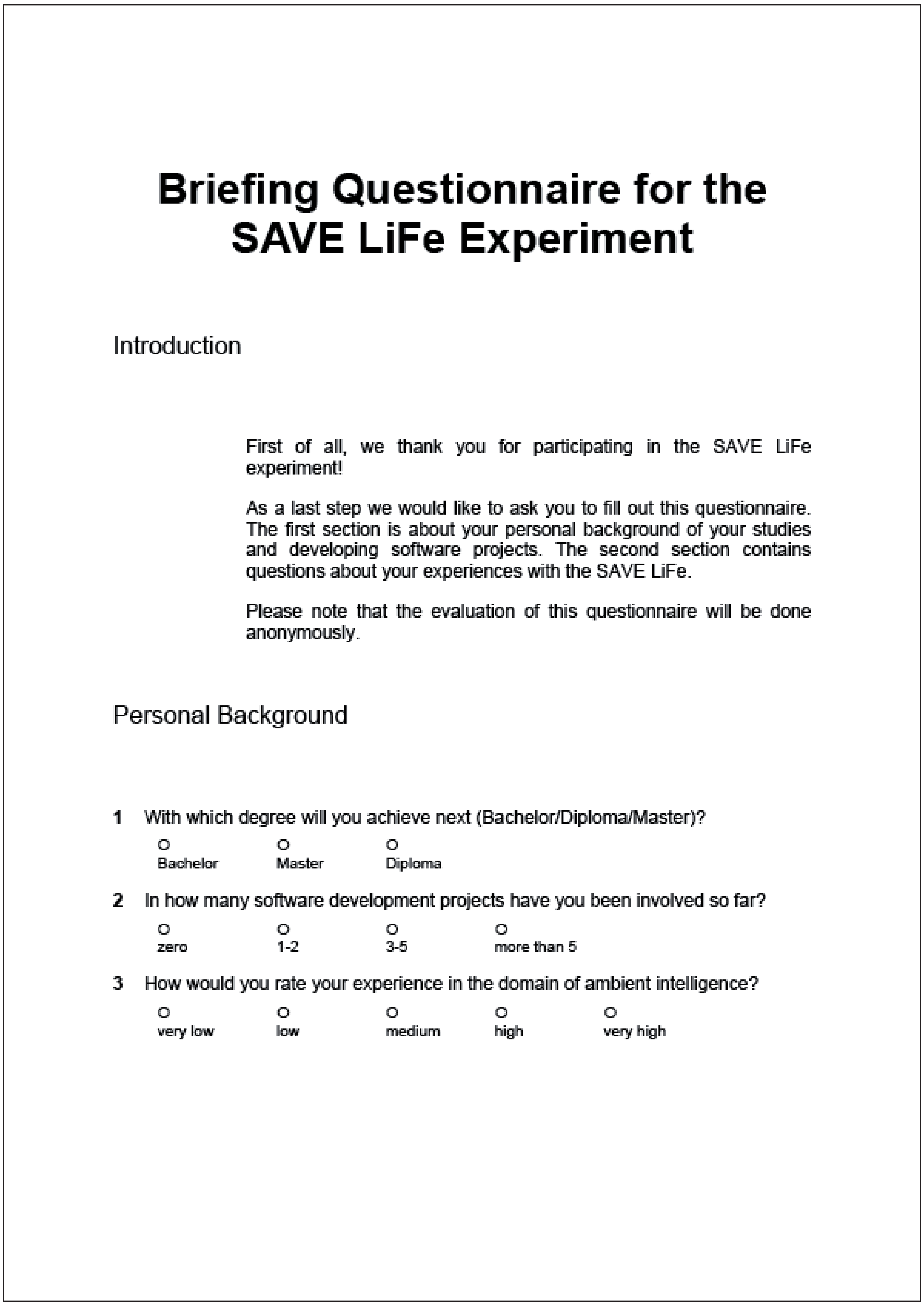
D.3.10 Method: distillViolations

```
// method distillViolations
public DeltaComplianceStatus distillViolations(DeltaComplianceStatus
                                                        deltaCompliance) {
        // filter for compliance status for divergences only
        foreach dependency in deltaCompliance {
                if (dependency.getStatus() == CONVERGENCE) {
                        deltaCompliance.remove(dependency);
                }
                if (dependency.getStatus() == ABSENCE) {
                        deltaCompliance.remove(dependency);
                }
        }
        return deltaCompliance;
}
```

Appendix E Experiment Live Feedback

This section presents the material used for the experiment on live compliance checking with SAVE LiFe as described in Section 6.1.

E.1 Briefing Questionnaire



Briefing Questionnaire for the SAVE LiFe Experiment

Introduction

First of all, we thank you for participating in the SAVE LiFe experiment!

As a last step we would like to ask you to fill out this questionnaire. The first section is about your personal background of your studies and developing software projects. The second section contains questions about your experiences with the SAVE LiFe.

Please note that the evaluation of this questionnaire will be done anonymously.

Personal Background

1 With which degree will you achieve next (Bachelor/Diploma/Master)?

O Bachelor	O Master	O Diploma

2 In how many software development projects have you been involved so far?

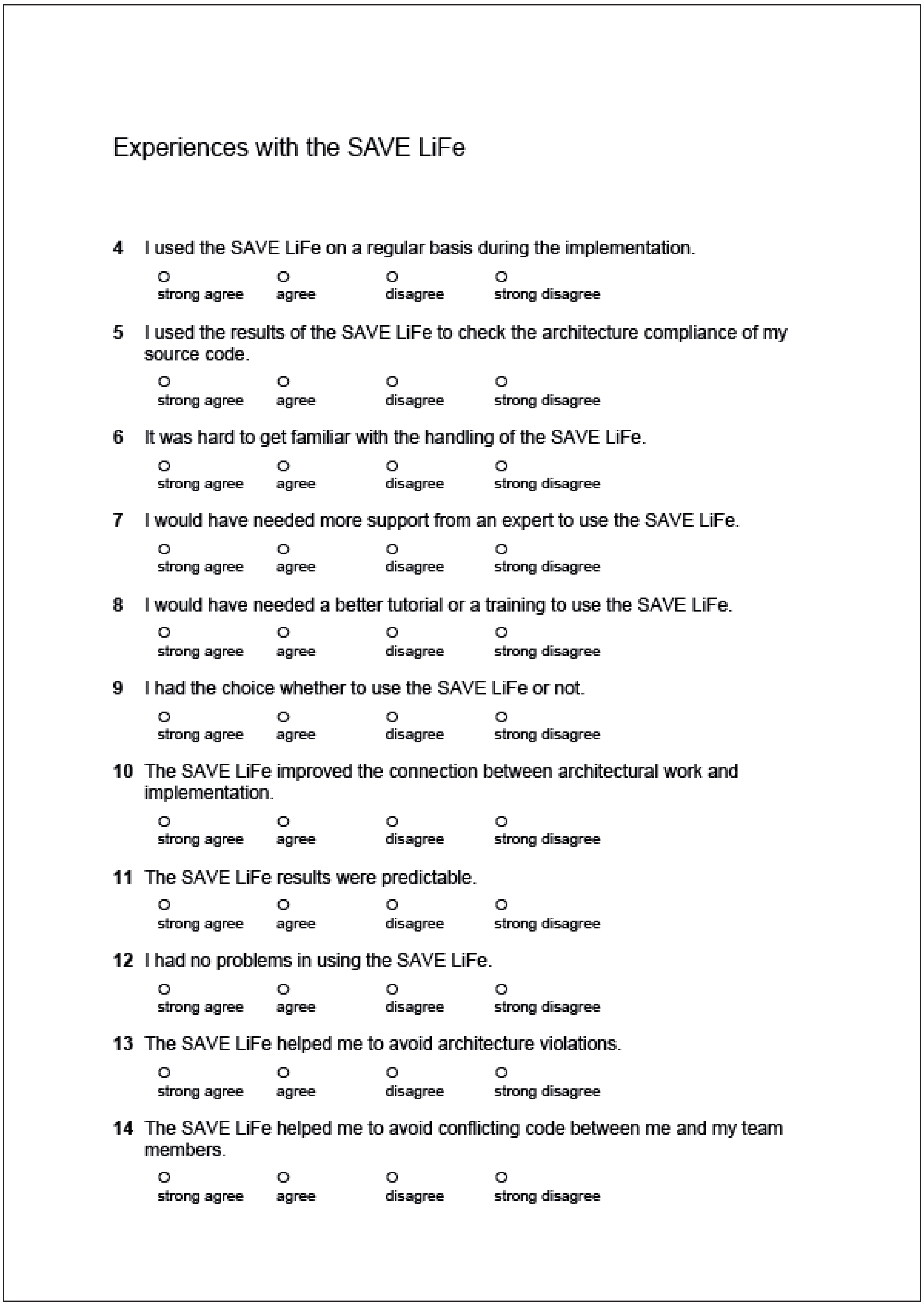
O zero	O 1-2	O 3-5	O more than 5

3 How would you rate your experience in the domain of ambient intelligence?

O very low	O low	O medium	O high	O very high

Figure 72: Experiment Live Feedback: Briefing Questionnaire

E.2 Debriefing Questionnaire



Experiences with the SAVE LiFe

4 I used the SAVE LiFe on a regular basis during the implementation.

○ strong agree ○ agree ○ disagree ○ strong disagree

5 I used the results of the SAVE LiFe to check the architecture compliance of my source code.

○ strong agree ○ agree ○ disagree ○ strong disagree

6 It was hard to get familiar with the handling of the SAVE LiFe.

○ strong agree ○ agree ○ disagree ○ strong disagree

7 I would have needed more support from an expert to use the SAVE LiFe.

○ strong agree ○ agree ○ disagree ○ strong disagree

8 I would have needed a better tutorial or a training to use the SAVE LiFe.

○ strong agree ○ agree ○ disagree ○ strong disagree

9 I had the choice whether to use the SAVE LiFe or not.

○ strong agree ○ agree ○ disagree ○ strong disagree

10 The SAVE LiFe improved the connection between architectural work and implementation.

○ strong agree ○ agree ○ disagree ○ strong disagree

11 The SAVE LiFe results were predictable.

○ strong agree ○ agree ○ disagree ○ strong disagree

12 I had no problems in using the SAVE LiFe.

○ strong agree ○ agree ○ disagree ○ strong disagree

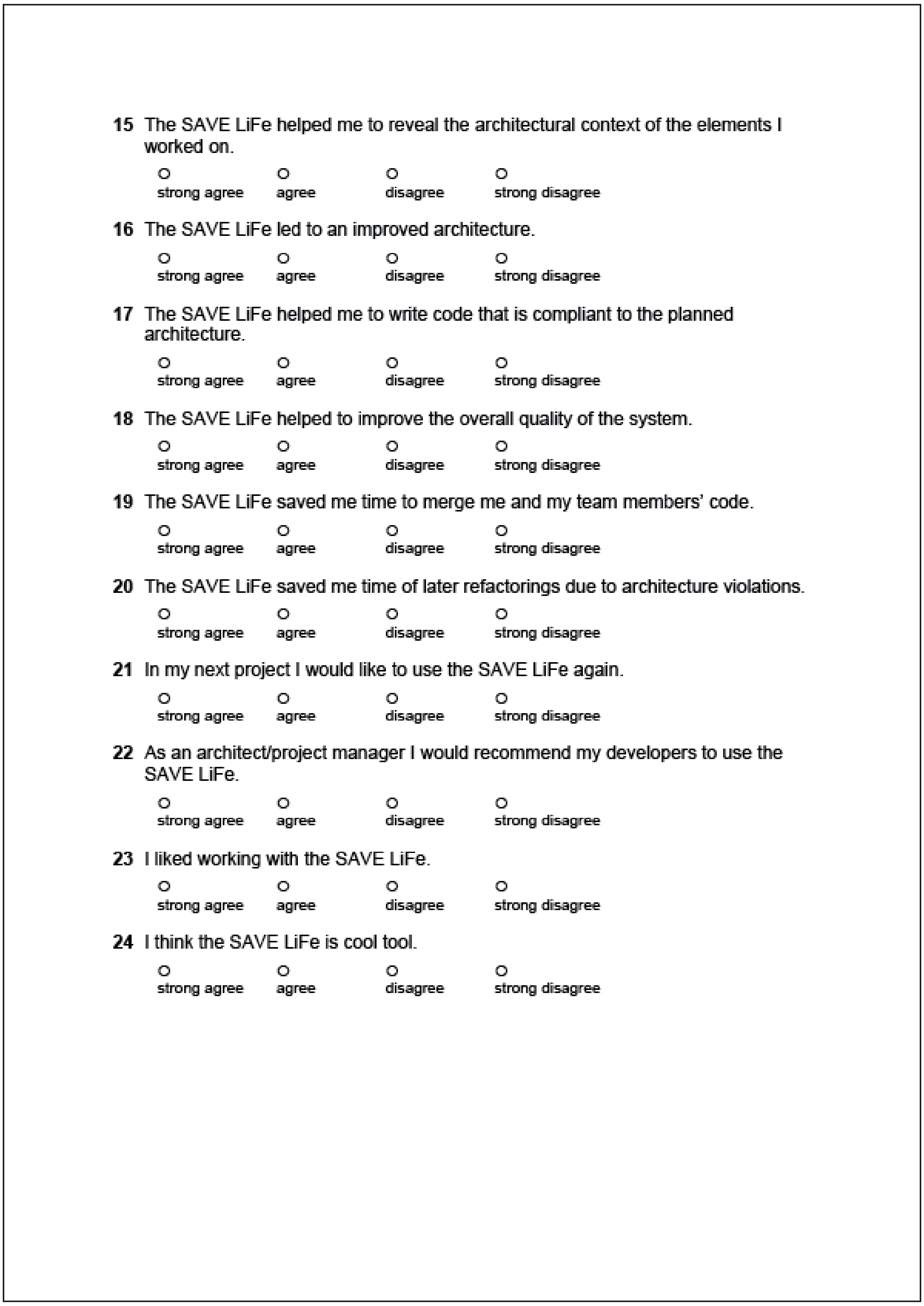
13 The SAVE LiFe helped me to avoid architecture violations.

○ strong agree ○ agree ○ disagree ○ strong disagree

14 The SAVE LiFe helped me to avoid conflicting code between me and my team members.

○ strong agree ○ agree ○ disagree ○ strong disagree

Figure 73: Experiment Live Feedback: Debriefing Questionnaire



15 The SAVE LiFe helped me to reveal the architectural context of the elements I worked on.

○ strong agree ○ agree ○ disagree ○ strong disagree

16 The SAVE LiFe led to an improved architecture.

○ strong agree ○ agree ○ disagree ○ strong disagree

17 The SAVE LiFe helped me to write code that is compliant to the planned architecture.

○ strong agree ○ agree ○ disagree ○ strong disagree

18 The SAVE LiFe helped to improve the overall quality of the system.

○ strong agree ○ agree ○ disagree ○ strong disagree

19 The SAVE LiFe saved me time to merge me and my team members' code.

○ strong agree ○ agree ○ disagree ○ strong disagree

20 The SAVE LiFe saved me time of later refactorings due to architecture violations.

○ strong agree ○ agree ○ disagree ○ strong disagree

21 In my next project I would like to use the SAVE LiFe again.

○ strong agree ○ agree ○ disagree ○ strong disagree

22 As an architect/project manager I would recommend my developers to use the SAVE LiFe.

○ strong agree ○ agree ○ disagree ○ strong disagree

23 I liked working with the SAVE LiFe.

○ strong agree ○ agree ○ disagree ○ strong disagree

24 I think the SAVE LiFe is cool tool.

○ strong agree ○ agree ○ disagree ○ strong disagree

E.3 Results Briefing and Debriefing Questionnaire

Questions	Factor	strong agree	agree	disagree	strong disagree
I used the evolution monitor on a regular basis during the implementation.	Developer Involvement -	1	5	3	2
I used the the results of the evolution monitor to check the architecture compliance of my source code.	Developer Involvement -	1	2	6	2
It was hard to get familiar with the handling of the evolution monitor.	Target Environment - Degree of Novelty	0	1	6	4
I would have needed more support from an expert to use the evolution monitor.	Target Environment - Champion Support	0	6	3	2
I would have needed a better tutorial or a training to use the evolution monitor.	Target Environment - Training	1	6	1	3
I had the choice whether to use the evolution monitor or not.	Perceived Control -	3	1	5	2
The evolution monitor improved the connection between architectural work and implementation.	Perceived Control - Process	1	6	3	1
The evolution monitor results were predictable.	Perceived Control - Predictability	0	2	8	1
I had no problems in using the evolution monitor.	Perceived Characteristics -	2	4	3	2
The evolution monitor helped me to avoid architecture violations.	Perceived Characteristics -	1	3	6	1
The evolution monitor helped me to avoid conflicting code between me and my team members.	Perceived Characteristics -	0	3	6	2
The evolution monitor helped me to reveal the architectural context of the elements I worked on.	Perceived Characteristics -	0	4	5	2
The evolution monitor led to an improved architecture.	Perceived Impacts -	0	3	5	2
The evolution monitor helped me to write code that is compliant to the planned architecture.	Perceived Impacts - Quality	2	4	4	1
The evolution monitor helped to improve the overall quality of the system.	Perceived Impacts - Quality	0	4	5	2
The evolution monitor saved me time to merge mine and my team members' code.	Perceived Impacts - Productivity	0	2	6	3
The evolution monitor saved me time of later refactorings due to architecture violations.	Perceived Impacts - Productivity	0	4	6	1
In my next project I would like to use the evolution monitor again.	Transfer Success - Use	1	4	3	2
As an architect/project manager I would recommend my developers to use the evolution monitor.	Transfer Success - Use	1	6	1	2
I liked working with the evolution monitor.	Transfer Success - Satisfaction	1	4	2	2
I think the evolution monitor is cool tool.	Transfer Success - Satisfaction	3	4	1	2

Figure 74: Experiment Live Feedback: Results Briefing and Debriefing Questionnaire

Lebenslauf

Name	Jens Knodel	
Wohnort	Friedrichstr. 73 67655 Kaiserslautern	
Geburtsdatum	15.11.1976	
Geburtsort	Gifhorn	
Familienstand	verheiratet	
Staatsangehörigkeit	Deutsch	
Schulbildung	1983 – 1987	Grundschule Bückeburg
	1987 – 1989	Orientierungstufe Bückeburg
	1989 – 1996	Gymnasium Adolfinum, Bückeburg Abschluss: Abitur
Studium	1996 – 1997	Fachhochschule Furtwangen Studiengang: Kommunikationsingenieurwesen
	1997 – 2002	Universität Stuttgart Studiengang: Softwaretechnik Abschluss: Diplom
Berufstätigkeit	seit 2002	Wissenschaftlicher Mitarbeiter am Fraunhofer-Institut für Experimentelles Software Engineering (IESE), Kaiserslautern

Kaiserslautern, den 17. November 2009

Volume 17	**Jean-François Girard** (2005), *ADORE-AR: Software Architecture Reconstruction with Partitioning and Clustering*
Volume 18	**Ramin Tavakoli Kolagari** (2006), *Requirements Engineering für Software-Produktlinien eingebetteter, technischer Systeme*
Volume 19	**Dirk Hamann** (2006), *Towards an Integrated Approach for Software Process Improvement: Combining Software Process Assessment and Software Process Modeling*
Volume 20	**Bernd Freimut** (2006), *MAGIC: A Hybrid Modeling Approach for Optimizing Inspection Cost-Effectiveness*
Volume 21	**Mark Müller** (2006), *Analyzing Software Quality Assurance Strategies through Simulation. Development and Empirical Validation of a Simulation Model in an Industrial Software Product Line Organization*
Volume 22	**Holger Diekmann** (2008), *Software Resource Consumption Engineering for Mass Produced Embedded System Families*
Volume 23	**Adam Trendowicz** (2008), *Software Effort Estimation with Well-Founded Causal Models*
Volume 24	**Jens Heidrich** (2008), *Goal-oriented Quantitative Software Project Control*
Volume 25	**Alexis Ocampo** (2008), *The REMIS Approach to Rationale-based Support for Process Model Evolution*
Volume 26	**Marcus Trapp** (2008), *Generating User Interfaces for Ambient Intelligence Systems; Introducing Client Types as Adaptation Factor*
Volume 27	**Christian Denger** (2009), *SafeSpection – A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts*
Volume 28	**Andreas Jedlitschka** (2009), *An Empirical Model of Software Managers' Information Needs for Software Engineering Technology Selection* *A Framework to Support Experimentally-based Software Engineering Technology Selection*
Volume 29	**Eric Ras** (2009), *Learning Spaces: Automatic Context-Aware Enrichment of Software Engineering Experience*
Volume 30	**Isabel John** (2009), *Pattern-based Documentation Analysis for Software Product Lines*
Volume 31	**Martín Soto** (2009), *The DeltaProcess Approach to Systematic Software Process Change Management*
Volume 32	**Ove Armbrust** (2010), *The SCOPE Approach for Scoping Software Processes*

Software Engineering has become one of the major foci of Computer Science research in Kaiserslautern, Germany. Both the University of Kaiserslautern's Computer Science Department and the Fraunhofer Institute for Experimental Software Engineering (IESE) conduct research that subscribes to the development of complex software applications based on engineering principles. This requires system and process models for managing complexity, methods and techniques for ensuring product and process quality, and scalable formal methods for modeling and simulating system behavior. To understand the potential and limitations of these technologies, experiments need to be conducted for quantitative and qualitative evaluation and improvement. This line of software engineering research, which is based on the experimental scientific paradigm, is referred to as 'Experimental Software Engineering'.
In this series, we publish PhD theses from the Fraunhofer Institute for Experimental Software Engineering (IESE) and from the Software Engineering Research Groups of the Computer Science Department at the University of Kaiserslautern. PhD theses that originate elsewhere can be included, if accepted by the Editorial Board.

ISBN 978-3-8396-0233-1

9 783839 602331

Fraunhofer
IESE

Technische Universität
Kaiserslautern
AG Software Engineering

PhD Theses in Experimental Software Engineering