Instruments to Evaluate and Improve IT Architecture Work

Leo Pruijt

ISBN: 978-94-6233-142-6

Instruments to Evaluate and Improve IT Architecture Work

Instrumenten ter Evaluatie en Verbetering
van IT-Architectuur Werk
(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Utrecht op gezag van
de rector magnificus, prof.dr. G.J. van der Zwaan,
ingevolge het besluit van het college
voor promoties in het openbaar te verdedigen op
woensdag 25 november 2015 des middags te 12.45 uur
door

Leo Jacobus Pruijt

geboren op 6 juli 1961
te Gouda

Promotor:	Prof.dr. S. Brinkkemper
Copromotoren:	Dr.ir. Raymond Slot MBA
	Dr.ir. J.M.E.M. van der Werf

Dit proefschrift werd mede mogelijk gemaakt door Hogeschool Utrecht.

Preface

About five years ago, I began to consider the possibilities of a PhD study. In the preceding years, my employer, the HU University of Applied Sciences Utrecht, had begun to promote and facilitate PhD research, but I had declined invitations, knowing that it would change the balance in my life considerably. However, several demanding though satisfying activities made me change my mind in the second half of 2010. First, I had worked with pleasure on a series of papers on layered software architectures for a magazine for professional software architects; written in cooperation with Wiebe Wiersema, a lector of the Information Systems Architecture Research Group (ADIS).

Second, Raymond Slot, another lector of ADIS, involved me in the ArchiValue project, a research collaboration of several academic institutions and professional organizations. Within the context of this project, the Enterprise Architecture Realization Scorecard (EARS) was developed and we were planning the first case study on the application of the instrument.

Last, the project within the third year specialization semester Advanced Software Engineering (ASE), which I coordinated, provided the opportunity to combine research, education, and software development. In the preceding years, we had formed a good-working team of lecturers, and quite some students that followed the specialization had proven to be keen and motivated. Furthermore, several professional organizations were willing to cooperate.

Being at the end of my PhD study, I can confirm that the above mentioned activities have provided a solid base for my research. In the last four years, we were able to extend and improve our work on the EARS instrument and on software layers, and to raise it to the academic level. Furthermore, the projects of the ASE specialization have yielded working prototypes, application case data, and initial tool tests with respect to software architecture compliance checking. The planning and elaboration of these projects in 2011, 2012, and 2013 required a huge effort, but the results were amazing, and the collaboration with and between the students, lecturers, and professionals in the participating organizations was excellent.

At present, I realize that five years ago I did not know where I was starting on. The PhD journey felt initially like starting a new job in another field, with another language, a different knowledge base, and other procedures and ethics. However, I consider these years as enriching years, in which I have learned a lot, stressed a lot, though also achieved a lot. Academic work is like monkish work. It requires patience, perseverance, and humility; mostly trained by acceptance (mostly after a cooling period) of the remarks and rejections of reviewers higher in the academic hierarchy.

Many people have contributed to my research and achievements, in one way or another. Without these contributions, I could not have accomplished the work presented in this dissertation. Therefore, I am grateful to all of them. However, it is impossible to include all names, since there are simply to many: colleagues of the Information Systems Research Group of the HU university of Applied Sciences Utrecht; members of focus groups on enterprise architecture and software architecture at Utrecht University; students of the specialization Advanced Software Engineering; colleagues teaching and coaching these students; professionals who provided feedback in the design stages of the instruments; professionals and their organizations participating in case studies on the application of the instruments; lecturers of courses on design science research and academic writing, which I followed; and finally, all the reviewers of my workshop, conference, and journal papers.

Special thanks go to my promoter, Sjaak Brinkkemper, and co-promoters, Raymond Slot and Jan Martijn van der Werf. Thank you for the guidance, encouragement, and feedback. You taught me the new job! Furthermore, I would like to thank the members of the reading committee, Professors Paris Avgeriou, Arie van Deursen, Erik Proper, Andreas Rausch, and Uwe Zdun, for their time and effort in reviewing and judging my dissertation.

In addition, I owe special thanks to several members of ADIS who co-authored my papers. Christian Köppe acted as a great, reliable partner in my research on software architecture, while Henk Plessius and I collaborated in research on the effectiveness and benefits of enterprise architecture. Finally, Wiebe Wiersema has inspired and mentored me since the beginning of our research group, both as lector and as professional partner.

Last, but not least, I am grateful to my dear wife, Birgitta, for her support and encouragement, and to our children, Gabe and Imme. You allowed me to work for many hours in “family time”, and you did not act too harsh on me when my mind had, again, not followed my body from my working desk to the dinner table.

Leo Pruijt, September 2015

Contents

Chapter 1

Introduction

This dissertation focuses on architecture work in the field of Information Technology (IT) and on instruments that may be used to evaluate, improve, and support IT architecture work. In this chapter, we first introduce the research domain and the problems that challenged us to initiate research. Second, we present the research questions, the starting points for research lines that address the challenges. Third, we provide an overview of our research approach and the used research methods. Finally, we explain the structure of the dissertation and how the following chapters are related to the research questions.

1.1 Architecture in the Domain of IT

In the last decades, architecture has emerged as a discipline in the scientific domain of IT. In the late 1980's, increased size and complexity of information systems made it necessary to use some logical constructs (or architecture) for defining and controlling the integration of all the components of a system (Zachman 1987). Since then, the discipline of architecture has evolved enormously in practice and research (Simon et al. 2013, Shaw and Clements 2006).

A well-accepted, generic definition of architecture is from ISO 42010: "The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution" (ISO 2007).

IT architecture work is demanding and challenging, and includes, inter alia, defining architecture goals and vision, identifying architectural significant requirements (functional and non-functional), designing and selecting solutions for these requirements, and ensuring that the solutions are implemented according to the architectural design.

1.1.1 Types of Architectures

Since the publication of "A framework of information systems architecture" (Zachman 1987), many types of architectures are distinguished in the domain of IT, and currently a variety of labels and definitions is used. IT Architecture is one of these term that lacks a universally accepted definition (Ross 2003). We use the

term in this dissertation as a broad term, to cover all types of architecture used in the domain of IT.

Five types of architecture that are relevant to our research are shown in Figure 1.1. The types of architecture, often distinguished in enterprise architecture frameworks, are structured as hierarchical layers, where each architecture layer reduces the degrees of freedom of the subsequent layers (Winter and Fischer 2007). Although each type of architecture has a different focus and represents different architecture products, consistency between these layers and products should be maintained. The focus of each type of architecture is as follows (Winter and Fischer 2007):

- Business architecture "represents the fundamental organization of the corporation (or government agency) from a business strategy viewpoint."
- Process architecture "represents the fundamental organization of service development, service creation, and service distribution in the relevant enterprise context."
- Integration architecture "represents the fundamental organization of information system components in the relevant enterprise context."
- Software architecture "represents the fundamental organization of software artifacts, e.g. software services and data structures."
- Technology (or infrastructure) architecture "represents the fundamental organization of computing / telecommunications hardware and networks."

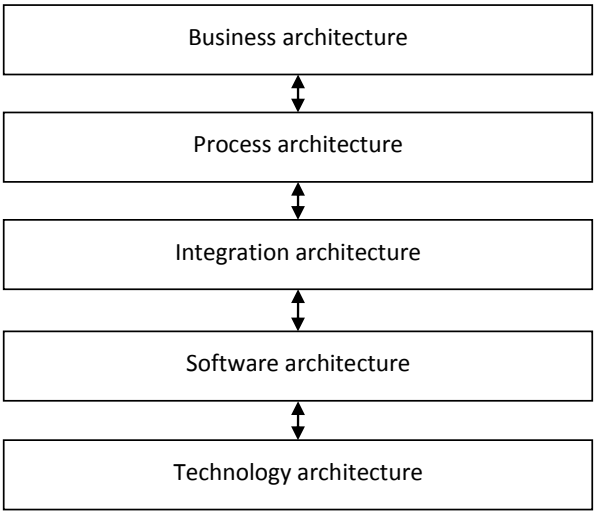

Figure 1.1: Five types of architecture in enterprise architecture (after Winter and Fischer 2007)

We have focused our work on enterprise architecture and on software architecture, which both will be introduced below. Although we regard both types to represent different domains and disciplines, software architecture is tightly related to the other types of architectures, as shown in Figure 1.1.

1.1.2 Enterprise Architecture

The term Enterprise Architecture (EA) is defined in various ways by different authors. A definition of an acknowledged author is the following: *enterprise architecture* is "a coherent whole of principles, methods, and models that are used in the design and realization of an enterprise's organizational structure, business processes, information systems, and infrastructure" (Lankhorst et al. 2009).

Currently, enterprise architecture is widely accepted in practice, especially in larger organizations (Bucher et al. 2006, Obitz and Babu 2009), and a considerable number of scientific studies have focused on EA, especially in the last decade (Simon et al. 2013). A diversity of EA management approaches has been proposed by academia and practitioners (Winter et al. 2010), like GERAM (FAC-IFIP Task Force 1999), DoDAF (Department of Defense 2009), and TOGAF (The Open Group 2009). The Open Group Architecture Framework (TOGAF) is the general EA framework that gained the widest adoption in practice (Obitz and Babu 2009) and research (Simon et al. 2013). TOGAF provides methods and tools for assisting in the production, acceptance, use, and maintenance of an EA (The Open Group 2009).

An overview of typical sub-architectures of enterprise architecture, according to TOGAF, is provided in Figure 1.2. Comparison of Figure 1 and 2 shows that

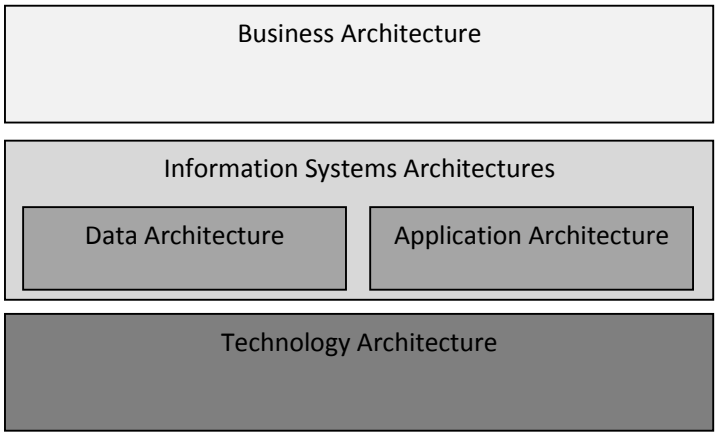

Figure 1.2: EA sub-architectures according to TOGAF (The Open Group 2009)

currently there is no general accepted terminology for types of architecture, and that only four of the five types of architecture from Figure 1.1 are present in Figure 2. The mapping from the types of architectures in Winter and Fischer's layered model to TOGAF's EA sub-architectures is as follows: Business architecture and Process architecture map to TOGAF's Business Architecture; Integration architecture maps to Information Systems Architecture; and Technology architecture maps to Technology Architecture.

In the more recent versions of TOGAF, software architecture is excluded from enterprise architecture and positioned as part of solution architecture. In contrast to EA, solution architecture typically focuses on a single project or project release. The difference between the work of an enterprise architect and solution architect is summarized as follows (The Open Group 2009):

- "The Enterprise Architect has the responsibility for architectural design and documentation at a landscape and technical reference model level."
- "The Solution Architect has the responsibility for architectural design and documentation at a system or subsystem level."

Enterprise Architecture Management Function

Over the last decades, Enterprise Architecture Management (EAM) is introduced in many large organizations. An EAM function forms a means to enhance the alignment of business and IT and to support the managed evolution of the enterprise (Buckl et al. 2009). In other words, the objective of EAM is to translate the broader goals and principles of an organization's strategy into concrete processes and IT systems, thereby enabling the organization to realize their goals and bridging business strategy formulation and the actual implementation of this strategy (Lange et al. 2012). To realize the organization's goals, an architecture development and realization process is essential to the EAM function. The main activities of an EA realization process with their results, as derived from the Architecture Development Method of TOGAF 9 (The Open Group 2009), are depicted in Figure 1.3 and are defined below.

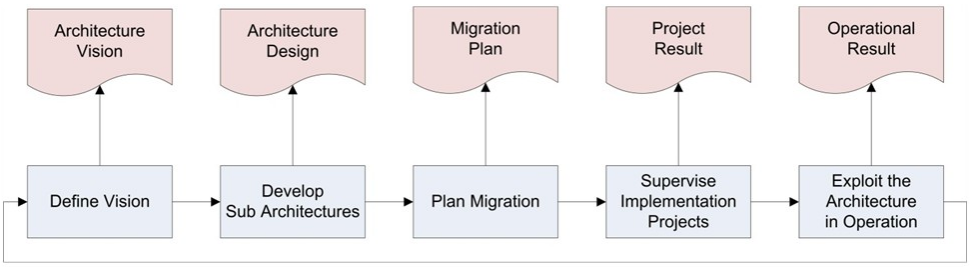

Figure 1.3: Activities and results of an effective EA realization process

- Define Vision: Determine the EA goals within scope of the architecture iteration, develop a high level, integrated and approved solution direction towards matching these goals, and create a concise plan to realize them. As shown in the figure by the feedback loop, an iterative approach may be used to develop the architecture.
- Develop Sub Architectures: Develop the required subsets of architectures to support the agreed architecture vision.
- Plan Migration: Search for opportunities to implement the architecture and plan the migration.
- Supervise Implementation Projects: Ensure conformance to the architecture during the development and implementation projects.
- Exploit the Architecture in Operation: Assess the performance of the architecture in operation, ensure optimal use of the architecture, and ensure continuous fit for purpose.

Our research focuses on the effectiveness of the EA realization process, since we consider an effective process conditional for an EAM function to add value to its organization.

1.1.3 Software Architecture

Software architecture, as a kind of solution architecture, takes care of the internal organization of an application; a single application in the enterprise architecture's application architecture, according to TOGAF (The Open Group 2009). A definition of Software Architecture (SA), often used within the software architecture research community, is the following: *software architecture* "provides the framework within which to satisfy the system requirements and provides both the technical and managerial basis for the design and implementation of the system" (Perry and Wolf 1992). This definition clearly states the objectives of software architecture. Another definition highlights the structural perspective (dominant in the ISO 42010 definition of architecture): "Software architecture involves: a) the structure and organization by which modern system components and subsystems interact to form systems; and b) the properties of systems that can best be designed and analyzed at the system level" (Kruchten et al. 2006b).

Compared to basic analysis and design activities, where the focus is primarily on the identification and design of individual functional requirements, software architecture adds a system wide perspective and focuses on non-functional requirements, like modifiability, reliability, and security. Since the late 1980's, software architecture has enjoyed a golden age of innovation and concept formulation, and it is beginning to enter the more mature stage of quiet discipline and unremarkable utilization (Clements and Shaw 2009).

To organize architecture models and documentation, different views are recognized such as logical view and deployment view, where a view is defined as "a representation of a set of system elements and relationships among them" (Clements et al. 2010). Many approaches that involve multiple views have been proposed, like the 4+1 view model (Kruchten 1995), Rozanski and Woods set of viewpoints (2005), and the views and beyond approach of the Software Engineering Institute (SEI) (Clements et al. 2010). Since the view and beyond approach extensively describes the views that are relevant to our work, we have based our terminology on SEI's work. The views and beyond approach distinguishes three categories of views (or viewtypes), which are shown in Figure 1.4 with their views.

The three viewtypes focus on different types of elements:

- Module: models software's implementation units and their relationships;
- Component and connector: models elements that have some run-time presence;
- Allocation: models software elements and their relationships to environmental elements.

Module Viewtype

Our research does not cover the full width of software architecture, but focuses on what we have labeled the *modular architecture*, corresponding with the views of the Module viewtype. As a small, introducing example, we present small-scale architecture diagrams from a student's project: a simple card game, namely Thirty-

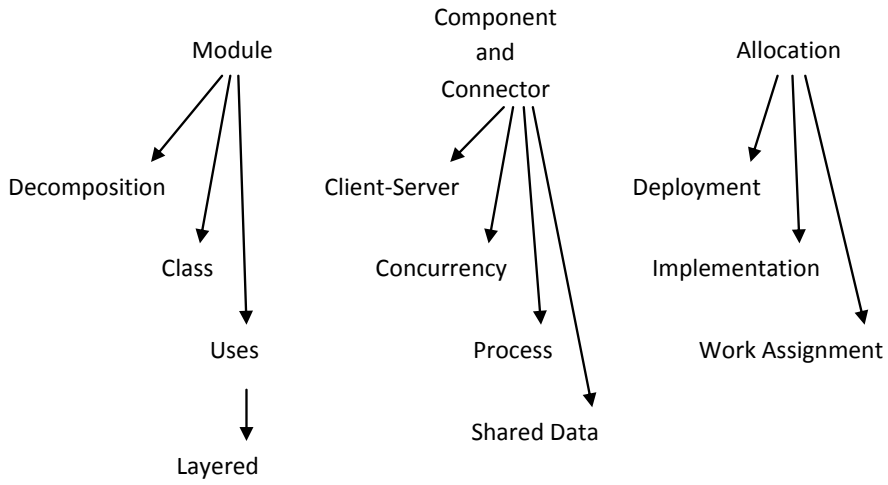

Figure 1.4: Viewtypes with their views (Clements et al. 2010)

One. A conceptual model of the modular architecture is shown in Figure 1.5. Seven modules of two types are included, namely two layers and five subsystems. Furthermore, usage relations (the dashed arrows) show the allowed usages. All other usages between the modules are prohibited. This module model is an example of an *intended architecture*, which is the outcome of the architecture design process (de Silva and Balasubramaniam 2012).

The design of a modular architecture may answer non-functional requirements, like maintainability, reusability, and portability (Parnas 1972). This requires a well-designed intended architecture and a complying *implemented architecture*, which is the model that has been realized or built-in in low-level design constructs and the source code (de Silva and Balasubramaniam 2012).

Architecture Compliance Checking

Architecture Compliance Checking (ACC) is an approach to assess whether the implemented architecture complies with the intended architecture, in order to identify potential erosion problems (de Silva and Balasubramaniam 2012). ACC techniques and tools aid the software architect's task to monitor that the solution is implemented according to the architectural design.

As an example of the result of an ACC, Figure 1.6 provides an implemented architecture diagram of the Thirty-One game code at root level. The model shows that the root of the source code contains a class and two packages. The black, dashed arrows represent usage relations, while the numbers indicate the numbers of

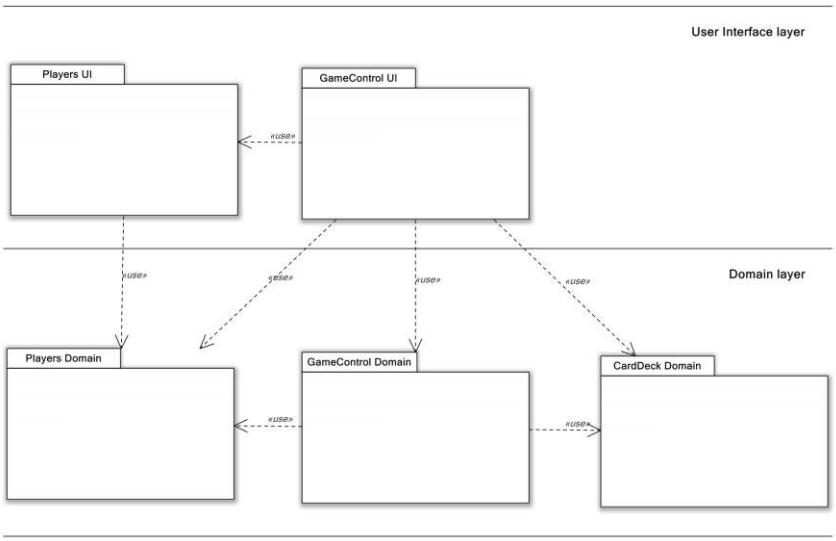

Figure 1.5: Model of an intended modular architecture of game Thirty-One

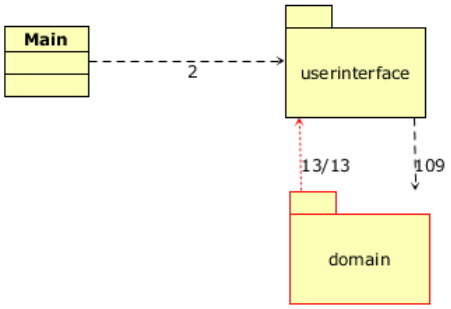

Figure 1.6: Model of an implemented architecture of game Thirty-One

dependencies as detected by HUSACCT (Pruijt et al. 2014), an ACC tool developed in the course of our research. An architectural inconsistency in usage style is visible in the figure as a red, dotted usage relation. The classes in package domain contain thirteen dependencies on classes in package userinterface, while the intended modular architecture in Figure 1.5 prohibits these usages for two reasons: 1) a lower level layer should not use a higher-level layer; and 2) no usage relation in Figure 1.5 justifies the usage of package userinterface by domain.

Semantically Rich Modular Architecture

In literature and practice, many modular architecture designs contain module types with different semantics, such as subsystems, layers, and components. In addition, different types of rules exist. For example, rule types that restrict the usage of a module, or rule types that restrict the naming or visibility of elements of a module. For some types of modules, one or more related rules apply. For instance, in case of module type layer, usage of higher level layers is forbidden, and in case of a strict layered model (Fowler et al. 2003), only usage of a directly lower level layer is allowed.

We introduce the term *semantically rich modular architecture* (SRMA) for an expressive modular architecture description, composed of semantically different types of modules, which are constrained by different types of rules. In our opinion, modules with specific semantics enhance the expressiveness of a modular architecture and support architecture reasoning. As an example of an SRMA, the intended architecture of the core of an E-commerce system is shown in Figure 1.7. The system is developed in C# and its architecture is based on the .NET common application architecture (MSDN 2009). Five different types of modules are present: layers (UML packages with stereotype << Layer>>), subsystems (UML packages), components (UML component), interfaces (UML interface), and external systems (UML packages System.Data and CommerceServer). Furthermore, 17 rules of eight different types are defined in this case. Appendix 1 describes the intended

architecture of this case, the implemented architecture and the results of a compliance check in detail.

SRMA Support and ACC

Adersberger and Philippsen (2011) consider the support of semantically rich architecture models essential for the integration of ACC in model-driven engineering. Furthermore, they make clear that support of semantically rich constructs reduce the number of rules that need to be specified. In practice and literature, many architecture models can be labeled as SRMA, since they contain modules of different types. Consequently, ACC-tools should provide support for SRMAs to minimize the gap between the intended architecture as designed and the architecture as represented in the ACC-tool.

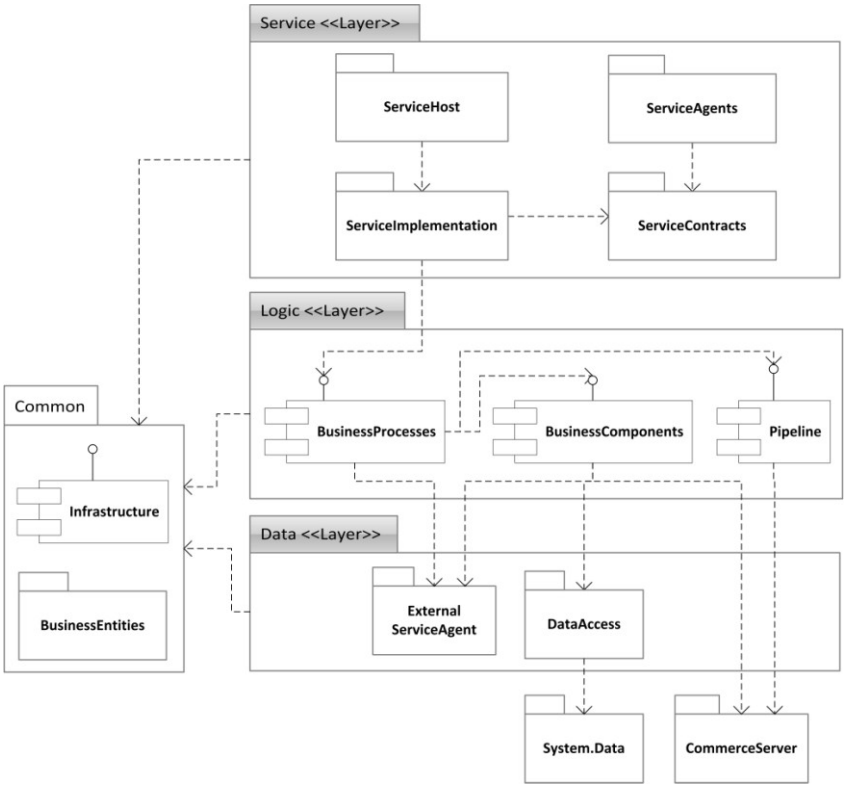

Figure 1.7: Intended architecture of the core of an E-commerce system

1.2 Challenges in the Domain of Architecture

Although architecture has been introduced in many IT departments, practicing architecture does not always proceed without problems. With our work, we intend to contribute to the advancement of architecture in the domain of IT and the effectiveness of architecture work by means of the development and improvement of supporting instruments and tools. In the context of ongoing research on architecture at HU University of Applied Sciences and Utrecht University, we have scoped our work to the domains of enterprise architecture and software architecture, and within these domains to a limited number of challenges. These challenges and our motivation to take up these challenges are described below.

1.2.1 Challenges in the Domain of Enterprise Architecture

Enterprise architecture management (EAM) has become increasingly recognized as a crucial part of both business and IT management, but there is still some way to go (Simon et al. 2013). EAM functions are, in general, still relatively immature (Bucher et al. 2006). Moreover, many EAM initiatives are confronted with substantial challenges and often fail (Löhe and Legner 2014).

Evidence of the immaturity of EAM functions has come from research on the state-of-the-art of EAM in practice, like the following findings:

- Most EAM functions, score low in maturity on "Development of architecture", "Use of architecture", and "Monitoring" (of the implementation activities) (van Steenbergen et al. 2010).
- The performance of the EAM function is rarely measured, even though this constitutes a valuable starting point for improvement (Winter et al. 2010).

The study "An Exploration of Enterprise Architecture Research" (Simon et al. 2013) shows that, up to 2012, the following three streams were dominating in EAM research: "EA frameworks", "Design & operations of EA management", and "EA conception & modeling". For future work, the study suggests greater emphasis on the following research areas: 1) business architecture management (at the strategic level); 2) standards management; 3) the integration with operational architecture management; and 4) EA lifecycle phases beyond documentation. The authors argue: "A shift of focus to EA processes and organization may provide the necessary basis and also facilitate giving more pragmatic advice, needed to attract practitioners".

Our research in the domain of enterprise architecture has focused on assessment of the effectiveness of the EA realization process of EAM functions. As such, it builds upon the above-cited challenges found by van Steenbergen et al. and Winter

et al. In addition, our research fits in the fourth research area mentioned for emphasis in future research, in the above-cited study of Simons et al.

1.2.2 Challenges in the Domain of Software Architecture

In the comprehensive survey "The Golden Age of Software Architecture", Shaw and Clements (2006) make clear that the discipline software architecture has grown phenomenally since the late 1980's. They also identified six areas where significant opportunities for new contributions in software architecture were possible. Three years later, in "The Golden Age of Software Architecture Revisited" (Clements and Shaw 2009), the three research areas below were repeated as opportunities with strong potential to make real improvements.

1. Object-oriented programming versus architecture
2. Design decisions and quality attributes
3. Conformance checking and architecture recovery

Our work in the domain of software architecture has focused on two research areas, of which the challenges are described below in more detail. The first one, architecture compliance checking, is clearly linked to Clements and Shaw's third research opportunity in the list above. The second one, quality of layered designs, fits well within the research area of Clements and Shaw's second opportunity.

In our work in practice and in education, we have frequently observed problems related to these two research areas. Our observations motivated us for research in these areas, and even more, when we found them confirmed in literature.

Architecture Compliance Checking

Problems with architecture compliance are not new, as is illustrated by the following statement, noted two decades ago: "high level models are almost always inaccurate with respect to the system's source code" (Murphy et al. 1995). The terms architecture conformance and architecture consistency are also often used in literature. We prefer the term architecture compliance, since in ACC it best expresses the objective of the activity: check if the implementation follows the design. Although, the results of the check may be used as well to alter the design as to adjust the implementation.

In 2006, architecture conformance was included in the list of six promising research areas (Shaw and Clements 2006). Three years later, in "The Golden Age of Software Architecture Revisited" (Clements and Shaw 2009), it was repeated as one of only three opportunities with strong potential to make real improvements. A part of the motivation in the last paper was described as follows: "Tools to analyze code for architecture conformance are still woefully inadequate and rely on humans

making suggestions (read: guesses) about architectural constructs that might be lurking in the code."

More recently, in a survey of techniques and technologies to control software architecture erosion (de Silva and Balasubramaniam 2012), architecture erosion is defined as "the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture". Two considerations from the study are of interest here: 1) "Approaches that link architectural models to implementation possibly make the strongest claim for constraining architecture erosion"; and 2) "However, despite the availability of a large number of dependency/static analysis tools, recent industry surveys indicate that these are not extensively used in projects for checking architectural conformance at code level."

In summary, research on ACC is needed to solve the following problems:

- The implemented architecture deviates frequently from the intended architecture.
- Supporting tools are still inadequate.
- The adoption of these tools in practice is low.

Quality of Layered Designs

One of the most common patterns used in the module view of software architecture, is the Layers pattern, or Layered style (Clements et al. 2010, Harrison and Avgeriou 2008). The use of layers to decouple the system's components in a vertical manner is crucial in order to support modifiability, portability, and reusability (Avgeriou and Zdun 2005). However, the layered view of architecture "is also poorly defined and often misunderstood" (Clements et al. 2010). The authors motivate this phenomenon as follows: "Because true layered systems have good properties of modifiability and portability, architects have incentive to show their systems as layered, even if they are not."

In line with the cited problem, we have frequently observed that many layered architectures are poorly designed and documented. In many cases, only the layers are listed. These cases miss a specification of the responsibilities per layer, the communication rules between the layers, and a justification. Such architectural products are very incomplete and provide little guidance to the developers.

Our concerns focus especially on the assignment of responsibilities, since we have found numerous cases in practice and education, where a certain type of responsibility was not only implemented in the intended layer, but also in other layers. Some striking examples of this problem are the following:

- User interface responsibility (e.g., with access to UI-libraries) in the domain layer, or in a layer intended for control responsibility only;
- Use case specific control functionality in the domain layer;
- Domain responsibilities in the presentation layer;

- Data access responsibilities in presentation and domain layers.

We concluded that research is needed that aims on qualitative improvement of layered designs, with the focus on responsibility assignment.

1.3 Research Questions

In line with our intentions, as described above, the main research question of this thesis is:

How can IT architecture work be evaluated and improved?

In the first place, the main question above provides an impression of our intention. The term evaluate should be interpreted as a broad term, covering similar terms in this dissertation, like assessment, check, or review. It is our aim to contribute the solution of the problems described in the previous section. Furthermore, the main question is wide enough to cover both enterprise architecture and software architecture. However, the main question embraces such a large field of research that further scoping is needed. As explained in the previous sections, we have delimited our research to three research areas. Each area relates to a major research questions below.

Since most of our work has focused on modular architectures in the domain of software architecture, we first discuss the questions related to software architecture. This order is also followed in the remainder of this dissertation.

1.3.1 Research Questions Software Architecture

RQ1: How can architecture compliance be evaluated and improved?
Based on the notion that monitoring architecture compliance requires tool support, while the adoption of Architecture Compliance Checking (ACC) supporting tools is still limited, we have conducted research on ACC support. We started from a functional point of view, by identifying requirements regarding ACC support and by studying existing static ACC tools. Next, we scoped our research to the following sub-questions:

RQ1.1: Do static ACC-tools provide functional support for semantically rich modular architectures?
In literature and practice, many modular architecture designs contain different types of modules, with different semantics, and different types of rules. Therefore, an important requirement in our ACC research concerns the support of semantically rich modular architectures (SRMAs).

RQ1.2: How can SRMA support be provided in the context of static ACC?
Since the research results of RQ1.1 showed that only limited ACC-tool support was available for semantically rich modular architectures, we concluded that research was needed to bridge the gap between modular architectures in software architecture designs on one side, and module and rule models in ACC-tools on the other side.

RQ1.3: How accurate do static ACC-tools report dependencies and violations against dependency rules?
Static ACC focuses on the existence of rule violating dependencies between modules. Because of the high number of dependencies at implementation level, accurate tool support is essential for effective and efficient ACC. However, the effectiveness of ACC is profoundly dependent on the ability of the used ACC-tool to detect the dependencies between units in the implemented software and to report violating dependencies.

RQ2: How can the quality of layered designs with respect to the assignment of responsibilities be evaluated and improved?
Layers are commonly used in modular architectures, but many layered architectures are poorly designed and documented. In this dissertation, we scoped our research in this area to supporting instruments with respect to the assignment of responsibilities to software layers. In the design of a layered model, the assignment of responsibilities is an essential step. Based on the allocated responsibilities, architectural rules are defined to restrict dependencies between the layers. Furthermore, software developers need to decide, based on the responsibilities of the layers, where specific functionality has to be implemented, and where not.

1.3.2 Research Question Enterprise Architecture

RQ3: How can the achievement of an EAM function with respect to the realization of its goals be evaluated and improved?
To add value to the organization, an Enterprise Architecture Management (EAM) function should be able to realize its goals in line with the corporate strategy. The EAM function works effectively, when it is able to transform a given baseline situation into a target situation, as specified by one or more goals set out to the EAM function. Since no instrument was available to assess and enhance the effectiveness of EAM functions conform this definition of effectiveness, research was aimed on the design of an instrument to discover the strengths and weaknesses in the realization process of an EAM function.

1.4 Lines of Research

Three lines of research are visible in our work, which relate to the three major research questions.

1.4.1 Architecture Compliance Checking Support

RQ1 has initiated a research line with the focus on the support of static architecture compliance checking, which line we have labeled as *Architecture Compliance Checking Support* (ACCS). In our view, ACC does not only enable architectural consistency, but by linking model to code, it also promotes architecture awareness. Furthermore, ACC might also be useful to improve the quality of intended modular architectures, based on the insights gained at implementation level. Since efficient ACC requires tool support, we have initiated a line of research on the effectiveness of the support provided by static ACC-tools. Static ACC tools analyze software without executing the code and focus on modular architectures.

1.4.2 Layered Architecture Design Support

RQ2 is addressed in a research line, which we have labeled as *Layered Architecture Design Support* (LADS). With the aim to advance the practice and education regarding layered architectures, we have initiated research on analysis of the observed problems and on instruments to design layered architectures of high quality. In this dissertation, we focus on the assignment of responsibilities to software layers. Outside the scope of this dissertation, we have been working on a fundamental approach to design logical and physical layered designs.

1.4.3 Enterprise Architecture Realization Assessment

RQ3 is addressed in a research line focusing on the ability of an enterprise architecture management function to realize its goals. We have labeled this line of research as *Enterprise Architecture Realization Assessment* (EARA). To contribute to the further development of the EAM practice, we have participated in research, which was instigated as part of a larger study on the value of enterprise architecture, sponsored by the Dutch Government and three profit organizations. We started from the notion that to add value to an organization, an EAM function should be able to realize its goals. Accordingly, an effective EA realization process is essential. Based on these notions, we initiated research on the design of an assessment instrument aimed on the improvement of an EAM function's ability to realize its goals.

1.5 Research Approach and Research Methods

1.5.1 Research Approach

The research approach of the work presented in this dissertation can best be characterized as design-science research. As explained in the previous paragraphs, we have based our work on problems identified in practice, and we have tried to contribute to the solution of these problems by the design of new instruments and the improvement of existing types of instruments.

The Information Systems Framework (Hevner et al. 2004), shown in Figure 1.8, provides an overview of Information Systems (IS) research in its context. The authors use the framework to explain that behavioral-science and design-science paradigms need to be combined in IS research. Furthermore, they use the framework to explain their guidelines concerning effective design-science research. The following guidelines are provided: 1) Design as an artifact; 2) Problem relevance; 3) Design evaluation; 4) Research contributions; 5) Research rigor; 6) Design as a search process; 7) Communication of research.

We also found guidance in the design-science research methodology (DSRM) process model (Peffers et al. 2008), which distinguishes the following activities:

1. Problem identification and motivation
2. Define the objectives of a solution
3. Design and development
4. Demonstration

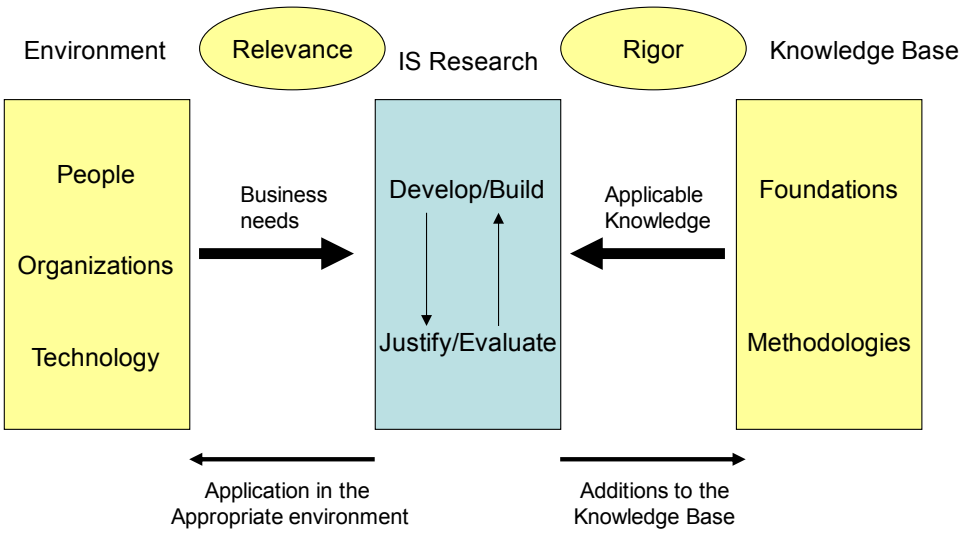

Figure 1.8: The IS research framework (Hevner et al. 2004).

5. Evaluation
6. Communication

In each line of research, we have pursued the process and guidelines described above, although in a more iterative way.

First, we identified problems in the practice of IT architecture, as described in the previous sections (Peffers activity 1, Hevner guideline 2).

Second, we identified requirements for the solution (Peffers activity 2, Hevner guideline 6), whereby we made use of the knowledge base in the form of scientific and professional literature (Hevner guideline 5). In several cases, this activity has delivered knowledge results that could be communicated afterwards, like a classification of common module and rule types, and a classification of dependency types.

Third, we designed and developed a solution to the given problem, based on the requirements from the previous step (Peffers activity 3). This approach resulted for each line of research in deliveries of constructs, a model, and a method, or an instantiation (Hevner guideline 1). Where needed, we made use of academic and professional literature, or we involved experts to find optimal solutions. During the design process, we have often made use of metamodeling techniques. These metamodels were important to identify key concepts, and to define, simulate and communicate functional mechanisms (Hevner guideline 6). In addition, the metamodels proved to be significant artifacts, which also could be communicated afterwards.

Fourth, to demonstrate the applicability and validate the solutions, the designed instruments were applied in real situations in professional organizations (Peffers activity 4, Hevner guideline 3). In several cases, laboratory experiments were conducted as well.

Fifth, evaluations of the solutions were conducted in several forms such as comparison of performance measures to the requirements, client feedback, and expert feedback (Peffers activity 5, Hevner guideline 3).

Sixth, the results of our research were communicated by means of conference and journal papers, presentations, web sites and videos, and as such were added to the knowledge base (Peffers activity 6, Hevner guideline 7).

Based on the demonstration and evaluation results, we have performed extra iterations of activity two to five; each iteration aimed at a set of new requirements and points of improvements acquired during the demonstration and evaluation activities.

1.5.2 Research Methods

To answer the research questions, we have used different research methods, both qualitative and quantitative. This section explains which research methods are used and in which research lines a method is used.

We have used four research methods, namely literature study, model development, laboratory experiment, and case study research. Below, these methods are described in more depth. These research methods are used in the three lines of research: Architecture Compliance Checking Support (ACCS), Layered Architecture Design Support (LADS), and Enterprise Architecture Realization Assessment (EARA). An overview of the research methods used in each research line is provided in Table 1.1. Since the research approach for all three lines of research was design-science research, as discussed in the previous section, the same methods are used in the different lines of research. Except the method "laboratory experiment", this method has been used extensively in the first research line, but not in the other lines of research.

Literature Study

Literature study was an important part of all three research lines, and was performed during the activities 1,2, 3 and 6 of the DSRM process model (Peffers et al. 2008). Scientific literature was searched and used extensively, but to align with the professional field, standard works from this field were searched and used, where needed. For example, in the ACCS research line, literature was studied to identify requirements to ACC support. In the LADS research line, responsibility types were distilled from leading literature in the field of software architecture and layers. In the EARA research line, the designed instrument was based on principles found in literature, and on bodies of knowledge from the professional field. Furthermore, literature was studied extensively in all three lines of research to find related work, in order to reflect on the objectives and results of our work.

Model and Method Building

Models and methods are important artifacts of design-science research. "*Models* aid problem and solution understanding and frequently represent the connection

Table 1.1: Application of research methods per line of research

Research method \ Line of research	1 ACCS	2 LADS	3 EARA
Literature study	x	x	x
Model development	x	x	x
Laboratory experiment	x		
Case study	x	x	x

between problem and solution components enabling exploration of the effects of design decisions and changes in the real world. *Methods* define processes. They provide guidance on how to solve problems, that is, how to search the solution space. These can range from formal, mathematical algorithms that explicitly define the search process to informal, textual descriptions of best practice approaches, or some combination" (Hevner et al. 2004).

In all three lines of research, we have built models in the design process of the solutions. In case of the EARA research line, mathematical formalization of the model has provided a rigorous foundation of the metrics included in the solution.

Laboratory Experiment

A laboratory experiment in software engineering is characterized as follows: "The change proposal is evaluated in an off-line laboratory setting (in vitro), where an experiment is conducted and a limited part of the process is executed in a controlled manner" (Wohlin et al. 2012).

In case of the research line of ACC support, we have extensively made use of laboratory experiments, since such an experiment focuses on one factor and provides objective results. We have used laboratory experiments to explore the SRMA support of ACC-tools, evaluate the accuracy of dependency detection and violation reporting of ACC-tools, and to confirm the relevance of our findings.

Case Study Research

In our research, we have conducted multiple case studies (Yin 2014) for demonstration and evaluation purposes. In the ACCS research line, six different business information systems of four organizations were subject of an architecture compliance check, supported by our own tool and several commercial tools. This way, we were able to test our concepts and the tool's performance in practice, which resulted in new insights and new requirements for the next versions of our metamodel and tool.

In the LADS research line, three cases were used to demonstrate the applicability of the instruments developed within this line of research: a design case, a review case, and a complex case of a large governmental software system. These cases were also used to evaluate, the completeness and accuracy of the instruments.

In the EARA research line, four cases (two full assessments and two brief assessments) were used to demonstrate and evaluate the EARS instrument. The full assessments were conducted at a large governmental organization and a large financial organization. In both cases, ten stakeholders were interviewed.

1.6 Dissertation Outline

This dissertation is the outcome of research performed to answer the research questions in section 1.3. The research questions are answered in the following chapters of this dissertation. Chapters 2 to 7 each answer one research question or sub-question. These chapters have been written as individual papers for publication in scientific conference proceedings or journals. This entails some overlap in the abstracts and introductions of chapter 2 to 5, since the publications have resulted from the same line of research. Chapter 7 is an exception, as it results from merging a conference and a journal paper.

Chapter 1: Introduction

In this chapter, we first introduce the research domain and the problems that challenged us to initiate research. Second, we present the research questions, the starting points for research lines that address the challenges. Third, we provide an overview of our research approach and the used research methods. Finally, we explain the structure of the dissertation and how the following chapters are related to the research questions.

Chapter 2: Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support

This chapter answers research question RQ1.1. First, requirements are presented regarding the support of Semantically Rich Modular Architectures (SRMA) in the context of Architecture Compliance Checking (ACC). An SRMA contains modules of semantically different types, like layers and components, constrained by rules of different types. On basis of the requirements and an inventory of common module and rule types, eight commercial and non-commercial tools were tested. The test results show large differences between the tools, but all could improve their support of SRMA.

The work in this chapter was presented and published at the *IEEE International Conference on Software Maintenance (ICSM)* (Pruijt et al. 2013a).

Chapter 3: A Metamodel for the Support of Semantically Rich Modular Architectures in the Context of Architecture Compliance Checking

A partial answer to research questions RQ1.2 is presented, including an approach for support of SRMAs in the context of static ACC. The approach is grounded in the SRMACC metamodel, which enables support of rich sets of module and rule types. Furthermore, the metamodel enables extensive support of the semantics of

these types. To validate the feasibility of the metamodel, an open source prototype implementation was developed, tested and applied in practice.

The work in this chapter was presented and published at the *WICSA Workshop on Software Architecture Erosion and Architectural Consistency (SAEreCon)* (Pruijt and Brinkkemper 2014).

Chapter 4: HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types

This chapter complements the answer to research questions RQ1.2 and presents HUSACCT, a static ACC tool that adds extensive support for semantically rich modular architectures (SRMAs) to the current practice of static ACC tools. HUSACCT provides support for five commonly used types of modules and eleven types of rules. The chapter illustrates how basic and extensive support of these types is provided, and how the support can be configured. In addition, the internal architecture of the tool is discussed.

The work in this chapter was presented and published at the *ACM/IEEE International Conference on Automated Software Engineering* (ASE) (Pruijt et al. 2014).

Chapter 5: On the Accuracy of Architecture Compliance Checking Support: Accuracy of Dependency Analysis and Violation Reporting

Research question RQ1.3 is addressed in this chapter, which presents a study on the accuracy of ACC tools regarding dependency analysis and violation reporting. On the base of an inventory of 34 different dependency types, ten tools were tested and compared by means of a custom-made benchmark. In a second test, the code of open source system FreeMind was used to investigate the performance of the tools. Based on analysis of the test results, ten hard-to-detect types of dependency were identified and four challenges in dependency detection. The relevance of these findings is substantiated by a frequency analysis of the hard-to-detect types of dependencies in five open source systems.

A shorter version of the work in this chapter was presented and published at the *IEEE International Conference on Program Comprehension* (ICPC) (Pruijt et al. 2013b). The complete chapter is submitted for journal publication (Pruijt et al. 2015).

Chapter 6: A Typology Based Approach to Assign Responsibilities to Software Layers

This chapter answers research question RQ2, and presents the Typology of Software Layer Responsibility (TSLR) and a complementary instrument, the

Responsibility Trace Table (RTT). These instruments aid the design, documentation, and review of a layered software architecture with respect to the assignment of responsibilities. The application of the TSLR and RTT is demonstrated in three cases.

The work in this chapter was presented and published at the *20th Conference on Pattern Languages of Programs (PLOP)* (Pruijt et al. 2013d).

Chapter 7: The EARScorecard – An Instrument to Assess the Effectiveness of the EA Realization Process

As an answer to research question RQ3, this chapter presents the Enterprise Architecture Realization Scorecard (EARS) and an accompanying method to discover the strengths and weaknesses in the realization process of an EA management function. Two assessment cases illustrate the use of the instrument.

Chapter 7 is the integration of a conference and a journal paper. The former was presented and published at the conference Trends in Enterprise Architecture Research (Pruijt et al. 2012), while the latter was published in the Journal of Enterprise Architecture (Pruijt et al. 2013c).

Chapter 8: Conclusions

The final chapter of the dissertation provides an overview of the most relevant findings and contributions of our research. Furthermore, the implications and limitations of our research are discussed, as well as an agenda of possible future work in the three lines of research.

Appendix 1: Application Case HUSACCT

To illustrate the applicability of our ACC approach and our tool HUSACCT, as described in Chapter 2-4, a case study is presented in this appendix. The assessed system is an E-commerce system of a governmental organization. The intended architecture, implemented architecture, and the results of the compliance check are described and illustrated.

We regard our architecture compliance checking tool HUSACCT as a significant artifact of our research. HUSACCT is free-to-use and open source. The executable, user manual and source code are downloadable at http://husacct.github.io/HUSACCT/. An introduction video is accessible at the same site.

Chapter 2

Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support

Architecture Compliance Checking (ACC) is an approach to verify the conformance of implemented program code to high-level models of architectural design. ACC is used to prevent architectural erosion during the development and evolution of a software system. Static ACC, based on static software analysis techniques, focuses on the modular architecture and especially on rules constraining the modular elements. A semantically rich modular architecture (SRMA) is expressive and may contain modules with different semantics, like layers and subsystems, constrained by rules of different types. To check the conformance to an SRMA, ACC-tools should support the module and rule types used by the architect. This chapter presents requirements regarding SRMA support and an inventory of common module and rule types, on which basis eight commercial and non-commercial tools were tested. The test results show large differences between the tools, but all could improve their support of SRMA, what might contribute to the adoption of ACC in practice.

2.1 Introduction

Software architecture is of major importance to achieve the business goals, functional requirements and quality requirements of a system. However, architectural models tend to be of a high-level of abstraction, and deviations of the software architecture arise easily during the development and evolution of a system (Murphy et al. 1995). Architecture Compliance Checking (ACC) is an approach to bridge the gap between the high-level models of architectural design and the implemented program code, and to prevent decreased maintainability, caused by architectural erosion. *Architectural erosion* is "the phenomenon that occurs when the implemented architecture of a software system diverges from its intended

architecture" (de Silva and Balasubramaniam 2012). Opposing terms are architecture compliance and its synonym architecture conformance. Knodel and Popescu defined *architecture compliance* as "a measure to which degree the implemented architecture in the source code conforms to the planned software architecture" (Knodel and Popescu 2007).

Many tools and techniques are available to analyze a software system, and to reconstruct, visualize, check, or restructure its architecture (Ducasse and Pollet 2009). In our study we focus on tools supporting static ACC, which analyze the software without executing the code. These tools, which we label as *static ACC-tools,* focus on the modular structure in the source code and identify structural elements such as packages and classes. In addition, they analyze use-relations between these elements, such as an invocation of a method or access of an attribute. Furthermore, these tools support the definition of rules on the structural elements in the code, or on logical modular elements that are mapped to the code. Finally, ACC-tools check the compliance and report violations to the rules. For example, if a method call from class A to class B in the code corresponds with a not-allowed dependency from a lower layer to a higher layer in the intended architecture, then the tool should report a violation.

Although Shaw and Clements included ACC in 2006 in their list of promising areas (Shaw and Clements 2006), the adoption of ACC-tools is still limited (de Silva and Balasubramaniam 2012, Gleirscher and Golubitskiy 2012), and research is necessary to advance and improve current methods and tools (Canfora et al. 2011). A few studies have compared ACC-tools and techniques, and these studies revealed large differences in terminology and approach. A high level overview of techniques and tools is included in a survey on architectural erosion (de Silva and Balasubramaniam 2012) and in a survey on software architecture reconstruction (Ducasse and Pollet 2009). Two other studies (Knodel and Popescu 2007, Passos et al. 2010) identified and compared five static ACC techniques at a more detailed level. One of these studies (Passos et al. 2010) also explored the effectiveness and usability of three tools, each representing one technique, by executing tests on the basis of a small system.

Our research builds on these previous studies, but we focus on ACC-tool support of *semantically rich modular architectures* (SRMAs). We use this term for expressive modular architectures, composed of different types of modules, which are constrained by different types of rules; explicitly defined rules, but also rules inherent to the module types. Kazman, Bass, and Klein have stated the principle that elements in a software architecture should be coarse enough for human intellectual control, but also specific enough for meaningful reasoning (Kazman et al. 2006). Modules with specific semantics, like subsystems, layers, components or facades, enhance the expressiveness of a modular architecture and support

architecture reasoning. Adersberger and Philippsen consider the support of semantically rich architecture models essential for the integration of ACC in model-driven engineering (Adersberger and Philippsen 2011). Furthermore, they make clear that support of semantically rich constructs reduces the number of rules that need to be defined, compared to semantically poorer boxes and lines models.

We started our study with the following research question: *Do static ACC-tools provide functional support for semantically rich modular architectures?* To answer this question, we identified requirements, developed test-ware based on the requirements, and we tested eight ACC-tools. We restricted our study to the functional support of SRMAs by ACC tools, and consequently we do not focus on other aspects, like usability, scalability or accuracy (in another study, we investigated the accuracy of dependency analysis and violation reporting (Pruijt et al. 2013b)). Other approaches than ACC that may be supported by the same tools, like architecture reasoning and re-engineering, are outside the scope of this paper as well.

The next section of this chapter identifies the information available in semantically rich modular architectures, presents requirements and a classification of common module and rule types. Section 2.3 describes the test method and introduces the tools, while Section 2.4 holds the test results. Section 2.5 discusses the test outcome and compares it to related work, while Section 2.6 concludes this chapter with recommendations, and addresses some issues that require further research.

2.2 Modular Architectures

2.2.1 Focus of Static ACC

Software architecture compliance checking covers a large field, since software architecture is a broad term. According to Perry and Wolf, software architecture "provides the framework within which to satisfy the system requirements and provides both the technical and managerial basis for the design and implementation of the system" (Perry and Wolf 1992). Static ACC does not cover the full width of software architecture, but only the static structure of the software: the modular architecture. According to the Views and Beyond approach (Bass et al. 2012, Clements et al. 2010), module styles focus on the structure of the units of implementation and not on runtime behavior or the allocation to non-software resources. Different module styles are defined such as the decomposition style, uses style, generalization style, and layer style.

A modular architecture should describe the modular elements, their form (properties and relationships) and rationale (Perry and Wolf 1992). Modular elements, properties and relationships, are in ACC's center of attention, and should be included in a complete compliance check. A *modular element*, or module, is an implementation unit of software with a coherent set of responsibilities (Clements et al. 2010). Properties and relationships express architectural rules. *Properties* are used to define constraints on the modular element and its content. *Relationships* are used to constrain how the different elements may interact or otherwise may be related (Perry and Wolf 1992).

2.2.2 Requirements Regarding SRMA Support

A semantically rich modular architecture may contain a lot of information about the modules and the rules constraining these modules. Modules may be of types with different semantics, while different types of rules may be used to constrain the modules. A rich set of module types provides a language to express characteristics of the modules in an architectural model, as well as default constraints associated to the type of module. A rich set of rule types provides a language to express constraints on the modules in an architectural model. Provision of a rich rule set allows architects to define logical rules in a comparable way as expressed in regular language, without the need to translate a logical rule to one or more rules at tool level.

Consequently, to support compliance checks of SRMA's, ACC-tools should preferably be able to: a) register common information in SMRAs (modular elements, properties and relationships of different types); b) prevent inconsistencies in the definition of the architectural model; and c) check the rules included in the architectural model and report violations. Inconsistencies in the model, like modules not properly mapped to code, will hamper the accuracy of the actual rule check. Consequently, inconsistencies should be recognized and reported.

In line with these requirements, we focused our research on the following questions. Do ACC-tools provide support for: a) common types of modules and their semantics; b) common types of rules; and c) inconsistency prevention within the defined architecture?

To determine the module types, rules types and inconsistency checks relevant to our research, we studied academic and professional literature, as well as software architecture documents from professional practice and ACC-tool documentation. The following subsections describe the outcome of our study.

2.2.3 Common Module Types

SMRAs may contain modules of different types. We identified the following six common types of modules relevant for static ACC:

1. *Physical clusters* are the type of modules that represent a wide variety of software structures or units in the code, like classes, Java packages, or C# namespaces (Clements et al. 2010). This type of module does not represent a unit in the design, but in the code.
2. *Logical clusters* represent units in the system design with clearly assigned responsibilities, but with no additional semantics. Comparable terms are subsystems, or packages.
3. *Layers* represent units in the system design with additional semantics. Layers have a hierarchical level and constraints on the relations between the layers. The concept of layering can be traced back to the works by Dijkstra (Dijkstra 1968) and Parnas (Parnas 1972). Although the layered style is not supported by UML (Shaw and Clements 2006), it is one of the most common styles used in software architecture (Clements et al. 2010, Harrison and Avgeriou 2008). We cite Larman (Larman 2005), who summarizes the essence of a layered design as "the large-scale logical structure of a system, organized into discrete layers of distinct, related responsibilities. Collaboration and coupling is from higher to lower layers."
4. *Components* within a software architecture are designed as autonomous units within a system. The term component is defined in different ways in the field of software engineering. In our use, a component within a modular architecture covers a specific knowledge area, provides its services via an interface and hides its internals (in line with the system decomposition criteria of Parnas (Parnas 1972)). Consequently, a component differs from a logical cluster in the fact that it has a Façade sub module and hides its internals. Since our definition of component is intended for modular architectures, it does not include runtime behavior, and a module in a module view may turn into many runtime components within the "component and connector view" (Clements et al. 2010).
5. *Facades* are related to a component and act as an interface as described under components. We use the term façade, referring to the façade pattern (Gamma et al. 1995), to differentiate with the Java interface, which has not exactly the same meaning as a design-level interface. A facade may be mapped to multiple elements at implementation level, like Java interface classes, exception classes and data transfer classes.
6. *External systems* represent platform and infrastructural libraries or components used by the target system. Useful ACC support includes the identification of external system usage and checks on constraints regarding their usage (Ali et al. 2012).

2.2.4 Example of an SRMA

An example of an SRMA, with modular elements of different types, is shown in Figure 2.1. The model shows a part of a modular architecture of one of the systems at an airport, where it was subject of an ACC. This system is used to manage the state and services of human interaction points where customers communicate with baggage handling machines, self-service check-in units, et cetera.

Various notations for modular architecture diagrams are used in practice (Clements et al. 2010). The example in Figure 2.1 shows UML icons, but also an identification of the layers, not included in UML. The model combines three modular styles, namely the decomposition style, uses style, and layered style. Examples of modules of different types are visible in Figure 2.1. such as "Interaction layer", logical cluster "HiWeb", component "HiManager", façade "HimInterface", and external system "Hibernate". The modules are easily identifiable, but the rules are not. In this case, the basic principle is, "no module is allowed to use another module", except when a dependency relation indicates "is allowed to use". Furthermore, the rules related to the layered style are not visible, but the default rules apply: Interaction Layer is not allowed to use Technology Layer (skip call ban); Technology Layer is not allowed to use Service layer or Interaction Layer (back call ban).

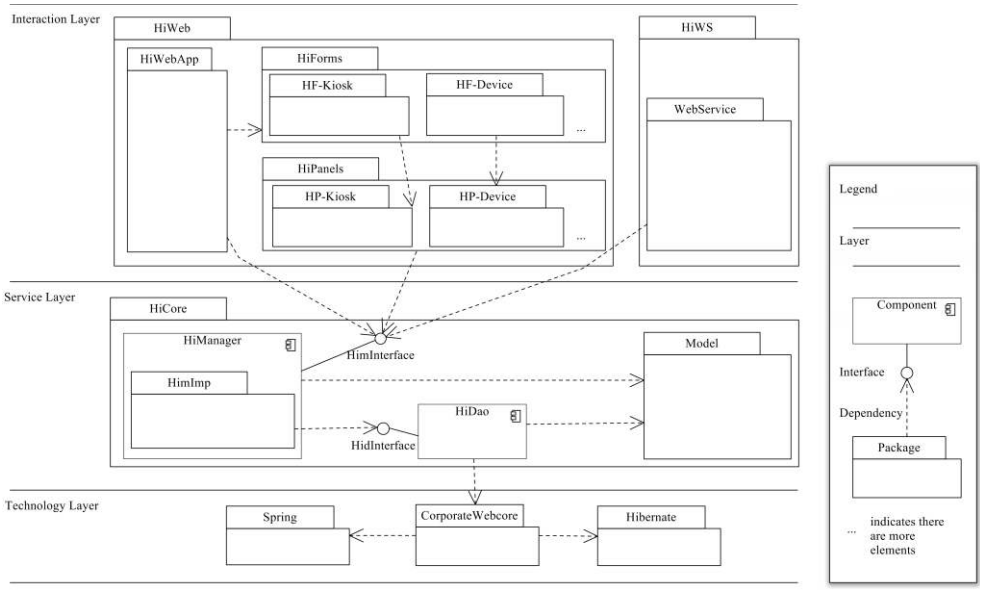

Figure 2.1: Example of a semantically rich modular architecture model

2.2.5 Common Rule Types

SMRAs may contain rules of different types, where each rule type characterizes the constraint. Constraints in a software architecture are categorized in literature (Perry and Wolf 1992, Clements et al. 2010) as properties and relationships. Our inventory of architectural rule types, in principle verifiable by static ACC, resulted in two categories related to properties and relationships: Property rule types; and Relation rule types.

Property rule types constrain the elements included in the module; their sub modules, et cetera. Clements et al. (Clements et al. 2010) distinguish the following properties per module: Name, Responsibility, Visibility, and Implementation information. We identified rule types associated to these properties and named them accordingly, except two types (Façade convention, Inheritance convention), which represent the property Implementation information. The identified rule types are shown in Table 2.1. The table contains per rule type: a description, an example, and an exemplary reference to literature covering the topic. The example rules constrain the modules of the modular architecture shown in Figure 2.1.

Naming conventions may be useful, since names are used by practitioners to unify software architecture and its implementation (Woods and Rozanski 2010). Responsibility conventions are useful to preserve the designed distribution of responsibilities over modules. Visibility conventions and Façade conventions can be used to enforce implementation hiding. Inheritance conventions may be used to enforce a selected generalization style. Finally, exceptions to property rules may be useful too. For instance, an exception to the Visibility convention example in Table 2.1 is, “HiManager classes have package visibility or lower, except for façade HimInterface.”

Relation rule types specify whether a module A is allowed to use a module B. The basic types of rules are “is allowed to use” and “is not allowed to use”. However, we encountered useful specializations of both basic types, which we included in the classification shown in Table 2.1. When several rules of the same type are defined on the same from-module, then they should be interpreted as complementary rules; even if the word “only” is part of the name of the rule type.

Table 2.1: Common rule types (Ref= primary literature reference)

Category\Type of Rule	Description (D), Example (E)	Ref
Property rule types		
Naming convention	D: The names of the elements of the module must adhere to the specified standard. E: HiDao elements must have suffix DAO in their name.	[1]
Responsibility convention	D: All elements of the module must adhere to the specified responsibility. E: HiForms is responsible for presentation logic only.	[2]
Visibility convention	D: All elements of the module have the specified or a more restricting visibility. E: HiManager classes have package visibility or lower.	[3]
Facade convention	D: No incoming usages of the module are allowed, except via the façade. E: HiManager may be accessed only via HimInterface.	[4]
Inheritance convention	D: All elements of the module are sub classess of the specified super class. E: HiDao classes must extend CorporateWebCore.Dao.GenEntityDao.	[1]
Relation rule types		
Is not allowed to use	D: No element of the module is allowed to use the specified to-module. E: HF-Kiosk is not allowed to use HP-Device.	[5]
Back call ban (specific for layers)	D: No element of the layer is allowed to use a higher-level layer. E: Service Layer is not allowed to use the Interaction Layer.	[6]
Skip call ban (specific for layers)	D: No element of the layer is allowed to use a lower layer that is more than one level lower. E: Interaction Layer is not allowed to use the Infrastructure Layer.	[6]
Is allowed to use	D: All elements of the module are allowed to use the specified to-module. E: HiWebApp is allowed to use HiForms (including its submodules).	[3]
Is only allowed to use	D: No element of the module is allowed to use other than the specified to-module(s). E: HF-Kiosk is only allowed to use HP-Kiosk.	[1]
Is the only module allowed to use	D: No elements, outside the selected module(s) are allowed to use the specified to-module. E: HiDao is the only module allowed to use CorporateWebcore.	[1]
Must use	D: At least one elements of the module must use the specified to-module. E: HiDao must use CorporateWebcore.	[5]

[1] (Passos et al. 2010), [2] (Larman 2005), [3] (Clements et al. 2010), [4] (Gamma et al. 1995),
[5] (Knodel and Popescu 2007), [6] (Sarkar et al. 2006)

Some rule types are complex, because they include dependency checks on other modules than only the from-module and to-module. Exceptions to all relation rules are complex, as well as the two following types: “Is only allowed to use”, and “Is the only module allowed to use”. Complex rule types are very useful in practice, for the following reasons:

- Complex rule types allow architects to define rules in a comparable way as expressed in regular language. Complex rules of type “Is only allowed to use” may constitute a significant part of the total rule set (Terra and Valente 2009).
- Complex rule types help to transform rules in a UML-like diagram to rules in most ACC-tools. For instance, the dependency relationship from module HF-Kiosk to module HP-Kiosk in Figure 2.1 expresses the rule “HF-Kiosk is only allowed to use HP-Kiosk.” Transformation is often necessary. The basic principle underlying UML-like diagrams is restricting (no other than the defined dependencies are allowed), while in most tools, the basic principle is non-restricting (all dependencies are allowed, unless there is a not-allowed-to-use rule).
- Complex rule types may diminish the number of rules, since one complex rule often replaces many “is not allowed to use” rules. For instance, when the “is only allowed to use” rule type is not supported by a tool, than the dependency relationship from module HF-Kiosk to module HP-Kiosk in Figure 2.1 may have to be translated to many “not allowed to use” rules from HF-Kiosk to all the other modules, except to HP-Kiosk.

2.2.6 Associations between Module and Rule Types

Optimal support of SRMAs includes the automatic provision of rule types inherent to the type of module. For instance, layers are inherently associated to a “Back call ban” rule and a “Skip call ban” rule. Furthermore, components are inherently associated to a “Façade convention” rule (and possibly a “Visibility convention” rule, if supported by the implementation language). Options to disable an inherent rule, for instance in case of a relaxed layered model, or to define an exception, will enhance the usability.

2.3 Test Method and Tested Tools

2.3.1 Test Method

Based on the requirements and classification of module types and rule types described in Section 2.2, a test was designed to assess the ACC-tools on their SRMA support. For each rule type, at least two test cases were included: one without, and one with violations to the rule. A special test software system was developed in Java. This system included the various module types and separate packages for each rule type, which contained classes with injected violations to a rule and classes without. In addition, a test script was prepared to instruct the tester and to document the test results. The test script and test system are available on request.

After the test preparation, the eight ACC-tools were tested. During the first step of the test of a tool, the intended architecture was entered. Thereafter, the modules were mapped to source code units and the rules were entered into the tool. If a tool did not support a rule type explicitly, then we looked for a workaround; such as a combination of separate rules. The first step was concluded by test actions aimed at the tool's ability to prevent inconsistencies in the architecture definition. During the second step, the outputs of the tool's dependency analysis and conformance check were studied and compared with the expected result. During the third step, reports were prepared, after which the tools could be compared on their SRMA support.

Two iterations of testing and reporting were conducted. The first iteration was performed with 25 bachelor students in the course of a third year specialization semester "Advanced Software Engineering", where each team studied and tested a tool. In a second iteration, the authors studied the tools, and verified and refined the results of the students, by using the tools and repeating the tests. ConQAT was added afterwards to our tool set and was tested only by the authors.

2.3.2 ACC-Tools Included in the Test

Many tools are available with some facilities to support ACC. Our research focused on tools with explicit support of ACC. We selected eight publicly available tools, which were mentioned in academic work (e.g., (Ducasse and Pollet 2009) (Passos et al. 2010) (Adersberger and Philippsen 2011)), were able to analyze Java, and provided evaluation or research licenses (two vendors rejected and one did not respond). We excluded tools that focus mainly on architecture visualization, metrics and/or architecture refactoring. The eight tools included in our study are shown in Table 2.2, which also gives an overview of functionalities, code variants and licensing.

The tools provide their support of ACC in various ways. The eight tools can be subdivided in the following four categories of tools.

1. Macker and Sonar Architecture Rule Engine (Sonar ARE) are text-based tools, which support relation conformance rules. These tools provide HTML-based violation reports.
2. dTangler and Lattix are based on the Dependency Structure Matrix (DSM) technique, complemented with text-based editors to define rules. The DSM is used to select modules and to show dependencies and violations. Lattix is also able to visualize architectures graphically, and provides extensive reporting facilities.

Table2.2: Characteristics of the tools in the test

Tools[1] → **Characteristics ↓**	**ConQAT AA**	**dTangler**	**Lattix**	**Macker**	**SAVE**	**Sonar ARE**	**Sonargraph Architect**	**Structure 101**
General functionalities								
Dependency browsing		√	√				√	√
Dependency visualization			√		√		√	√
Architecture compliance checking	√	√	√	√	√	√	√	√
Architecture refactoring/simulation			√				√	√
Team support							√	√
Code variants								
Java	√	√	√	√	√	√	√	√
Other languages	√		√		√			√
Source file analysis					√	√	√	
Compiled file analysis	√	√	√	√		√	√	√
Licensing								
Free: commercial and non-commercial use	√	√		√		√		
Paid: commercial use			√		√		√	√

[1] ConQat AA– version 2011.9 – www.conqat.org;
dTangler - GUI version 2.0 - web.sysart.fi/dtangler;
Lattix LDM - version 7.2 - lattix.com;
Macker - version 0.4.2 - sourceforge.net/projects/macker;
SAVE - version 1.7 - iese.fraunhofer.de;
Sonar ARE - version 3.2 - docs.codehaus.org/display/SONAR/Architecture+Rule+Engine;
Sonargraph Architect (fusion of Sotograph and SonarJ) - version 7.0 - hello2morrow.com;
Structure101 - version 3.5 - structure101.com.

3. ConQAT Architecture Analysis (ConQAT AA) and SAVE are strictly based on the Reflexion Model (RM) technique (Murphy et al. 1995), and both tools provide a graphical editor to define the intended architecture and to show violations after the evaluation. Textual reports are generated at request.
4. Sonargraph Architect and Structure101 are diagram-based too, but these tools are not based on the RM-technique. To define modules and rules, these tools provide diagrams in which the horizontal and vertical position of a module implies rules. Violations are shown in these diagrams, but textual reports are provided in addition.

2.4 Test Results

2.4.1 Support of Common Module Types

In Section 2.2 we identified six common types of modules, relevant for static ACC. The results of our tests concerning the support of these module types are shown in Table 2.3, and the most interesting findings are described below.

Clusters are supported by all tools. Five of the eight tools support *physical clusters*. The advantages to use them are that they allow fast, ad hoc rule checking; for instance, when there is no formal modular architecture. The disadvantage is the diminished or lost traceability to the formal modular architecture, if there is one. Sonar ARE is the only tool that supports only this type of modules. *Logical clusters* are supported by seven tools. Although in very different ways, these tools provide support to register logical clusters and to map the logical clusters to code units. Furthermore, support is provided to define rules constraining logical clusters and to check these rules at code level.

Layers are supported by only one tool, Structure101, on all indicators: modules can be marked as layers; back call and skip call rules are reported; and layers are visualized. Two other tools support the definition and visualization of layers, but do not provide inherent support of the related rules.

Components and *Facades* are supported by SAVE and Sonargraph Architect, on the following indicators: modules can be marked as component; facades can be defined. SAVE visualizes components and facades, but does not actively support any of their semantics. Sonargraph Architect visualizes facades and supports their semantics; it reports facade-skip violations automatically when a facade is associated to a module. ConQAT AA seems to support components at first glance, since it depicts all modules as UML components. However, it does not provide any other icons and does not support the semantics of a component; reason why we classified ConQAT's components as logical clusters.

External systems are not designated as a special module type by all tools, except Sonargraph Architect, but all enable conformance checks on modules mapped to external libraries.

Five tools support visualization of modular architectures. However, only two tools offer three or more different icons. A notable observation is that the tools that support semantically rich modules all have their own terminology, icons, rules and ways to visualize the architecture. SAVE provides an UML-like notation, while

Table 2.3: Tool support of common module types
(+ = explicit support; ± = partial support; - = no support)

	ConQAT AA	dTangler	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
Clusters								
Physical cluster	-	+	+	+	-	+	-	+
Logical cluster	+	+	+	+	+	-	+	+
Layers								
A module can be marked as layer	-	-	-	-	+	-	+	+
Back call violations are reported	-	-	-	-	-	-	-	+
Skip call violations are reported	-	-	-	-	-	-	-	+
Components and Facades								
A module can be marked as component	-	-	-	-	+	-	+	-
Facade can be defined	-	-	-	-	+	-	+	-
Facade-skip violations are reported	-	-	-	-	±	-	+	-
External systems								
A module can be marked as external system	-	-	-	-	-	-	+	-
A module can be mapped to an external system	+	+	+	+	+	+	+	+
Rules constraining their use are checked	+	+	+	+	+	+	+	+
Visualization								
Clusters are visualized	+	-	+	-	+	-	+	+
Layers are recognizable visualized	-	-	-	-	+	-	+	+
Components are recognizable visualized	-	-	-	-	+	-	-	-
Facades are recognizable visualized	-	-	-	-	+	-	+	-
External systems are recognizable visualized	-	-	-	-	+	-	+	-

Sonargraph Architect and Structure101 position the modules horizontally and vertically. SAVE discerns five module types, while Sonargraph Architect discerns six types (which are only partly overlapping with those of SAVE), whereas Structure101 does not show the logical meaning of a module, but uses an icon to show the type of the related physical item.

2.4.2 Support of Common Rule Types

In Section 2.2 we identified twelve common types of rules, relevant for static ACC. The results of our tests concerning the support of these rule types are shown in Table 2.4. Explicit support of a rule type is depicted by a "+", meaning that one

Table 2.4: Tool-support of common rule types
(+ = explicit support; ± = partial support; - = very weak or no support)

Support is provided for	**ConQAT AA**	**dTangler**	**Lattix**	**Macker**	**SAVE**	**Sonar ARE**	**Sonargraph Architect**	**Structure 101**
Property rule types								
Naming convention	-	-	-	-	-	-	-	-
Responsibility convention	-	-	-	-	-	-	-	-
Visibility convention	-	-	-	-	-	-	±	±
Facade convention	-	±	±	±	±	-	+	-
Superclass inheritance convention	-	-	-	-	-	-	-	-
Relation rule types								
Is not allowed to use	+	+	+	+	+	+	+	±
Back call ban (inherent to layer)	-	-	-	-	-	-	-	+
Skip call ban (inherent to layer)	-	-	-	-	-	-	-	+
Is allowed to use	+	+	+	+	±	-	+	+
Is only allowed to use	±	±	±	±	±	-	±	±
Is the only module allowed to use	±	±	±	±	±	-	±	±
Must use	-	-	-	-	+	-	-	-
Exception (to relation rules)	±	±	±	±	-	-	±	±
Visualization of rules and violations								
Rules are visualized	+	-	+	-	+	-	+	+
Violations are visualized	+	+	+	-	+	-	+	+

logical rule can be registered as one rule in the tool. Partial support, depicted by "±", means that it is possible to register a rule of this type, but only via a workaround; often a combination of several rules. The most interesting findings from the test are described below.

Property Rule Types

Property rule types are poorly supported. No tool provides facilities to specify and check conventions regarding naming, responsibility, or inheritance. Although names are used, in combination with regular expressions, to map modules to the code, no facilities are provided to check all the packages and/or classes contained by a module on conformance to a naming convention.

Only rule types to enforce implementation hiding are supported by some tools. Visibility convention rules are partly supported by Sonargraph Architect and Structure101. These tools provide a property to restrict the accessibility of a module, but do not check at code level on accessibility settings; reason why they did not score a "+". However, when a module is marked as hidden or private, violation messages are reported, when dependencies to the module are detected from outside.

Façade convention rules are supported explicitly only by Sonargraph Architect. Four other tools enable the definition of this type of rules by default means, resulting in a combination of separate rules, so their support is scored with "±".

Relation Rule Types

Relation rule types are supported by all the tools, but no more than three rule types are explicitly supported per tool.

Complex rule types (Is only allowed to use, Is the only module allowed to use, Exceptions to a relation rule) are not- explicitly supported, or not at all. Without explicit support, workarounds are needed, for instance for the rule "HF-Kiosk is only allowed to use HP-Kiosk". In Lattix, dTangler and Macker, two combined rules are needed such as: "HF-Kiosk Cannot-Use $Root" + "HF-Kiosk Can-Use HP-Kiosk". Since these rules are not related to each other, they form a threat to the maintainability and traceability of the set of rules. Sonargraph Architect and Structure101 may require the specification of more than two rules or property settings for complex rules, and sometimes many rules are needed, depending on the number and position of other modules. Sonar ARE provides no support at all to check complex rules. ConQAT AA and SAVE work quite differently from the other tools, since no transformation is required of rules in UML-like diagrams to rules in the tool. SAVE supports only the "Must use" rule type explicitly, while ConQat AA supports "Is allowed to use" and "Is not allowed to use" rule types. Complex rules can be checked, but this requires interpretation of the architecture model and the conformance check output.

Visualization
Six tools are able to visualize rules and violations. Lattix and dTangler show colors in a DSM. ConQAT AA, SAVE, Sonargraph Architect, and Structure101 use lines in diagrams to define and show rules, and to show violations. However, not all rules are visible in these diagrams.

2.4.3 Support of Inconsistency Prevention

In Section 2.2 we defined the requirement, "ACC-tools should prevent inconsistent definitions of modules and rules." The results of our tests concerning this requirement are shown in Table 2.5. Most tools allow, without a warning, incomplete or contradictory definitions of modules and/or rules. ConQAT AA scored best and prevented six out of six types of inconsistency included in our test. Lattix prevented five out of six types, while the other tools prevented or warned for upmost three types. Six of the tools start the compliance check without a warning when the defined modules and rules model is inconsistent. In such a case, the tool does not check all the rules as intended by the user, and consequently the outcome of the check may be unreliable.

2.5 Discussion

To our opinion, all tested tools are providing useful functionality to support ACC or ad hoc rule checking. Apart from our laboratory experiments described in this

Table 2.5: Prevention of Inconsistencies
(+ = supported; - = not supported; n/a = not applicable)

	ConQAT AA	dTangler	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
Modules must have (unique) name or ID	+	+	+	+	-	n/a	+	-
A module may have only one parent.	+	-	+	-	+	n/a	-	-
Modules must be mapped to code file(s)	+	-	-	-	-	n/a	-	+
Mapped code files must exist	+	-	+	-	-	n/a	-	-
Rules must be completely specified	+	-	+	+	+	-	+	+
Rules cannot be contradictory	+	-	+	-	+	-	+	+
Tool checks model prior to conf. check	+	-	+	-	-	-	-	-

chapter, we used all eight tools to analyze an open source system. Furthermore, we performed ACCs on professional software systems with use of Lattix, Sonargraph Architect, and Structure101. Based on these experiences we can conclude that these tools are of great help for architecture reconstruction and ACC. However, our tests show that all eight tools could improve their support regarding SRMAs, though in varying degrees. Not one of the tested tools is able to support all the module types and rule types included in our classification. However, we encountered interesting examples of partial support. SAVE supports the graphical definition of modules of nearly all the types in our classification; only physical clusters are missing. However, SAVE's rule language is very limited, and the semantics of the modules are not supported. ConQAT provides the same types of diagrams, but complements the rule setting capabilities considerably. Furthermore, ConQAT checks the consistency of the defined architectural model accurately. However, ConQAT provides one type of module only, and does not support any semantics. Sonargraph Architect and Structure101 are the only tools that actively support the semantics of two module types in our classification. Sonargraph Architect supports the definition of Facades and relates the "Façade convention" rule to a defined façade. Structure101 supports the definition of Layers and relates a layer to the "Back call ban" and the "Skip call ban" rules. Combination of these examples of partial support builds an image of the provision of full functional SRMA support.

Another observation during our study is that the combination of visualization, rule definition, and rule checking appears to be challenging. Lattix, dTangler, and Macker provide no support to define the architecture via a graphical editor, but enable the definition and checking of quite a diversity of rules, including complex rules. ConQAT, SAVE, Sonargraph Architect and Structure101 provide graphical support to define and check the architecture, but lack the freedom of rule definition, as provided by the textual-rule based tools. Furthermore, sometimes we experienced serious problems related to the graphical models. Defining sub-subsystems, exceptions, and other complex rules in the graphical models, is hard in some tools, and impossible in others. Furthermore, it may result in many lines, which makes the diagrams unreadable. Structure101 and Sonargraph Architect have introduced additional rule-setting techniques to reduce the number of required rule-lines. In these tools, the module type, the horizontal and vertical position, and the value of a visibility property per module may imply dependency rules. On top of that, Sonargraph Architect provides a "transversal access" variable per module as well. To our opinion, the combination of all the rule-setting techniques increases the complexity considerably, and it reduces the transparency of the set of defined rules.

2.5.1 Limitations

Our study can be characterized as a quasi-experiment, according to Wohlin et al. (Wohlin et al. 2012), since we did not work with a randomized selection of tools. Consequently, our findings may not be generalized to other tools, even though we tested eight tools in a small market.

Furthermore, we do not claim that our classification of common module types and common rule types is complete, since *common* is not a qualified term. We aimed to cover the most used types of modules and rules, reasoning from the functional point of view; the architect's view, not the tool builder's view. Creating the classification proved a valuable step in our study. The classification was used as a basis for our tests and will be used as starting point for our future work.

2.5.2 Related Work

Requirements regarding the functional support of ACC can be derived from quite a number of sources, like general literature on software architecture and design, and studies on ACC. In Section 2.2 we described the most relevant sources used for our requirements and classification. Several studies on ACC propose the inclusion of support for some specific module and/or rule types, for instance (Adersberger and Philippsen 2011) (Ali et al. 2012) (Knodel and Popescu 2007) (Passos et al. 2010). However, to the best of our knowledge, none of these studies or other studies on ACC have provided and substantiated a broad inventory and classification of module and rule types. We intentionally did not include very specific or detailed module or rule types, but kept the set of requirements broad and not too specific. However, some interesting studies elaborate on particular types. For instance, Adersberger and Philippsen (Adersberger and Philippsen 2011) describe the constraints and checks regarding components in detail. Furthermore, Terra and Valente (Terra and Valente 2009) identified different types of dependencies (accessing methods and fields, declaring variables, creating objects, extending classes, implementing interfaces, throwing exceptions, and using annotations), and based fine grained rule types on these dependency types. Lattix, SAVE and Structure101 provide support to define or configure rules at this level of detail.

Not much comparative research on ACC-tools has been performed, as described in the Introduction section. Only Passos et al. (Passos et al. 2010) presented similar work. They evaluated three tools, including Lattix and SAVE, on the basis of a very small system. During our study no findings have arisen that contradict their tool evaluations. Our study adds a substantiated set of requirements focused on SRMA support, as well as test results of eight tools.

2.6 Conclusion

Architecture compliance checking (ACC) relies on the support of tools to define modules and rules, to analyze the code, to check the compliance, and to report violations to the rules. In this study, we have investigated the support of semantically rich modular architectures (SRMAs) provided by static ACC-tools. We identified requirements to the support of SRMAs and classified module types and rule types relevant for static ACC. Furthermore, we prepared a test, and we tested eight tools on their support of SRMAs.

We started our study with the following research question: Do static ACC-tools provide functional support for semantically rich modular architectures? We focused our test on the support of: a) common types of modules and their semantics; b) common types of rules; and c) inconsistency prevention within the defined architecture.

Our tests regarding the *support of common module types* show that five tools support non-semantic clusters only. The three other tools distinguish also one or more semantically rich module types from our classification. SAVE supports the graphical definition of five types of modules, but does not support their semantics. Sonargraph Architect supports the semantics of a Façade actively, while Structure101 supports the semantics of Layers actively. However, no tool provides the combined support of layers, components, and facades.

Our tests regarding the *support of common rule types* show that per tool only a few rule types are explicitly supported. Complex relation rules are by no tool explicitly supported. Consequently, complex relation rules at logical level require workarounds at tool-level, which often result in two or more unrelated rules; a threat to the maintainability and traceability of the set of rules. Furthermore, only two of the five property types are supported, and only partially, not explicitly.

Our tests regarding the *support of inconsistency prevention* show that only two tools, ConQAT and Lattix, score high on the prevention of inconsistencies in the module and rule model, while inconsistent models may result in an unreliable outcome of the compliance check.

Based on our study and experiments, we present the following recommendations to ACC-tool developers:

1. *Widen the scope of the tools from dependency checking to software architecture compliance checking, including SRMAs.* Provide explicit support for semantically rich module types with their related rule types. The requirements and the classification of common module and rule types, presented in this chapter, may be used as a starting point.

2. *Minimize the difference between logical rules, as perceived by the architect, and the technical implementation in the tool.* Offer rule types that match with logical rule types, including exceptions, and support each type explicitly.
3. *Provide one method to define and edit rules.* Do not mix several rule setting mechanisms. Keep it simple to the user.
4. *Provide several, best adaptable, views* on the modular structures, the rules, and the violations against the rules: reports, browsers, and diagrams. Do not mix too much types of information into one view.
5. *Check on inconsistencies in the architecture definition*, and inform the user when it is incorrect or incomplete.

Not all issues identified in this study can be solved easily. The provision of SRMA support calls for further research. Techniques need to be identified, and support needs to be designed and tested on effectiveness by means of prototypes and case studies. Specific topics deserve attention too. For instance, visualization, rule definition, and rule checking appeared to be a challenging combination. Furthermore, automatic recognition of responsibility at code level, needed to check against the defined responsibility of a module, is an unresolved issue, though responsibility is an important property of a module at design level.

In conclusion, the eight tested tools provide useful support for ACC, but all could improve their support of SRMAs. Solutions need to be found to reduce the gap between documented modular architectures in software architecture documents on one side, and module and rule models in ACC-tools on the other side. More-complete functional support of SRMAs might contribute to the adoption of ACC and ACC-tools, and consequently could improve the effectiveness of software architecture in the practice and education of software engineering.

Chapter 3

A Metamodel for the Support of Semantically Rich Modular Architectures in the Context of Architecture Compliance Checking

Architecture Compliance Checking (ACC) is an approach to verify the conformance of implemented program code to high-level models of architectural design. Static ACC is focused on the module views of architecture and especially on rules constraining the modular elements. This chapter proposes an approach for support of semantically rich modular architectures (SRMAs) in the context of static ACC. An SRMA contains modules of semantically different types, like layers and components, constrained by rules of different types. Our approach is grounded in a metamodel, which enables support of rich sets of module and rule types and which enables extensive support of the semantics of these types. To validate the feasibility of the metamodel, an open source prototype implementation was developed, tested and applied in practice.

3.1 Introduction

Software architecture is of major importance to achieve the business goals, functional requirements and quality requirements of a system. However, architectural models tend to be of a high-level of abstraction, and deviations of the software architecture arise easily during the development and evolution of a system (Murphy et al. 1995). Architecture Compliance Checking (ACC) is an approach to bridge the gap between the high-level models of architectural design and the implemented program code, and to prevent decreased maintainability, caused by architectural erosion. *Architectural erosion* is "the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture" (de Silva and Balasubramaniam 2012). The opposing term, *architecture compliance*, is defined by Knodel and Popescu as "a measure to which degree the implemented architecture in the source code conforms to the planned software architecture" (Knodel and Popescu 2007).

Many tools and techniques are available to analyze a software system and to reconstruct, visualize, check, or restructure its architecture (Ducasse and Pollet 2009). In our research, we focus on tool support for static ACC, in which the software is analyzed without executing the code. Tools of this type, which we label as *static ACC-tools,* focus on the modular structure in the source code.

Although Shaw and Clements included ACC in 2006 in their list of promising areas (Shaw and Clements 2006), the adoption of ACC-tools is still limited (Gleirscher and Golubitskiy 2012, de Silva and Balasubramaniam 2012) and research is necessary to advance and improve current methods and tools (Canfora et al. 2011). Different studies have compared ACC-tools and techniques, and these studies revealed large discrepancies in terminology, approach and performance (Ducasse and Pollet 2009, Knodel and Popescu 2007, Passos et al. 2010, Pruijt et al. 2013b, de Silva and Balasubramaniam 2012).

Our research builds on these studies, but we focus on ACC support of *semantically rich modular architectures* (SRMAs). We use this term for expressive modular architecture descriptions, composed of semantically different types of modules, like layers, subsystems and components, which are constrained by rules of different types. These may be explicitly defined rules, like “module A is only allowed to use module B”, but may also be rules inherent to the semantics of a module type, like “a layer is not allowed to use higher-level layers”. In contrast to an SRMA, a semantically poor modular architecture description includes modules of only one module type and rules of only a few rule types, like the two basic types “is not allowed to use” and “is allowed to use”.

Adersberger and Philippsen (2011) consider the support of semantically rich architecture models essential for the integration of ACC in model-driven engineering. Furthermore, they make clear that support of semantically rich constructs reduces the number of rules that need to be specified, compared to semantically poorer boxes and lines models. Modules with specific semantics enhance the expressiveness of a modular architecture and support architecture reasoning. A rich set of module types provides a language to express characteristics of the modules in an architectural model. A rich set of rule types provides a language to express constraints on the modules in an architectural model. This language allows architects to define logical rules in a comparable way as expressed in regular language, without the need to translate a logical rule to one or more rules at ACC-tool level.

In a previous study, we compared eight commercial and academic ACC-tools on their support of SRMAs (Pruijt et al. 2013a). We concluded that the tested tools were providing useful support for dependency checking, but only limited support

for SRMAs. Five tools supported no semantic differences between modules. The other three tools provided some specific kind of support of layers, components or facades, but none provided extensive support for more than one type. Furthermore, all tools restricted rule support to dependency rules only, and to simple rule types, like “is allowed to use”, “is not allowed to use”, or “must use”. More complex rules were not supported explicitly and in many cases one logical rule required the combination of several rules to be specified in the ACC-tool. Consequently, a gap has to be bridged between architecture design of SRMAs and ACC tools, with as potential disadvantages, loss of architectural rules, reduced traceability, reduced overview, and reduced productivity.

In this study, we focus on the following research question: *How can support be provided for SRMAs in the context of static ACC?* To answer this question, we followed a process of design research (Peffers et al. 2008). Iteratively we identified requirements, studied existing tools, designed a metamodel, developed and tested an open-source ACC-tool as prototype, and we applied this tool during ACC’s on professional systems. These iterations spanned three consecutive years in which groups of students in computer science participated.

The contribution of this study is twofold. First, we present a metamodel for extensive support of SRMAs in the context of static ACC. The metamodel adds to the knowledge base and may be used to enhance existing tools, or to develop new approaches. Second, we introduce an open-source implementation prototype of the metamodel, which illustrates the feasibility of the metamodel.

The next section of this chapter describes and illustrates the concept “semantically rich modular architecture”, and it introduces a classification of common module types and common rule types. We use these types in our research to concretize requirements to SRMA support. Section 3.3 introduces our metamodel for SRMA support in the context of static ACC. Section 3.4 presents the prototype implementation of our metamodel concisely. Section 3.5 compares the outcome of our study to related work, and Section 3.6 concludes this study and addresses future work.

3.2 Semantically Rich Modular Architectures

According to Perry and Wolf, software architecture "provides the framework within which to satisfy the system requirements and provides both the technical and managerial basis for the design and implementation of the system" (Perry and Wolf 1992). Static ACC does not cover the full width of software architecture, but only the static structure of the software (planned and implemented); in other words, the module views of architecture (Clements et al. 2010), or *modular architecture*.

A planned modular architecture should describe the modular elements, their form (properties and relationships) and rationale (Perry and Wolf 1992). Modular elements, properties and relationships, are in ACC's center of attention, and should be included in a complete compliance check. A *modular element*, or *module*, is an implementation unit of software with a coherent set of responsibilities (Clements et al. 2010). *Properties* and *relationships* express architectural rules that constrain a modules' implementation (Perry and Wolf 1992).

3.2.1 Example of an SRMA

A semantically rich modular architecture includes modules of semantically different types, while a variety of types of rules may constrain the modules. As an example of an SRMA, Figure 3.1 shows a small part of an architecture model of one of the systems at an airport. This system is used to manage the state and services of human interaction points where customers communicate with baggage handling machines, self-service check-in units, et cetera. Examples in the rest of this document refer to elements in Figure 3.1.

Figure 3.1 shows UML icons for three semantically different types of modules: packages, components and interfaces. Layers are the fourth module type in the model (indicated by lines, since layers are not supported by UML). Finally, Spring and Hibernate represent the fifth type of module in the model: external system.

UML dependency relations in this example indicate is-only-allowed-to use rules; for instance, module HiWebApp is only allowed to use the modules HiForms and HimInterface, no others. Some other rules are not visible in the diagram. For example, rules related to the layered style, like "Technology Layer is not allowed to use Interaction Layer. Other examples of not visible rules are naming rules and rules inherent to components with interfaces.

3.2.2 Common Module and Rule Types

To enable compliance checks of SRMAs, rich sets of module and rule types should be supported. In a previous study (Pruijt et al. 2013a), we presented a classification of common module types and common rule types. In this study, we use these

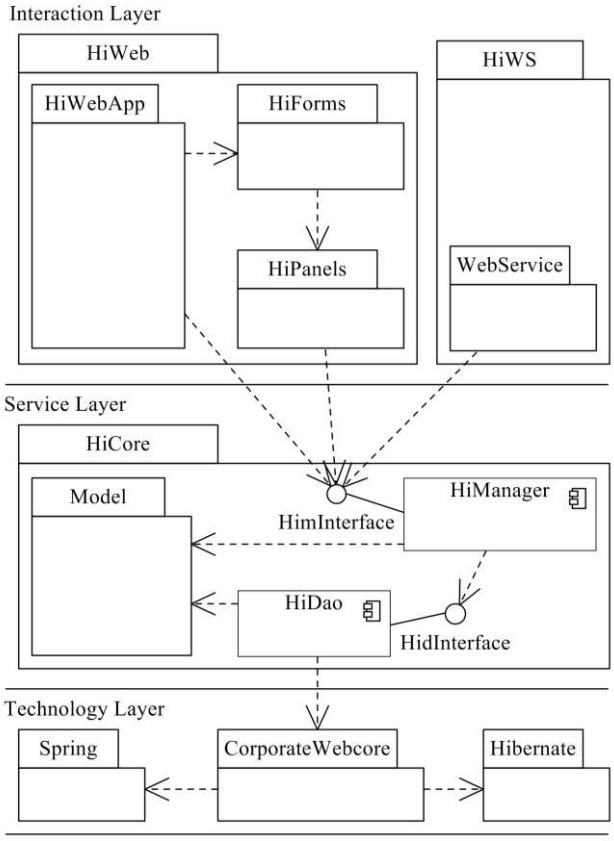

Figure 3.1: Example of an SRMA model

common types as functional requirements to SRMA support. The next sub-sections describe these common module and rule types concisely to enhance practical understanding before the metamodel is presented. For a more in-depth discussion of the common module and rule types, we refer to our previous study.

Common Module Types

SRMAs may contain modules of different types, with very different semantics. We identified five common types of modules relevant for static ACC:

Logical clusters represent units in the system design with clearly assigned responsibilities, but with no additional semantics. Comparable terms are subsystems, or packages.

Layers represent units in the system design with additional semantics. Layers have a hierarchical level and constraints on the relations between the layers. We cite Larman (Larman 2005), who summarizes the essence of a layered design as

"the large-scale logical structure of a system, organized into discrete layers of distinct, related responsibilities. Collaboration and coupling is from higher to lower layers."

Components within software architecture are designed as autonomous units within a system. The term component is defined in different ways in the field of software engineering. In our use, a component within a modular architecture covers a specific knowledge area, provides its services via an interface and hides its internals (in line with the system decomposition criteria of Parnas (1972)). Consequently, a component differs from a logical cluster in the fact that it has a Facade sub module and hides its internals. Since our definition of component is intended for modular architectures, it does not include runtime behavior as in the "component and connector view" of architecture (Clements et al. 2010).

Facades are related to a component and act as an interface as described under components. We use the term facade, referring to the facade pattern (Gamma et al. 1995), to differentiate with the Java interface, which has not exactly the same meaning as a design-level interface. A facade may be mapped to multiple elements at implementation level, like Java interface classes, exception classes and data transfer classes.

External systems represent platform and infrastructural libraries or components used by the target system. Useful ACC support includes the identification of external system usage and checks on constraints regarding their usage (Ali et al. 2012).

Common Rule Types

Modular architectures may contain rules of different types, where each rule type characterizes another kind of constraint on a module. These constraints are categorized in literature (Clements et al. 2010, Perry and Wolf 1992) as properties and relationships. Our inventory of architectural rule types, in principle verifiable by static ACC, resulted in two categories related to properties and relationships: Property rule types and Relation rule types. The identified rule types are described and exemplified in Table 3.1.

Property rule types constrain a certain characteristic of the elements included in the module and their sub modules. Clements et al. (Clements et al. 2010) distinguish the following properties per module: Name, Responsibility, Visibility, and Implementation information. We identified rule types associated to these properties and named them accordingly, except two types (Facade convention, Inheritance convention), which represent the property Implementation information.

Relation rule types specify whether a module A is allowed to use a module B. The basic types of rules are "is allowed to use" and "is not allowed to use". However, we encountered useful specializations of both basic types, which we

included in the classification. Table 3.1 shows the two included specializations of

Table 3.1: Common rule types (Ref= primary literature reference)

Category\Type of Rule	Description (D), Example (E)	Ref
Property rule types		
Naming convention	D: The names of the elements of the module must adhere to the specified standard. E: HiDao elements must have suffix DAO in their name.	1
Responsibility convention	D: All elements of the module must adhere to the specified responsibility. E: HiForms is responsible for presentation logic only.	2
Visibility convention	D: All elements of the module have the specified or a more restricting visibility. E: HiManager classes have package visibility or lower.	3
Facade convention	D: No incoming usages of the module are allowed, except via the façade. E: HiManager may be accessed only via HimInterface.	4
Inheritance convention	D: All elements of the module are sub classess of the specified super class. E: HiDao classes must extend CorporateWebCore.Dao.GenEntityDao.	1
Relation rule types		
Is not allowed to use	D: No element of the module is allowed to use the specified to-module. E: HiPanels is not allowed to use HiWS.	5
Back call ban (specific for layers)	D: No element of the layer is allowed to use a higher-level layer. E: Service Layer is not allowed to use the Interaction Layer.	6
Skip call ban (specific for layers)	D: No element of the layer is allowed to use a lower layer that is more than one level lower. E: Interaction Layer is not allowed to use the Infrastructure Layer.	6
Is allowed to use	D: All elements of the module are allowed to use the specified to-module. E: HiWebApp is allowed to use HiForms.	3
Is only allowed to use	D: No element of the module is allowed to use other than the specified to-module(s). E: HiForms is only allowed to use HiPanels.	1
Is the only module allowed to use	D: No elements, outside the selected module(s) are allowed to use the specified to-module. E: HiDao is the only module allowed to use CorporateWebcore.	1
Must use	D: At least one elements of the module must use the specified to-module. E: HiDao must use CorporateWebcore.	5

[1] (Passos et al. 2010), [2] (Larman 2005), [3] (Clements et al. 2010), [4] (Gamma et al. 1995),
[5] (Knodel and Popescu 2007), [6] (Sarkar et al. 2006)

"Is not allowed to use" (both specific for layers), and the three specializations of "is allowed to use".

3.3 SRMACC Metamodel

In this section, we introduce a metamodel to provide support for SRMAs in the context of static ACC. The metamodel, we labeled it "SRMACC metamodel", identifies, describes and relates the core concepts needed to address the following objectives regarding SRMA support. The first is to provide *basic SRMA support*, which includes the provision of sets of common module and rule types and the functionality to check rules of these types. The second is to provide *extensive SRMA support*, which adds support of the semantics of the common module and rule types. The third is to enable *configuration* of the provided support.

To enable reuse and different implementations, the metamodel has a conceptual character. For reasons of readability, the metamodel is presented in four UML class diagrams, each focusing on a different aspect. Composition associations without a name in these diagrams, should be read as "isComposedOf".

3.3.1 Definition of the Modular Architecture

The metamodel in Figure 3.2 focuses on the definition of the planned modular architecture. The model shows that an instance of a *SoftwareArchitecture* (within the context of static ACC) is composed of a set of *Modules*, the architectural elements, and a set of *AppliedRules,* constraints on the architectural elements (properties and relationships, in terms of Perry and Wolf (1992)). *AppliedRules* are characterized by their *RuleTypes*, which are grouped into *Categories* (e.g.,

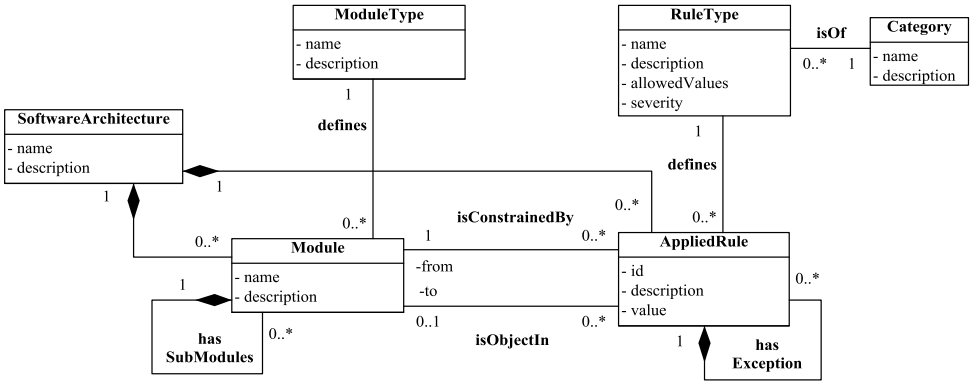

Figure 3.2: Metamodel: Definition of the Planned Modular Architecture

"Property rule" and "Relationship rule" within our classification).

Module represents instances of architectural elements, which may be composed of many sub-modules recursively. In line with the constraints of the decomposition style (Clements et al. 2010), a module can have only one parent. Basic support of SRMAs includes the provision of a set of *ModuleTypes*, like the types discussed in Section 3.2, which define the semantic properties of the modules.

AppliedRule represents instances of rules, where each instance constrains a Module; the from-module in the metamodel. An AppliedRule is of a certain *RuleType*, which defines the kind of constraint applied to the from-module. For example, the AppliedRule "HiManager may be accessed only via HimInterface" of RuleType "Facade convention" constrains Module-from "HiManager". Some types of applied rules include also a Module-to in their constraint, in which case a relationship is defined. For example, AppliedRule "HiDao must use CorporateWebcore", of RuleType "Must use", constrains Module-from "HiDao" in its use of Module-to "CorporateWebcore". Finally, support of exceptions, is also included within the metamodel. An exception rule is also an instantiation of AppliedRule, however the exception rule is linked to the original rule via association *hasException*, in order to make the exception traceable to the original rule.

3.3.2 Support of the Semantics of the Types

Inclusion in the metamodel of ModuleType and RuleType, with their properties and associations, enables support for the semantics of the provided types. First, type-specific properties may be included and configured. For example, RuleType with name "Visibility convention" defines not only the type of constraint of an AppliedRule, but it also defines, inter alia, the Category, the values allowed to include in a rule, and the severity of a violation against a rule of this type.

Second, more advanced support of the semantics of the types may be provided when logical relationships between the types are included in the model; shown in

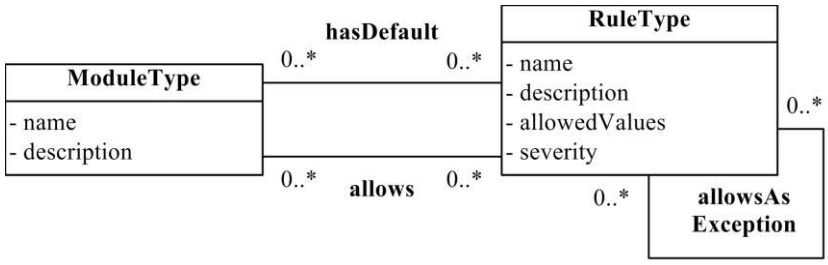

Figure 3.3: Metamodel: Support of the Semantics of the Types

Figure 3.3 as three associations. The association *hasDefault* may be used to create rules (inherent to the type of module) automatically when a module is created. For instance, when a module of type "Layer" is created, a "Back call ban" rule and a "Skip call ban" rule might be generated, based on included instantiations of association *hasDefault*.
The association *allows* may be used to present to the tool-user a list of RuleTypes, suitable to the ModuleType of the constrained module. For example, a "Back call ban" is allowed only in case of ModuleType "Layer", and a module of type "External system" is not allowed to be constrained by any type of rule, since it is not the subject of the ACC (conversely, usage of an external system may be constrained).
Finally, the association *allowsAsException* specifies for a certain RuleType, which RuleTypes are allowed as an exception to an instantiated AppliedRule. For instance, as an exception to a rule of type "Naming convention", only a rule of the same type is allowed.

3.3.3 Module Mapping

A *Module* may represent one or more implementation units of a software application. To enable ACC on various versions of the software, the metamodel in Figure 3.4 includes the association, *Module mapsTo DefinedSoftwareUnit*. An

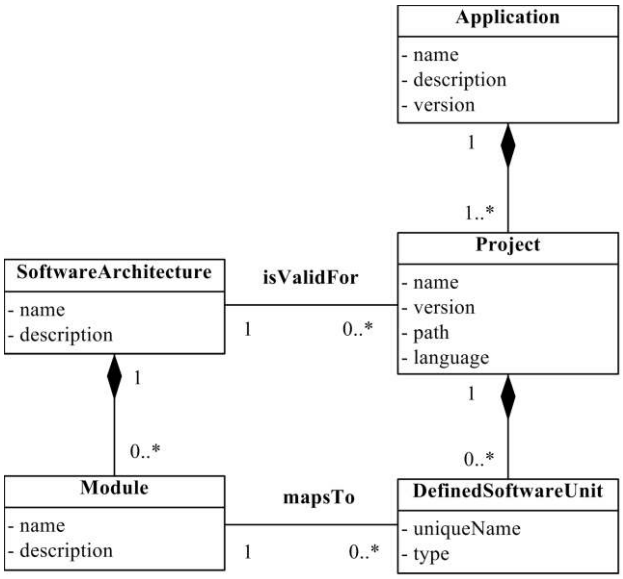

Figure 3.4: Metamodel: Module Mapping

instantiation of *DefinedSoftwareUnit* represents an implementation unit of a certain type (package, class, …) and in case of a composite unit, all its underlying units. To be able to find the unit when an ACC is performed, attribute uniqueName needs to be set with a string in the form of a path-and-name-combination or in the form of a regular expression. At this point, support to the tool-user is desirable, which may be provided based on analysis data of the current version of the software.

The metamodel in Figure 3.4 includes also the basics for ACC support of complex *Applications*, which are subdivided in technical *Projects*. Each project may have its own class path and programming language and possibly its own SoftwareArchitecture. Our metamodel also features that a SoftwareArchitecture with the same sets of Modules and AppliedRules may be reused in different projects. In that case, only the mapping will differ per project.

3.3.4 Compliance Checking

The metamodel in Figure 3.5 shows the concepts needed for the actual compliance check between the planned modular architecture and the implemented architecture. The planned architecture is composed of Modules and AppliedRules. The implemented modular architecture, including all the code-types in the software, relevant properties of these types and the dependencies between the types, is represented by *AnalyzedSoftwareUnit* and *Dependency*. In dependency and violation reports it is useful to include the type of the dependency (Pruijt et al.

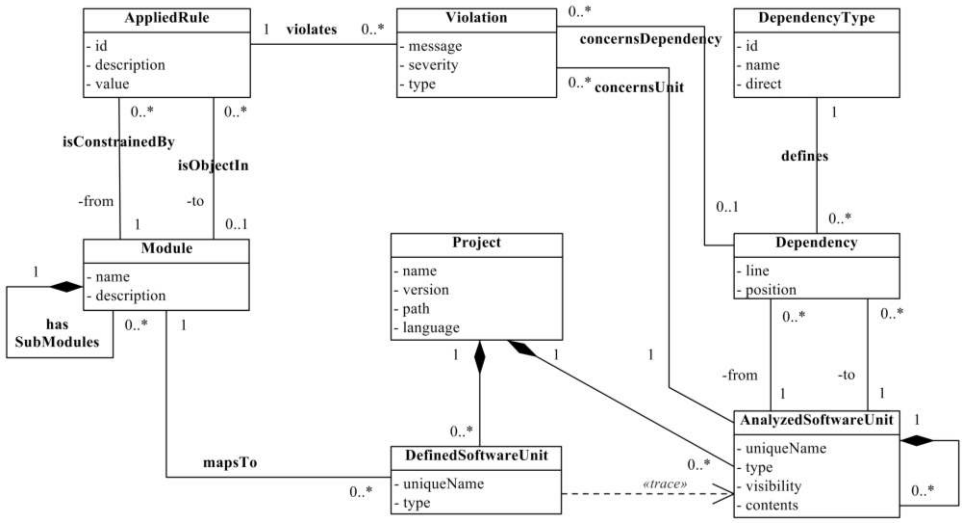

Figure 3.5: Metamodel: Compliance Checking

2013b). Reason why *DependencyType* is included, which stands for the set of dependency types. It enables a standardized presentation of these types to the tool-user, in forms and reports.

An instance of *Violation* represents an infringement of an AppliedRule by an AnalyzedSoftwareUnit; for instance, caused by a forbidden Dependency. One AnalyzedSoftwareUnit may include many code constructs that infringe the same or different AppliedRules. Each infringement is registered as a separate Violation, to enable detailed violation reporting.

At the beginning of a compliance check, the instantiations of AnalyzedSoftwareUnit and Dependency with their mutual associations need to be provided by a code-analysis process. Next, each AppliedRule can be checked, based on the traced links between the Modules related to the AppliedRule in the planned architecture and the AnalyzedSoftwareUnits in the implemented architecture. The metamodel contains the data to check AppliedRules of all the common RuleTypes in our classification. Since these RuleTypes focus on different constraints, the required data and behavior to check an AppliedRule differ per RuleType.

3.4 SRMACC Prototype

We have validated the feasibility of our approach to provide SRMA support in the context of ACC through a prototype implementation, a test of this prototype, and pilot applications of this prototype. Based on our notion of SRMA-support, we have iteratively designed, developed and applied an open source ACC-tool, named HUSACCT (HU Software Architecture Compliance Checking Tool). These iterations spanned three consecutive years. The first year, we focused on layered architectures, the second year on the provision of all the common module and rule types in the classification, and the third year on extensive support of these types. Each iteration, we used the metamodel to consider, discuss and improve our approach.

Students in computer science participated in the project, of which the results, including an introduction video, are attainable via http://husacct.github.io/HUSACCT/. HUSACCT has been developed in Java and analyzes Java and C# code. The tool provides support to define planned SRMAs, to analyze implemented architectures, and to execute conformance checks. We are using the tool to perform ACCs on professional systems, but we are using the tool also in courses on software architecture to introduce the students in architecture reconstruction, and compliance checking. We are continuing our work on the tool to improve on issues like architecture visualization, accuracy, and scalability.

Support of SRMAs conform the metamodel does not have to be implemented in the same way as in HUSACCT. For example, the presentation to the user may vary. HUSACCT supports the definition of the planned architecture via a GUI-form, though support via an architecture diagram editor is possible too. As another example, the outcome of a conformance check may be presented in terms of violations, but also in terms of Murphy's Reflection Model (Murphy et al. 1995) (convergence, divergence and absence).

3.4.1 Metamodel Implementation

Definition of the Modular Architecture

Figure 3.6 shows the view "Define Architecture", used for the creation and maintenance of the planned modular architecture; in this case of the example system depicted in Figure 3.1. The panel "Module Hierarchy" shows *Modules* of different *ModuleTypes*: Layers (e.g., Interaction Layer), Logical clusters (e.g., HiWeb), Components with Facades (e.g., HiManager with HimInterface), and External systems (e.g., Hibernate).

The panel "Rules" shows two generated *AppliedRules* attached to layer "Service Layer". These two rules are of the *RuleTypes* "Back call ban" and "Skip call ban". Existing rules can be edited and new rules can be specified in a separate panel that pops up when the Edit or Add-button is activated. Exceptions to a rule are also

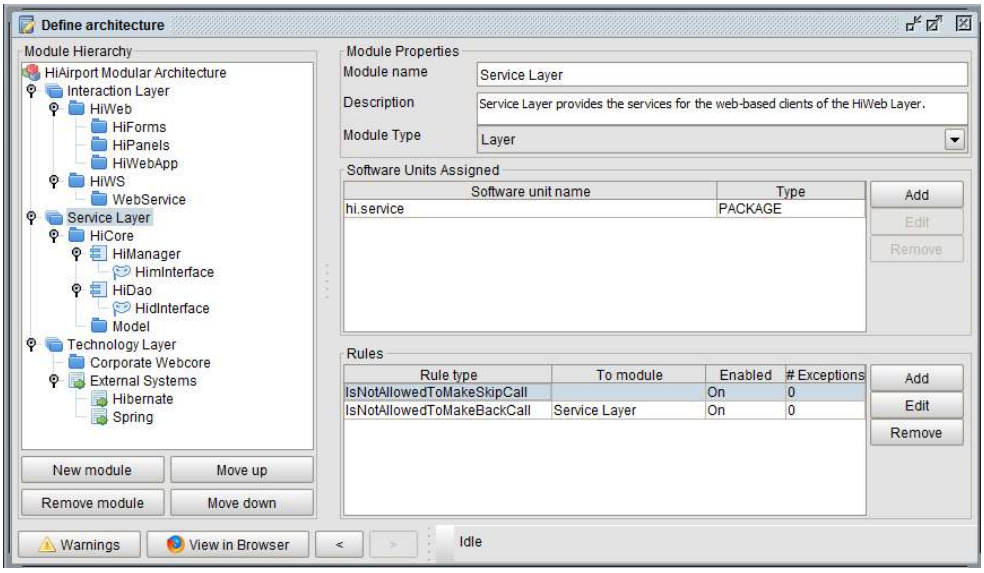

Figure 3.6: HUSACCT Define Architecture view (with as case the example depicted in Figure 3.1)

specified in this pop-up panel. To enable traceability, an exception rule is linked to the main rule, as shown in the metamodel by association AppliedRule *hasException*.

Support of the Semantics of the Types
Extensive support of the semantics of the module and rule types is provided in several ways. First, when a rule is created, only rule types are selectable, which suit the type of the constrained module, (association ModuleType *allows* RuleType in the metamodel). Second, when an exception is created, only rule types are selectable, which suit to the type of the main rule (association RuleType *allowsAsException* in the metamodel). Third, when a module is created, applied rules inherent to the module type will be created automatically (association ModuleType *hasDefault* RuleType in the metamodel). Here, the support is made configurable. For example, to configure that by default layers are allowed to skip call, but not to back call. Fourth, when a module is created of type component, a sub-module of type facade is created automatically, in line with our definition of component.

Module Mapping
Mapping *Modules to DefinedSoftwareUnits* is supported in panel "Software Units Assigned" within view "Define Architecture". In the example in Figure 3.6, package "service", an *AnalyzedSoftwareUnit* within the analyzed code, is assigned to Module "Service Layer". Available software units in the analyzed code are shown and selectable when the button "Add" is activated.

Compliance Checking
HUSACCT is able to check *AppliedRules* of eleven different *RuleTypes*. The result of a conformance check are presented in a GUI-browser, in reports, and in diagrams. Conceptually, conformance checks are executed in line with the SRMACC metamodel, but technically, there are differences. An important one is that the analyzed code in HUSACCT is stored conform the FAMIX model (Tichelaar et al. 2000). When needed, the concepts *AnalyzedSoftwareUnit* and *Dependency* in the metamodel are extracted from the data in the FAMIX model.

3.4.2 SRMA Test of HUSACCT

As part of a previous study (Pruijt et al. 2013a), we have designed and implemented a test to assess ACC-tools on their SRMA support. The test includes all common module and rule types from our classification. HUSACCT has been tested with the same testware. The test results demonstrate that HUSACCT provides explicit support for all the module types and for eleven of the twelve rule types (the rule type "Responsibility convention" is not supported, since it requires

human interpretation). However, graphical support and support of "External systems" is limited, currently.

3.4.3 Pilot Applications of HUSACCT

At the end of each of the three development iterations, we performed ACCs with our tool on professional systems at governmental and commercial organizations. In total, six different business information systems of four organizations were subject of an ACC, which we performed with the students participating in the project. The ACCs have yielded interesting results for the customer organizations and have been important for our research. This way, we were able to test our concepts and the tool's performance in practice, which resulted in new insights and new requirements for the next development iteration of our metamodel and tool.

Some general findings are of interest here. First, semantically rich module types were present in all cases; a confirmation of the relevance of rich sets of ModuleTypes in ACC. Layers dominated the modular architecture of all six case systems, while internal components with access restricting facades were included in two case systems. Second, we encountered and tested rules of nine different rule types; a confirmation of the relevance of rich sets of RuleTypes in ACC. Third, the customers appreciated the introduction of ACC in their organization, even though in five of the six cases violations were detected (up to 1500).

3.5 Related work

Other studies on ACC have mentioned or proposed the inclusion of support for a specific semantic module type; for instance for layers (Passos et al. 2010), components (Adersberger and Philippsen 2011, Knodel and Popescu 2007), or external systems (Ali et al. 2012). However, to the best of our knowledge, other studies on ACC have provided neither a comprehensive set of requirements regarding SRMA support, nor a foundational metamodel to address these requirements. Moreover, no metamodel on ACC (with or without SRMA support) has been published before, which is as comprehensive and detailed as our metamodel, which enables support for four modular styles (Clements et al. 2010): the decomposition style, uses style, layer style, and generalization style. Koschke (Koschke 2010) has presented an interesting metamodel on ACC according to the Reflection Model approach (Murphy et al. 1995). The concepts and associations in this metamodel can be mapped to the concepts in our metamodel, but compared to our SRMACC metamodel, Koschke's model is very abstract, with a smaller set of concepts and without attributes. Furthermore, it is restricted to dependency rules only, and it does not enable differentiations of module and rule types. Other studies

(e.g., (Koschke and Simon 2003, Rahimi and Khosravi 2010)) present metamodels with even fewer concepts, since they focus on one specific aspects of ACC.

In a previous paper (Pruijt et al. 2013a) we reported on the results of an SRMA-test on eight academic and commercial ACC-tools. We demonstrated that the SRMA support of these tools was limited: up to explicit support of the semantics of only one module type and up to explicit support of only a few rule types. Consequently, we concluded that all eight tools could improve their support of SRMAs. The same SRMA test has been used to test HUSACCT, as described in the previous section. The test results show that extensive SRMA support is possible on the base of the SRMACC metamodel.

Support of Common Module Types

Five of the eight tested tools in our previous study were not at all supporting semantic differences between modules. Three other tools[1] were providing some kind of support for layers, components and facades. SAVE supported the graphical definition of subsystems, layers, components and interfaces, but provided no support of the semantics of these module types. Sonargraph Architect supported the facade pattern and imposed a "Facade convention" rule on defined interfaces. Structure101 supported the concept of layering by imposing "Back call ban" and "Skip call ban" rules on vertically positioned modules. Compared to Sonargraph Architect and Structure101, our approach adds combined support (basic and extensive) for all common module types, and in a consistent way, which allows extension of the set of types. Furthermore, it adds configuration options to tune the semantic support.

Support of Common Rule Types

All eight tested tools in our previous study restricted rule support to dependency rules only, and to simple rule types, like "is allowed to use", "is not allowed to use", or "must use". Compared to these tools, our approach adds explicit support for complex dependency rules and for property rules, including traceable exceptions. These additions are relevant. Rules of the added types are used in practice, like "Naming convention" (Woods and Rozanski 2010), and may even

[1] SAVE - version 1.7 - iese.fraunhofer.de;
Sonargraph Architect - version 7.0 - hello2morrow.com;
Structure101 - version 3.5 - structure101.com.

constitute a significant part of the total rule set, as in case of rule type "Is only allowed to use" (Terra and Valente 2009).

3.6 Conclusion

Architecture compliance checking (ACC) relies on the support of tools to compare the planned architecture with the implemented architecture. We focused our research on support of semantically rich modular architectures (SRMAs) in the context of static ACC. In a previous study, we studied eight ACC-tools and concluded that the support of SRMAs was limited. In this chapter, we have presented the SRMACC metamodel for extensive support of SRMAs in the context of static ACC. The metamodel provides the fundamental concepts for: a) the definition of the planned modular architecture, including common module and rule types; b) extensive semantic support of these types; c) module mapping; and d) conformance checking. Therefore, the metamodel may be helpful to enhance existing tools or to develop new approaches. We have validated the feasibility of our approach and metamodel through an open source prototype implementation, a test of this tool (HUSACCT), and pilot applications of this tool.

Future work on SRMA support in the context of static ACC includes ongoing improvement of our HUSACCT prototype, case study research, research on visualization techniques of SRMAs in the context of ACC, research on improvement advice on planned SRMAs, and research on the inclusion of support of other module views and architectural patterns.

In conclusion, our study shows that extensive SRMA support is possible in the context of static ACC. SRMA support widens the scope of ACC and enhances the architectural process. Furthermore, we believe that SRMA support may contribute to the adoption of ACC and consequently to the effectiveness of software architecture in practice and education.

Chapter 4

HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types

Architecture Compliance Checking (ACC) is an approach to verify the conformance of implemented program code to high-level models of architectural design. Static ACC focuses on the module views of architecture and especially on rules constraining the modular elements. This chapter presents HUSACCT, a static ACC tool that adds extensive support for semantically rich modular architectures (SRMAs) to the current practice of static ACC tools. An SRMA contains modules of semantically different types, like layers and components, which are constrained by rules of different types. HUSACCT provides support for five commonly used types of modules and eleven types of rules. We describe and illustrate how basic and extensive support of these types is provided and how the support can be configured. In addition, we discuss the internal architecture of the tool.

4.1 Introduction

A*rchitecture compliance*, is "a measure to which degree the implemented architecture in the source code conforms to the planned software architecture" (Knodel and Popescu 2007). Architecture Compliance Checking (ACC) is an approach to bridge the gap between the high-level models of architectural design and the implemented program code. Static ACC does not cover the full width of software architecture, but only the static structure of the software (intended and implemented); in other words, the module views of architecture (Clements et al. 2010), or *modular architecture*. An intended modular architecture should describe the modular elements, their form (properties and relationships) and rationale, where properties and relationships express architectural rules that constrain a modules' implementation (Perry and Wolf 1992). Modular elements, properties and relationships, are in ACC's center of attention.

Although Shaw and Clements include ACC in 2006 in their list of promising areas (Shaw and Clements 2006), the adoption of ACC-tools is still limited (de Silva and Balasubramaniam 2012). With our research, we intend to contribute to the advancement of current methods and tools. We have focused on ACC support of *semantically rich modular architectures* (SRMAs). We use the term SRMA for an expressive modular architecture description, composed of semantically different types of modules (e.g., layers, subsystems, components), which are constrained by different types of rules, such as basic dependency constraints, constraints related to layers, naming constraints. In practice and literature, many architectures can be labeled as SRMA, since they contain modules with different semantics.

In the last four years, we have iteratively identified requirements regarding SRMA support, studied existing ACC tools, designed a metamodel, developed, and tested HUSACCT, and we applied this tool during ACC's on professional systems. In a first publication (Pruijt et al. 2013a), we presented requirements to SRMA support, and we compared eight commercial and academic ACC-tools on basis of the requirements. We concluded that only limited support was available for SRMAs. Furthermore, that solutions were needed to bridge the gap between modular architectures in software architecture documents on one side, and module and rule models in ACC-tools on the other side.

In a second publication (Pruijt and Brinkkemper 2014), we presented the SRMACC metamodel, whereof the central part regarding SRMA-support is included in Figure 4.1. It includes the concepts, their attributes, and associations, relevant to this chapter. As shown in the figure, an SRMA contains *Modules* of

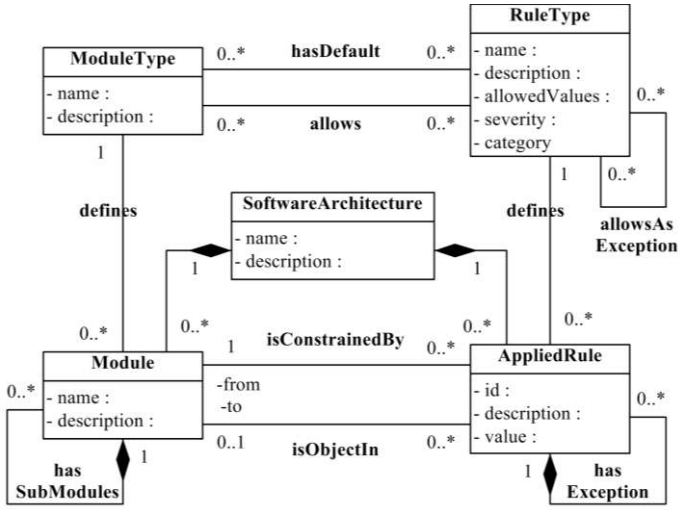

Figure 4.1: Part of SRMACC metamodel

different *ModuleTypes*, where *AppliedRules*, each of a certain *RuleType*, may constrain the *Modules*. For a detailed discussion of the complete metamodel, we refer to (Pruijt and Brinkkemper 2014).

This chapter describes and illustrates how HUSACCT provides extensive and configurable SRMA support. The remainder of this chapter is structured as follows. Section 4.2 describes and illustrates the functionality of HUSACCT with the focus on SRMA support. As running example, we use the internal architecture of the tool itself, as it is a suitable example of an SRMA, and it helps to explain how we addressed the most important design challenges. Section 4.3 describes related work, and Section 4.4 concludes with the status and outlook of our tool.

4.2 HUSACCT

HUSACCT (HU Software Architecture Compliance Checking Tool) is a tool that provides support to analyze implemented architectures, define intended architectures, and execute conformance checks. Browsers, diagrams and reports are available to study the decomposition style, uses style, generalization style and layered style (Clements et al. 2010) of intended architectures and implemented architectures. HUSACCT is free-to-use and open source. It has been developed in Java and analyzes Java and C# source code. The executable and source code are downloadable at http://husacct.github.io/HUSACCT/. An introduction video and documentation are accessible at the same site.

In HUSACCT, an ACC starts with the definition of the modules and rules in the intended architecture. Next, the intended modules are mapped to the implemented software units. Finally, the conformance of the implemented architecture to the intended architecture can be validated. The following subsections follow these steps and explain how HUSACCT provides basic, extensive, and configurable SRMA support. Thereafter, we describe how we addressed some design challenges in the tool's architecture.

4.2.1 Rich Sets of Module and Rule Types

Basic SRMA support includes the provision of rich sets of module and rule types and the functionality to check rules of these types. In our first SRMA-publication (Pruijt et al. 2013a), we identified common module and rule types and discussed their grounding in literature. During the development of HUSACCT, we aimed at support of these common types. Currently HUSACCT provides support for five common ModuleTypes and eleven common RuleTypes.

The module and rule types are used in view "Define Intended Architecture", shown in Figure 4.2. This view supports the creation and maintenance of the intended modular architecture. The panel "Module Hierarchy" shows the

ModuleTypes currently supported: Component (e.g., *Module* Analyse), Interface (e.g., Interface<Analyse>), Layer (e.g., Presentation), Subsystem (e.g., Common), and External system (e.g., ExternalSystems).

As case, the main part of the architecture of HUSACCT itself is presented. At top-level five components are visible, which all have a layered design internally. As example, three layers are shown within component Analyse. This component is responsible for the analysis of the implemented architecture. The domain layer is responsible for the analyzed data and is designed as a component, with an interface to hide its internals.

The panel "Software Units Assigned" shows that a package and a class are assigned to module Analyse. Inherently, all software units assigned to its submodules are assigned as well. How implemented software units must be assigned to intended modules differs from system to system in practice. Consequently, manual work is required. To enhance the efficiency and accuracy of this work, analyzed software units are made selectable. Once the software units are assigned, defined architecture diagrams can be created, like the ones in Figure 4.4 and 4.5, in which defined modules and dependencies (the black, dashed lines) are included.

The panel "Rules" shows that four *AppliedRules* of three different *RuleTypes* are constraining module Analyse. A new rule, together with its exceptions, can be specified in a separate panel that pops up when the Add-button is activated. An exception rule is part of a main rule, as visible in the metamodel. That way it is easy to maintain an overview. For example, the first rule of component Analyse is of type "Facade convention", which bans usage of the component, other than via its

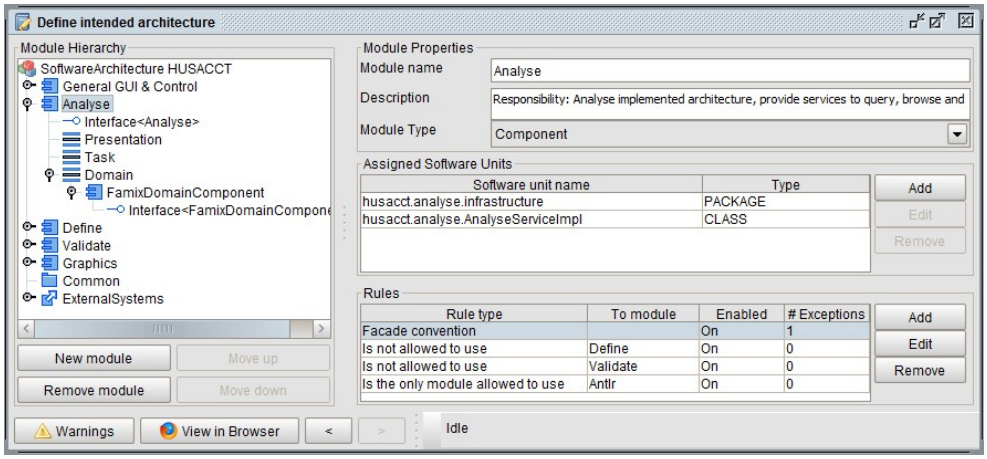

Figure 4.2: Define intended Architecture, with as case the SA of HUSACCT itself

interface. Except for a module in component General GUI & Control, that acts as broker.

4.2.2 Extensive Semantic Support

Extensive semantic support of the module types and rule types prevents inconsistencies in the defined architecture, and it saves work and time. For example, in case of HUSACCT's intended architecture, most rules and all the interfaces are added automatically. HUSACCT provides extensive SRMA support in the following ways.

First, when a rule is created, only rule types are selectable which are allowed for the type of the constrained module. For example, in case of module type Layer, all rule types are allowed, except a rule type specific for Components, and rule type "Is allowed to use", which is reserved for exceptions. The list of allowed rule types for module type Layer is shown in Figure 4.3.

Second, when an exception rule is created, only rule types are selectable which suit to the type of the main rule. For instance, an exception to a rule of type "Facade convention" may only be of type "Is allowed to use".

Third, when a module is created of type Component, a sub-module of type Interface is created automatically; in line with our definition of component.

Fourth, when a module is created, zero, one or more applied rules will be created, based on the associated default rule types. For example, in case of module type Component, an accompanying default rule of type "Facade convention" is generated automatically.

4.2.3 Configurable Support

ACCs with other tools taught us that non-configurable tool support may result, in

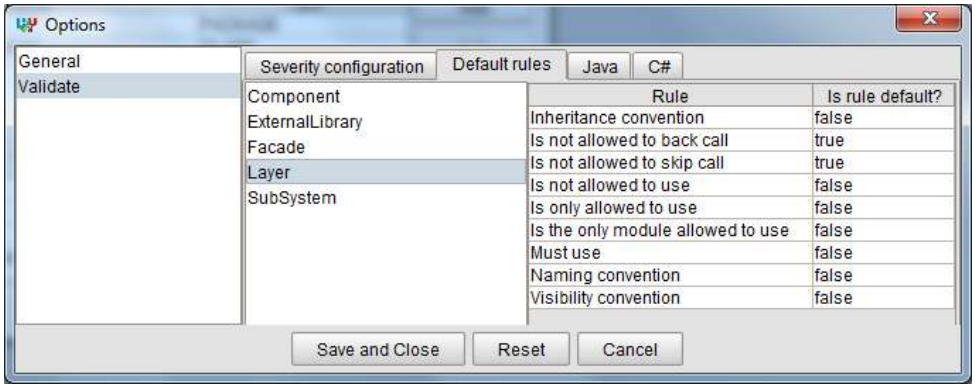

Figure 4.3 HUSACCT: Configuration of default rule types

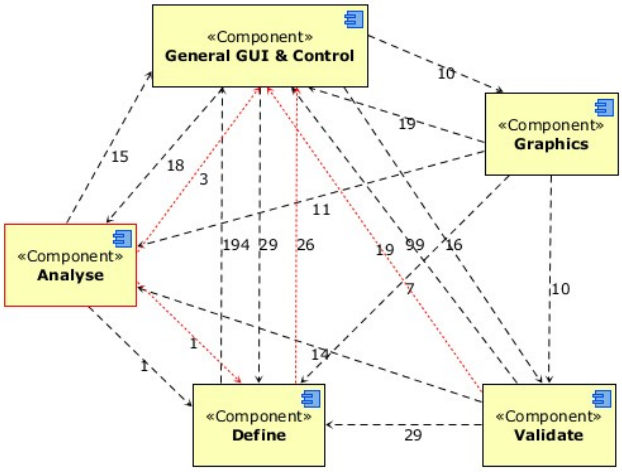

Figure 4.4 Intended architecture: Top-level components

certain situations, in invalid violation messages. Reason why we made all rules accessible and incorporated the following configuration options: 1) generated default rules may be disabled (just as user defined rules); 2) exceptions to generated default rules may be specified (just as exceptions to user-defined rules); 3) tool-users may configure the default rule types per module type. Figure 4.3 serves as an example for the third option. It shows that two rule types are assigned as default for module type "Layer". These two rule types together enforce a strict layered model. However, a tool-user is able to configure that in his software architecture a relaxed layered model is standard. Consequently, only an "Is not allowed to back call" rule will be generated when a module of type Layer is added.

4.2.4 Conformance Checking

Within HUSACCT, the component Validate is responsible for conformance checking. The results of a conformance check are presented in a GUI-browser, in reports, and in diagrams.

Figure 4.4 and 4.5 show *Intended architecture diagrams* with the results of a conformance check on the rules of the intended architecture in Figure 4.2. Violations are shown as red, dotted lines, where the number indicates the number of violations between the two related modules. Details about these violations (like rule type, involved classes, or dependency type) are shown when a line is selected. For example, of the 194 dependencies in Figure 4.4 from Define to General GUI & Control (the black, dashed line), 26 are violating (the red, dotted line). In this case, all are violating a rule of type "Facade convention". It concerns dependencies to classes within component Analyse, which pass the interface.

Figure 4.5 shows the violations between the layers within the component Analyse. Five back call violations are visible from layer Task to Presentation. The other 17 violations, from Task to Domain, are violations against a "Facade convention" rule. These violations from Task to Domain are shown in more depth in Figure 4.6, an *Implemented architecture diagram* (zoomed-in on these two layers; some classes and packages are hidden). It shows that two implemented classes make use of the service implementation class and pass the interface class of the FamixDomainComponent. Even worse are the violating dependencies from package analyser directly to package famix.

4.2.5 Design Challenges

The development of HUSACCT started after a phase of requirement analysis, in which two organizations were involved; the Dutch Tax Administration and InfoSupport. Based on the requirements and the team structure, we had to address design challenges, like: 1) the sets of module and rule types had to be extendible; 2) the tool should work in GUI mode, but also in batch (e.g., daily build process); 3) six development teams had to work concurrently (students in computer science contributed to the development during the first two releases); 4) the set of supported OO programming languages had to be extendible.

To address the first challenge, the SRMACC metamodel was developed, and during the implementation of the concepts, hard-wired dependencies to individual types were prevented as much as possible; for example, by usage of the strategy pattern.

To address the second and third challenges, HUSACCT's software is divided into five components, where each component covers a knowledge area. The

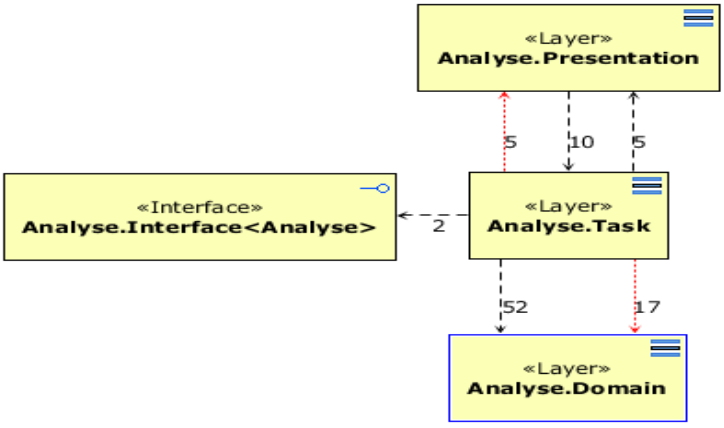

Figure 4.5: Intended architecture: Analyse component

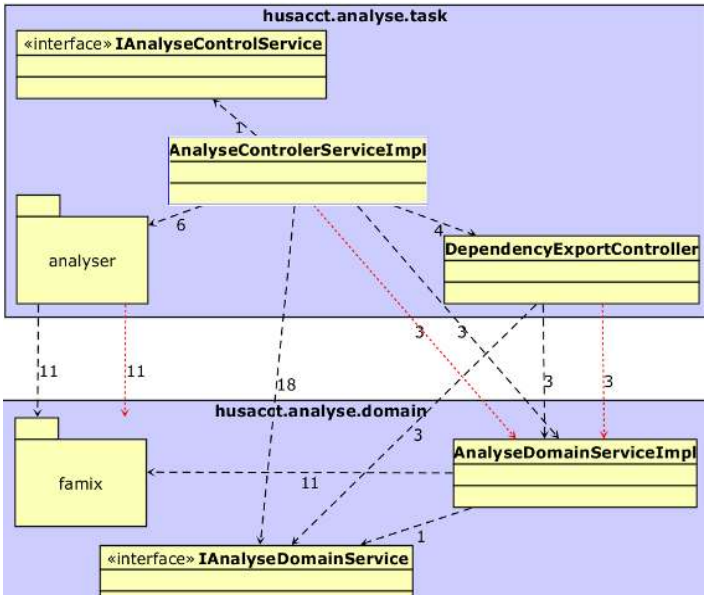

Figure 4.6: Implemented architecture: Analyse package

components hide their internals, offer services to other components, and exchange data only via data transfer objects. That way, services may be activated via a GUI or in batch (challenge 2), and each component can be assigned to a separate development team (challenge 3).

To address the fourth challenge, two design decisions were taken. First, ANTLR (www.antlr.org) was selected to read and process the source code, because grammars are available for many programming languages. Second, the FAMIX model (Tichelaar et al. 2000) was selected to store analyzed code data internally, in a language independent format. Since, after the analysis, all services acquire their data from the FAMIX model, language dependencies are minimized.

4.3 Related Work

In a previous study (Pruijt et al. 2013a), we reported on the results of an SRMA-test on eight academic and commercial ACC-tools. We concluded that the tested

tools were providing useful support for dependency checking, but only limited support for SRMAs.

Five of the eight tested tools in our previous study were providing only one type of module. Three other tools[1] were providing more types of modules, but only with limited support of their semantics. One tool, SAVE, supported the graphical definition of four module types, but provided no support of their semantics. The two other tools provided semantic support for one type of module: Sonargraph Architect for Interface; and Structure101 for Layer. Compared to these tools, HUSACCT adds semantic support for all its types of modules in a consistent way, which allows extension of the set of module types. Furthermore, it adds configuration options to tune the semantic support.

All eight tested tools in our previous study restricted rule support to dependency rules only, and to simple rule types. Compared to these tools, HUSACCT adds support for property rules (e.g., "Naming convention", "Inheritance convention"), complex dependency rules (e.g., "Is only allowed to use", "Is the only module allowed to use), and exceptions (exceptions are presented as parts of a main rule, not as independent rules).

4.4 Status and Outlook

HUSACCT provides support to analyze implemented architectures, define intended architectures, and execute conformance checks. HUSACCT distinguishes itself from other ACC tools in its extensive and configurable support of rich sets of module and rule types.

HUSACCT is a free-to-use open source tool, but it is not intended to compete licensed tools. In contrast, we want to contribute to the adoption and quality of ACC. HUSACCT is intended for: 1) introduction of ACC within software development organizations; 2) practical support in courses on software architecture. We use the tool to introduce our students in software architecture, architecture reconstruction, and compliance checking. The tool helps them to relate abstract models to code and to understand the different types of modules and rules.

[1] SAVE - version 1.7 - iese.fraunhofer.de;
Sonargraph Architect - version 7.0 - hello2morrow.com;
Structure101 - version 3.5 - structure101.com.

HUSACCT is in its fourth year of development and each year we performed ACCs with our tool on open source systems and professional systems. The ACCs yielded interesting results for customer organizations and helped us to test and improve the tool. Furthermore, they confirmed the relevance of SRMA support, since in many cases semantically rich module types were present.

Last year, we have worked on the improvement of the accuracy, performance, and usability of the tool, and with considerable results. For instance, analysis and processing time of the source code of HUSACCT version 1.0 (136K lines of code) was reduced from hours in version 2.0 to less than 20 seconds in version 3.2. Future work will focus at first on further improvements of existing functionality, such as the architecture diagrams. Thereafter, we plan to extend the tool with more options for ACC and architecture reconstruction.

In conclusion, HUSACCT shows that extensive and configurable SRMA support is possible. SRMA support widens the scope of ACC and enhances the architectural process. Furthermore, we believe that SRMA support will contribute to the adoption of ACC and consequently to the effectiveness of software architecture in the practice of software engineering.

Chapter 5

The Accuracy of Dependency Analysis in Static Architecture Compliance Checking

Architecture Compliance Checking (ACC) is an approach to verify conformance of implemented program code to high-level models of architectural design. Static ACC focuses on the modular software architecture and on the existence of rule violating dependencies between modules. Accurate tool support is essential for effective and efficient ACC. This chapter presents a study on the accuracy of ACC tools regarding dependency analysis and violation reporting. Ten tools were tested and compared by means of a custom-made benchmark. The Java code of the benchmark testware contains 34 different types of dependencies, which are based on an inventory of dependency types in object oriented program code. In a second test, the code of open source system FreeMind was used to compare the ten tools on the number of reported rule violating dependencies and the exactness of the dependency and violation messages. On the average, 77 percent of the dependencies in our custom-made test software were reported, while 72 percent of the dependencies within a module of FreeMind were reported. The results show that all tools in the test could improve the accuracy of the reported dependencies and violations, though large differences between the ten tools were observed. We have identified ten hard-to-detect types of dependencies and four challenges in dependency detection. The relevance of our findings is substantiated by means of a frequency analysis of the hard-to-detect types of dependencies in five open source systems.

5.1 Introduction

Software architecture is of major importance to achieve the business goals, functional requirements, and quality requirements of a system. In practice, a variety of architectural models is used to describe how systems are structured and how the components interact. However, the models tend to be of a high-level of abstraction, and deviations of the software architecture arise easily during the development and evolution of a system (Murphy et al. 1995). Architecture Compliance Checking (ACC) is an approach to bridge the gap between the high-level models of architectural design and the implemented program code, and to prevent architectural erosion (de Silva and Balasubramaniam 2012). Knodel and Popescu (2007) defined architecture compliance as "a measure to which degree the implemented architecture in the source code conforms to the planned software architecture". The terms architecture compliance and architecture conformance are both used in literature.

Many tools and techniques are available to analyze a software system and to reconstruct, visualize, check or restructure its architecture (Ducasse and Pollet 2009). In our study, we focus on tools supporting static ACC, which analyze software without executing the code. These tools, which we label as static ACC-tools, focus on the modular structure in the source code. The tools identify structural elements, such as packages and classes, and use-relations between these elements, such as an invocation of a method or access of an attribute. To support ACC, the tools provide facilities to: a) define modular elements and rules restricting these elements and their relationships; b) check the compliance to these rules; and c) report violations to these rules. For example, a tool should report a violation if a method-call in the code from class A to B corresponds with a dependency from module X to module Y in the planned architecture, while a rule exists that forbids such a dependency.

Although ACC-tools predominantly check for the same kind of inconsistencies between the implemented and intended modular architecture, only a few studies have compared these tools. Previous studies have identified large differences in terminology and approach (Knodel and Popescu 2007, Passos et al. 2010, Van Eyck et al. 2011). For instance, the study of Passos et al. (2010) identified and evaluated three techniques of static architecture checking. Furthermore, they explored the effectiveness and usability of three supporting tools by executing tests, based on a simple system with a basic architecture. Our research follows Passos et al. We aspire to contribute to the evolution of ACC, motivated by the notion that the adoption of ACC-tools is still limited (de Silva and Balasubramaniam 2012, Gleirscher et al. 2013). Further research is necessary to advance and improve current methods and tools (Canfora et al. 2011). We focus on

the effectiveness of ACC, since it is of primary interest to practitioners and researchers. The "Quality in use model" of ISO 25010 (ISO/IEC 2011) defines effectiveness as "accuracy and completeness with which users achieve specified goals". In another study, we investigated the functional completeness of ACC support; more specifically, the support of semantically rich modular architectures in the context of ACC (Pruijt et al. 2013a).

In this study, we focus on the accuracy of ACC support, which we scoped to the main question: *How accurate do ACC-tools report dependencies and violations against dependency rules?* Accuracy is relevant, since emerging trends are to use code analysis throughout the coding process (Binkley 2007), and to extract and update architectural views continuously (Canfora et al. 2011). Although static analysis is theoretically not difficult, the complexities of modern programming languages significantly impede source code analysis (Binkley 2007). Nevertheless, unlike performance, accuracy of ACC does not receive much attention. The accuracy of dependency and violation reporting is omitted in many papers on ACC approaches, e.g. (Murphy et al. 1995, Sangal et al. 2005, Bischofberger et al. 2004, Huynh et al. 2008, Koschke et al. 2009, Deissenboeck et al. 2010, Adersberger and Philippsen 2011, Haitzer and Zdun 2012), and when discussed, it is restricted to false positives only. To operationalize our main question, we decomposed it into the following research questions:

RQ1: Do ACC tools find all the dependencies between modules in the software (no false negatives)?

RQ2: Do ACC tools report all the violating dependencies in the software (no false negatives)?

RQ3: Do ACC tools report non-violating dependencies as violations (false positives)?

RQ4: Do ACC tools report the exact type and location of violations and dependencies?

RQ5: Are there types of dependencies, which proved hard-to-detect by several tools?

To answer these questions, we inventoried types of dependencies that can be established in object oriented program code. Next, we developed a custom-made test application in Java that included these types of dependencies and an accompanying test script (we will use the working title "benchmark test" to refer to this test software and test script). After completion, we used the benchmark test to assess ten ACC-tools. In addition, we selected the open source system FreeMind and used its code to examine the same tools on their ability to report dependencies and violations accurately.

The contribution of this study is threefold.

- We present two Java-based tests, the benchmark test and the FreeMind test, to assess the ability of a tool to detect dependencies of 34 different types. The testware of these tests is available on request and can be used for other types of static code analysis tools as well.
- We present the results of the tests on ten commercial and non-commercial ACC-tools with respect to the accuracy of dependency detection and the exactness of the dependency and violation messages.
- We identify ten types of dependencies that proved hard-to-detect by several tools in the tests. Furthermore, we identify challenges in dependency detection, and we substantiate the relevance of these challenges by means of analysis data of five open source systems.

This paper extends earlier work (Pruijt et al. 2013b) in which we have reported on the accuracy of dependency analysis and violation reporting of seven ACC-tools. First, we add the test results of three ACC-tools, of which two were presented at ICSE in recent years (Buckley et al. 2013, Deissenboeck et al. 2010). Second, we describe and illustrate the dependency types in the tests in more detail and provide an improved definition of indirect dependencies in the context of ACC. Third, we present more test results and explain these results more extensively. Fourth, we identify ten hard-to-detect types of dependency and four challenges in dependency detection. Fifth, we present the frequencies of the hard-to-detect dependency types in five open source systems. Sixth, we have improved the testware of the benchmark test and FreeMind test, and made both sets of testware utilizable for other researchers. Seventh, we extended the benchmark test at the point of the detection of local variables, and we retested all tools at this point.

In the remainder of this chapter, the next section provides an introduction in dependency analysis. Section 5.3 introduces the tested tools, Section 5.4 describes the method and results of the benchmark test, and Section 5.5 does the same for the FreeMind test. Section 5.6 describes method and results of a frequency analysis of hard-to-detect dependency types. Section 5.7 discusses the key findings and discusses the identified challenges in dependency detection. Section 5.8 discusses the threats to validity, and Section 5.9 relates our findings to other work. Section 5.10 concludes this chapter; it answers the research questions, summarizes the results of this study, and casts a glance at future work.

5.2 Dependency Analysis

Software architecture (SA) compliance checking covers a broad field, since software architecture "provides the framework within which to satisfy the system requirements and provides both the technical and managerial basis for the design

and implementation of the system" (Perry and Wolf 1992). Static ACC does not cover the full width of SA, but covers the modular architecture. According to Perry and Wolf (1992), this architecture should describe the modular elements, their form (properties and relationships) and rationale. In this study, we focus on the relationships between modules. *Relationships* are used to constrain how the different elements may interact or otherwise may be related. In static ACC's center of attention are *uses relations*: "Module A *uses* module B if A *depends* on the presence of a correctly functioning B to satisfy its own requirements" (Clements et al. 2010).

Dependency analysis is "the process of determining a program's dependences" (Podgurski and Clarke 1990). Various types of dependencies are distinguished in literature. Callo Arias et al. (2011) consider that all types fit into three main categories: structural dependencies, behavioral dependencies, and traceability dependencies. The category of structural dependencies, dependencies among parts of a system, is of interest to our study, since static analysis tools focus on dependencies that can be found by inspecting the source code. For instance, Lattix's LDM tool "uses a standard notion of dependency, in which a module *A* depends on a module *B* if there are explicit references in *A* to syntactic elements of *B*" (Sangal et al. 2005).

Many references of different types can be established in object oriented program code. To prepare our test, we inventoried references in Java code and classified them into types of structural dependencies. We based our classification of dependency types on professional literature on Java and on research papers distinguishing different dependency types, like (Feilkas et al. 2009, Ko et al. 2006, Terra and Valente 2009, Saraiva et al. 2010, Stafford and Wolf 2001).

5.2.1 Example of a Modular Architecture

A small modular architecture in UML notation, which will be used to illustrate the different types of dependency included in our test, is shown in Figure 5.1. In this diagram, two modules, ModuleA and ModuleB, are shown, each with two submodules. The classes in the submodules are related via associations, showing for instance that an instance of Class1 may know upmost one instance of Class2. The dependency arrows (the dashed arrows) show that ModuleA1 is allowed to use ModuleB1 and that ModuleA2 is allowed to use ModuleB. However, not all rules are visible. The following list shows the full set of relationship rules, of which the first three rules are explicitly visible in the diagram, while the last two are implicit:

- ModuleA1 is allowed to use ModuleB1;
- ModuleA2 is allowed to use ModuleB, so also both sub modules, ModuleB1 and ModuleB2;

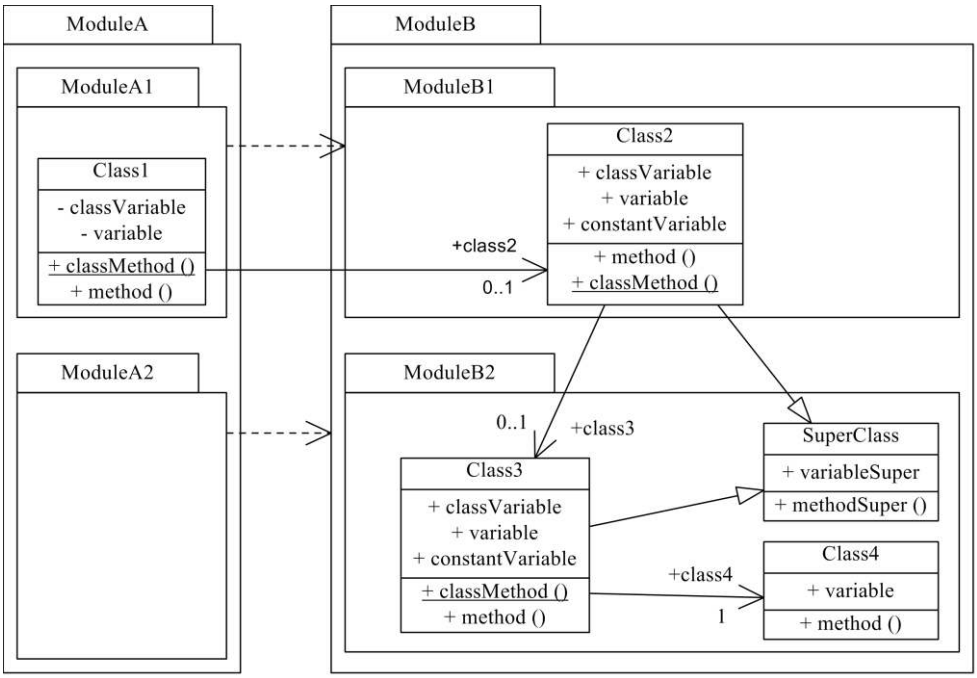

Figure 5.1: Explanatory model of a modular architecture in UML notation.

- ModuleA1 is not allowed to use ModuleB2;
- The submodules of ModuleA are allowed to use each other.
- The submodules of ModuleB are allowed to use each other.

5.2.2 Structural Dependency Types in Object Oriented Code

Many references of different types can be established in object oriented code. To prepare our test, we inventoried references in Java code and classified the structural dependencies, based on professional literature on Java, and on research papers that distinguish different dependency types, like (Feilkas et al. 2009, Ko et al. 2006, Terra and Valente 2009, Saraiva et al. 2010, Stafford and Wolf 2001). We identified six main types of dependency: Import, Declaration, Call, Access, Inheritance, and Annotation. For each main type, sub types may be defined. For instance, declarations of instance variables are distinguished from declarations of class variables, local variables, parameters, return types and from type casts.

An example of code per main dependency type is provided below. Each example contains a code construct that causes a rule violating dependency from Class2 to Class3 or to SuperClass, all in Figure 5.1.

- Import: import ModuleB.ModuleB2.Class3;
- Declaration: private Class3 class3;
- Call: variable = class3.method();
- Access: variable = class3.variable;
- Inheritance: public class Class2 extends SuperClass;
- Annotation: @Class3

5.2.3 Direct and Indirect Dependencies

In our study we have included another distinction, namely between direct and indirect dependency. In general, a dependency between two modules is direct, if the dependency relation does not involve an intermediate module. However, we use the term *direct dependency* more specifically, namely for a dependency of which the to-class (the depended-upon class) can be determined, completely based on the knowledge of the from-class (the class that contains the dependency). All six code examples above cause direct dependencies. For example, the dependency caused by the call statement in Class 2, may be traced to Class3, since variable class3 in Class 2 is declared to be of type Class3.

In general, a dependency relation is indirect, when the dependency exists transitively through an intermediate module. According to this definition, many indirect dependencies are present in program code. For example, if Class1 in Figure 5.1 contains a method that calls Class2.method(), that somewhere in a scenario calls Class3.method(), then ModuleA1 depends indirectly on ModuleB2 via ModuleB1. In static ACC, this example of an indirect dependency will not be reported as a dependency or violation, since it should result in an overload of dependencies and violations. To prevent an excess of dependencies and violations, we narrow the definition of an indirect dependency in case of static ACC as follows. An *indirect dependency* is a dependency in the from-class of which the to-class cannot be determined without the analysis of the code of another class. Such a dependency should be reported, if a code construct in the from-class has as immediate consequence that the to-class is used; for example in case of access of an inherited attribute, or in case of a call of a method that causes a dependency on the return type of the method.

In these cases, another class needs to be analyzed, or even several other classes, including super classes. The following code examples from Class1 in Figure 5.1 include a rule violating indirect dependency to Class3 or to SuperClass.

- Call: variable = class2.class3.method();
- Access: variable = class2.variableSuper;
- Inheritance: public class Class1 extends Class2;

5.3 ACC-Tools Included in the Test

Many tools are available with some of the facilities necessary to support ACC. However, our research focused on tools with explicit support of ACC. We selected publicly available tools, which were mentioned in academic work (e.g., (Ducasse and Pollet 2009, Passos et al. 2010, Adersberger and Philippsen 2011, Buckley et al. 2013)), were able to analyze Java, and provided evaluation or research licenses (two vendors rejected and one did not respond). We excluded tools that focus mainly on architecture visualization, metrics and/or architecture refactoring.

The ten tools included in our study are shown in Table 5.1, which also provides an overview of functionalities, code variants, and licensing per tool. The versions of the tools used in our tests together with an URL per tool[2], are described below Table 5.1, in the footnotes.

The tools provide their support of ACC in various ways:

- Dependometer, Macker and Sonar Architecture Rule Engine (Sonar ARE) are text-based tools, which support relation conformance rules. These tools provide HTML-based reports as output.
- dTangler and Lattix are based on the Dependency Structure Matrix (DSM) technique, complemented with text-based editors to define rules. The DSM is used to sort and select modules, to define rules, and to show dependencies and violations. Lattix is also able to visualize architectures graphically, and it provides extensive reporting facilities.
- ConQAT Architecture Analysis, JITTAC and SAVE are strictly based on the Reflexion Model technique (Murphy et al. 1995). These tools provide a graphical editor to define the intended architecture and to show violations (in terms of divergence and absence) after the evaluation. In addition, ConQAT and SAVE generate textual reports at request, supportive to consistency checks subsequent to software development activities. JITTAC aims at real-time feedback during software development, and for that reason it is tightly integrated in the Eclipse IDE. JITTAC indicates divergences to the architectural model in a diagram and in the source code editor; not only afterwards, but also the moment an inconsistency is programmed.
- Sonargraph Architect and Structure101 are diagram-based too, but these tools are not strictly based on the RM-technique. To define modules and rules, these tools provide diagrams in which the horizontal and vertical position of a module implies rules. Violations are shown in these diagrams, but textual reports are provided in addition.

Table 5.1: Characteristics of the tools in the test (√= supported)

Characteristic	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure101
General functionalities										
Dependency browsing		√	√	√	√				√	√
Dependency visualization				√	√		√		√	√
Architecture compliance checking	√	√	√	√	√	√	√	√	√	√
Architecture refactoring/simulation		√			√				√	√
Team support									√	√
Code variants										
Java	√	√	√	√	√	√	√	√	√	√
Other languages	√	√			√		√			√
Source file analysis		√[1]		√			√	√[1]	√[1]	
Compiled file analysis	√	√	√		√	√		√	√	√
Licensing										
Paid: commercial use					√		√		√	√

[1] In addition to Compiled file analysis.

[2] ConQAT Architecture Analysis - version 2011.9 - conqat.org;
Dependometer - version 1.2.5 - source.valtech.com/display/dpm/Dependometer;
dTangler - GUI version 2.0.0 - web.sysart.fi/dtangler;
JITTAC – version 0.2.0 - lero.ie/project/arc;
Lattix LDM - version 8.2.7 - lattix.com;
Macker - version 0.4.2 - sourceforge.net/projects/macker;
SAVE - version 1.7.1 - iese.fraunhofer.de;
Sonar ARE - version 3.2 - docs.codehaus.org/display/SONAR/Architecture+Rule+Engine;
Sonargraph Architect (fusion of Sotograph and SonarJ) - version 7.1.8 - hello2morrow.com;
Structure101 - version 3.5 - structure101.com.

5.4 Benchmark Test

Two separate tests were performed with the ten tools: the benchmark test, and the FreeMind test. This section describes the bench mark test and the next section the FreeMind test. Both tests were developed and improved iteratively. The first iteration of preparing, testing, and reporting was conducted with 25 students Computer Science in the course of a specialization semester "Advanced Software Engineering". Afterwards, the authors have improved the tests and tested the tools completely again in several iterations. The final test results are presented in this chapter. Furthermore, the final versions of the testware of the two tests are available on request.

5.4.1 Method

Based on the inventory of different types of dependencies, as described in Section 5.2, we distinguished six main types of structural dependencies: Import, Declaration, Call, Access, Inheritance, and Annotation. In addition, sub types were defined for the main types Declaration, Call, Access, and Inheritance. For instance, declarations of instance variables were distinguished from declarations of class variables, local variables, parameters, return types, and from type casts. In combination with the distinction between direct and indirect dependencies, we have distinguished 25 direct dependency types and nine indirect dependency types.

A test software system was designed and subsequently implemented in Java with Eclipse Indigo SR2. For each dependency type, at least one test case was implemented. Furthermore, a test script was prepared with instructions per step in the test process, and tables with test cases, with per test case information on the existing dependencies and cells for result notes.

To measure the sensitivity (also called the true positive rate, or the recall) of the ACC tools, 64 test cases in the test set were aimed at the detection of true positives and false negatives regarding dependency detection and violation reporting. In this chapter, we compute s*ensitivity* in percent, as: (number-of-true-positives/(number-of-true-positives + number-of-false-negatives)) * 100.

To measure the false positive rate of the ACC tools, 64 cases were aimed at the detection of false positives. The test code of these test cases was identical to the first 64 test cases, so dependencies to the same to-classes were contained. However, the from-classes, containing the code, were located in another package at the same hierarchical level. This way, violations of classes in the second package, based on architectural constraints defined at the first package, should be qualified as false positives. In this chapter, we compute the false positive rate in percent, as: (number-of-false-positives/
(number-of-false-positives + number-of-true-negatives)) * 100.

Several tools report violations and dependencies only at the level of from-class, to-class, without further detail. To be able to obtain reliable test results, but also to facilitate and simplify the test process, we implemented a separate from-class per test case. Furthermore, we limited the number of the dependencies to the minimum and where possible to only one dependency on the target class.

After the test preparation, the ten ACC-tools were tested. All the tools were subjected to the same test, described in the test script. During the first step of the test, the planned modular architecture was entered into the tool, including the mapping of modules to source code units, and the tool's output of the dependency analysis (if provided) was assessed. During the second step, the rules restricting the dependencies between modules were defined, and the output of the tool's conformance check was studied and compared with the expected result and with the output of the tool's dependency analysis. During the third step, the test results of the tools were aggregated and compared.

5.4.2 Included Dependency Types

Twenty-five direct dependency types were included in the test and nine indirect dependency types. For each direct and each indirect dependency type at least one separate test case was incorporated. For a part of the dependency types, additional test cases were created with variations of the type. This approach resulted in 34 direct and 30 indirect test cases.

Direct Dependency Types in the Test

The 25 direct structural dependency types in the test are shown in Table 5.2, together with a code example. Each code example shows a code construct that, if programmed within Class1 in Figure 5.1, would violate the intended architecture in Figure 5.1. This is because the code construct includes a dependency to an element of ModuleB2, while the intended architecture does not allow ModuleA1to use ModuleB2. In these cases, we expect ACC-tools to report a violation with at least a specification of the from-class and the to-class. Most of the examples refer to elements in Figure 5.1, but to keep the figure clear, some specific elements are not included, like an enumeration, exception, and interface.

Table 5.2: Direct dependency types in the benchmark test

Dependency type	**Example code**
Import	
Class import	import ModuleB.ModuleB2.Class3;
Declaration	
Instance variable	private Class3 class3;
Class variable	private static Class3 class3;
Local variable	public void method() { Class3 class3; }
Parameter	public void method(Class3 class3) {}
Return type	public Class3 method() {}
Exception	public void method() throws Class4{ throw new Class4 ("..."); }
Type cast	Object o = (Class3) new Object();
Call	
Instance method	variable = class3.method();
Instance method-inherited	variable = class3.methodSuper();
Class method	variable = class3.classMethod();
Constructor	new Class3();
Inner class method	variable = class3.InnerClass.method();
Interface method	interfaceI.interfaceMethod();
Library class method	libraryClassI.libraryMethod();
Access	
Instance variable	variable = class3.variable;
Instance variable-inherited	variable = class3.variableSuper;
Class variable	variable = Class3.classVariable;
Constant variable	variable = class3.constantVariable;
Enumeration	System.out.println(Enumeration.VALI);
Object reference	method(class3);
Inheritance	
Extends class	public class ClassI extends Class3 { }
Extends abstract class	Idem, but in this case Class3 should be abstract.
Implements interface	public class ClassI implements InterfaceI { }
Annotation	
Class annotation	@Class3

Indirect Dependency Types in the Test

We included nine indirect structural dependency types in our test, which are shown in Table 5.3, together with a code example per type. Each code example shows a code construct that, if programmed within Class1 in Figure 5.1, would violate the intended architecture in Figure 5.1. Because the code construct includes a dependency to an element of ModuleB2, while the intended architecture does not allow ModuleA1 to use ModuleB2. In these cases, we expect ACC-tools to report a violation with a specification of the from-class and the final to-class.

Table 5.3: Indirect dependency types in the benchmark test

Dependency type	**Example code**
Call	
Instance method	variable = class2.class3.method();
Instance method-inherited	variable = class2.methodSuper();
Class method	variable = class2.class3.classMethod();
Access	
Instance variable	variable = class2.class3.variable;
Instance variable-inherited	variable = class2.variableSuper();
Class variable	variable = class2.class3.classVariable;
Object reference-Reference var.	variable = class2.method(class2.class3.class4);
Object reference-Return value	Object o = (Object) class2.getClass4();
Inheritance	
Extends-implements variations	public class Class1 extends Class2 { } public class Class2 extends SuperClass { }

5.4.3 Findings: Accuracy of Dependency Detection

The test results of our benchmark tests are shown in detail in Table 5.4 and 5.5, while the most interesting findings are described below. Table 5.4 shows the results with regard to direct dependencies, and Table 5.5 shows the results with regard to indirect dependencies.

As a first observation, we noted that the false positive rate is null for all ten tested tools; thus, no false positive dependencies were reported. For the observations regarding the sensitivity of the tools, more text is needed. These results are described in detail in the following sub sections.

Direct Dependencies

Direct dependencies, caused by type declaration (except local variables), method call, variable access (except constants and object references), and inheritance, were detected by all tested tools, except ConQAT (which missed five type declaration dependency types) and SAVE (which missed two type declaration, one method call, and all six variable access dependency types). The following direct dependency types were often missing or were not reported accurately:

- Import dependencies were detected only by two tools: JITTAC, and SAVE; the two tools that analyze source files only. Import statements are not included in compiled files.
- A type declaration of an initialized local variable was detected only by the following six tools: Dependometer, JITTAC, Lattix, SAVE, Sonargraph, and Structure101. However, a type declaration of a not-initialized local variable was detected only by JITTAC and SAVE; the two tools that analyze the source files only. Not-initialized local variables are removed in compiled files. Interesting, since the tools that analyze compiled files were able to detect other declaration cases without initialization.
- A call of an instance method of an inner class was reported by all tools, except SAVE. However, the tools differ considerably in the accuracy of the reported to-class. JITTAC, Macker, Sonargraph and Sonar ARE were specific and reported the outer and inner class. ConQAT, Dependometer, dTangler, Lattix and Structure101 were less accurate and reported only the outer class.
- Access of a constant variable was detected only by three tools: Dependometer, JITTAC, and Sonargraph Architect. We included three test cases: one with a constant instance variable, one with a constant class variable, and one with an interface class variable. However, the results per tool were the same over these three test cases. Tools that analyze compiled code only, have problems with the recognition of constants, since their values are in-lined by the Java compiler.

Table 5.4: Benchmark test, Detection of direct dependencies (0 = not detected; 1 = detected)

Dependency type	**ConQAT**	**Dependometer**	**dTangler**	**JITTAC**	**Lattix**	**Macker**	**SAVE**	**Sonar ARE**	**Sonargraph**	**Structure101**	
Import											
Class import	0	0	0	1	0	0	1	0	0	0	2
Declaration											
Instance variable	0	1	1	1	1	1	1	1	1	1	9
Class variable	0	1	1	1	1	1	1	1	1	1	9
Local variable, initialized	0	1	0	1	1	0	1	0	1	1	6
Parameter	0	1	1	1	1	1	1	1	1	1	9
Return type	0	1	1	1	1	1	0	1	1	1	8
Exception	1	1	1	1	1	1	1	1	1	1	10
Type cast	1	1	1	1	1	1	0	1	1	1	9
Call											
Instance method	1	1	1	1	1	1	1	1	1	1	10
Instance method, inherited	1	1	1	1	1	1	1	1	1	1	10
Class method	1	1	1	1	1	1	1	1	1	1	10
Constructor	1	1	1	1	1	1	1	1	1	1	10
Inner class method (instance)	1	1	1	1	1	1	0	1	1	1	9
Interface method	1	1	1	1	1	1	1	1	1	1	10
Library class method	1	1	1	1	1	1	1	1	1	1	10
Access											
Instance variable (read, write)	1	1	1	1	1	1	0	1	1	1	9
Instance variable, inherited	1	1	1	1	1	1	0	1	1	1	9
Class variable	1	1	1	1	1	1	0	1	1	1	9
Constant variable	0	1	0	1	0	0	0	0	1	0	3
Enumeration	1	1	1	1	1	1	0	1	1	1	9
Object reference, param. value	0	1	0	0	0	0	0	0	1	1	3
Inheritance											
Extends class	1	1	1	1	1	1	1	1	1	1	10
Extends abstract class	1	1	1	1	1	1	1	1	1	1	10
Implements interface	1	1	1	1	1	1	1	1	1	1	10
Annotation											
Class annotation	0	1	0	1	1	0	0	0	1	1	5
Detected (out of 25)	16	24	20	24	22	20	15	20	24	23	
Sensitivity (in %) (average = 83)	64	96	80	96	88	80	60	80	96	92	

Dependometer and Sonargraph were detecting an access of a constant variable only with the option marked to include the source code in the analysis. Although SAVE analyzes source code, it did not report a dependency in one of the three test cases.

- Access of an object reference in the form of a parameter value (or argument), was detected only by three tools: Dependometer, Sonargraph, and Structure101. Another test case of this dependency type, with an object reference included in an if-clause, was detected by two tools only: Dependometer, and Sonargraph.
- Dependencies of type annotation, were detected only by five tools: Dependometer, JITTAC, Lattix, Sonargraph, and Structure101.

Indirect Dependencies

Indirect dependencies caused by method call and variable access (except an object reference as return value), were detected by all tested tools, except SAVE, which did not report access dependencies. Even double indirect dependencies were detected (for instance, from Class 1 in Figure 5.1, via Class 2 and Class 3 to Class 4). However, the following indirect dependency types were often missing or were not reported accurately:

- A call of an inherited instance method was reported accurately only by three tools: JITTAC, SAVE, and Structure101. These tools reported an indirect dependency to the super class where the method was actually implemented, although this method was called via a subclass. The other tools reported a dependency to the intermediate subclass, but not to the super class where the method was implemented. Consequently, these tools did not report a violation in the test cases where the subclass is part of an allowed-to-use module, while its super class is part of a not-allowed-to-use module.
- Access of an inherited instance variable was reported accurately only by two tools: JITTAC, and Structure101. These tools reported an indirect dependency to the super class where the variable was actually implemented, although this variable was accessed via a subclass. The other tools reported a dependency to the subclass, but not to the super class where the variable was implemented (except SAVE, which did report no dependency at all). Consequently, these tools did not report a violation in the test cases where the subclass was part of an allowed-to-use module, while its super class was part of a not-allowed-to-use module.

Table 5.5: Benchmark test - Detection of indirect dependencies
(0 = not detected; 1 = detected)

Dependency type	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure101	
Call											
Instance method	1	1	1	1	1	1	1	1	1	1	10
Instance method, inherited	0	0	0	1	0	0	1	0	0	1	3
Class method	1	1	1	1	1	1	1	1	1	1	10
Access											
Instance variable	1	1	1	1	1	1	0	1	1	1	9
Instance variable, inherited	0	0	0	1	0	0	0	0	0	1	2
Class variable	1	1	1	1	1	1	0	1	1	1	9
Object reference – Reference var.	1	1	1	1	1	1	0	1	1	1	9
Object reference – Return value	0	1	0	0	0	0	0	0	0	1	2
Inheritance											
Extends – implements variations	0	0	0	0	0	0	0	0	0	0	0
Detected (out of 9)	5	6	5	7	5	5	3	5	5	8	
Sensitivity (in %) (average = 60)	56	67	56	78	56	56	33	56	56	89	

- Access of an object reference, received as return value of a method call, was reported by only two tools: Dependometer, and Structure101.
- An inherited dependency on a super-super class or interface of the from-class, solely based on extends/implements constructs, was not reported by any tool. We included three variations in our test (extends-extends, extends-implements, implements-extends), but none was reported.

5.5 FreeMind Test

In addition to the benchmark test we have conducted tests with an open source system. These tests were aimed at quantitative and qualitative tool comparison. Two types of test were conducted, one aimed at the accuracy of dependency detection, and the other on the accuracy of violation and dependency reporting.

5.5.1 Method

FreeMind

We used the mind-mapping tool FreeMind. Three main packages in FreeMind, as shown in Figure 5.2, were included in our tests: accessories, plugins and freemind.

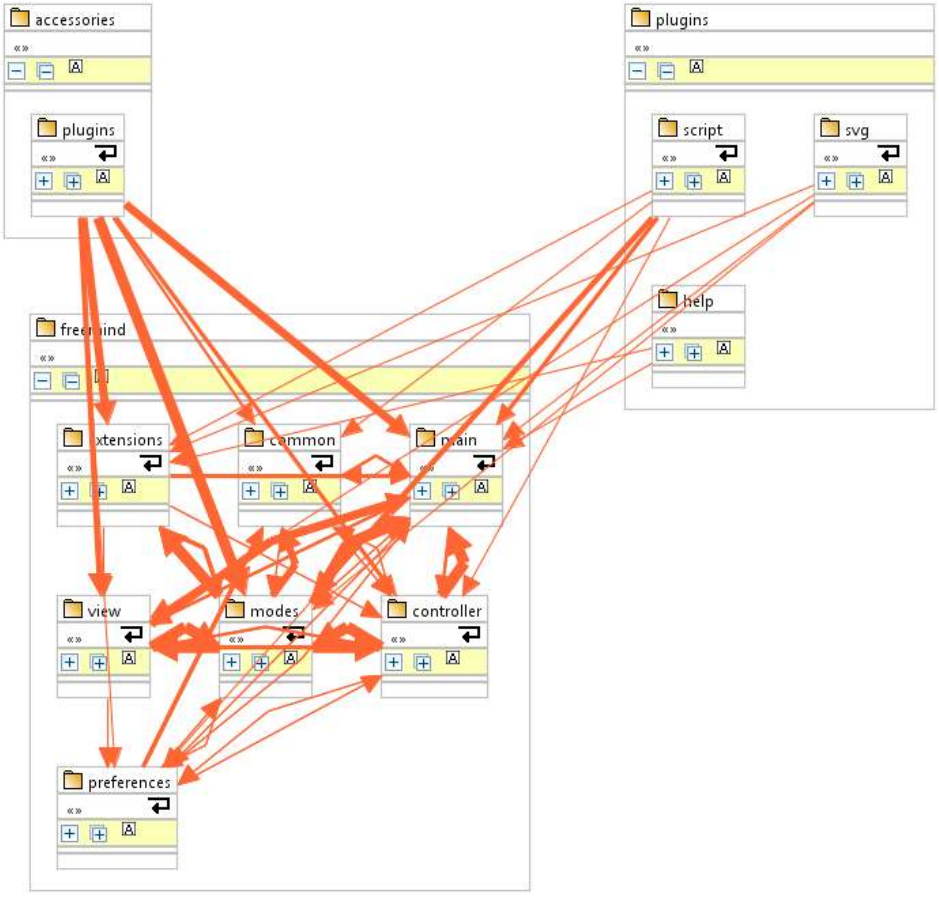

Figure 5.2: The package structure of Freemind, with dependency relations, as depicted by SAVE. Thick lines represent more dependency relations than thin lines.

The following packages were excluded from the test, since these packages were available in the source code, but not in the compiled code: plugins.latex.*, plugins.collaboration.*, and tests.*. We used version 0.9.0 for our tests; retrieved on 23-08-2012 from http://freemind.sourceforge.net/wiki/index.php/Download.

We selected FreeMind, because it suited to the following criteria. First, the system needed to be written in Java, just as the benchmark test. Furthermore, source code files and compiled code files needed to be present, since some tools use source code, others compiled code, while some use both. Second, the system needed to have an uncomplicated implemented architecture, to enable a straightforward, error-free registration of the intended architecture in the tested tools. Third, the system needed to contain reasonable numbers of dependencies between its modules. Ideally, these dependencies should cover a wide range of possible dependency types. Fourth, the number of classes had to be lower than 1000, due to size constraints of some ACC-tool licenses.

Method: Accuracy of Dependency Detection

The objective of this test was to determine how well the ACC tools were able to report dependencies of different types, just as in the benchmark test, but now in a real system. For this purpose, we selected the large class ScriptingEngine within sub package plugins.script, since it contained a considerable number of dependencies of different types. Furthermore, the class touched a diversity of object oriented specialties, including super class, inner class, and anonymous class.

Identification of dependency causing constructs

We identified all code constructs within class ScriptingEngine, which caused dependencies to package "freemind", by manual inspection of the code, aided by the supporting facilities of the Java editor in the Eclipse IDE. To ensure the accuracy of our work during this step, one author made an inventory of the dependency-causing constructs, the depended-upon classes, and the related dependency types, while another author checked the inventory afterwards. Based on the inventory, 109 dependency-causing code constructs were included in a score form.

Tool selection and testing

We tested all ten tools to determine which depended-upon classes were reported for class ScriptingEngine, by following the steps below.

1. Registration of rule: plugins.script.ScriptingEngine is not allowed to use package freemind.
2. Activation of the conformance check.
3. Study of the reported violations and dependencies.

Next, we selected the tools that were providing sufficient information to be able to trace reported dependencies to code constructs. The tools differ considerably in the exactness of dependency messages, as will be discussed in the result sub section. Only the following five tools provided detailed enough information to be included in this test: JITTAC, Lattix, SAVE, Sonargraph Architect, and Structure101. With these five tools we also went through the following steps:

4. Tracing of the reported dependencies to the manually identified dependency constructs.
5. Scoring of the detected dependency constructs in a scoring form per tool.

Scoring

We scored mildly, meaning that we marked a dependency as detected, if one of the reported dependencies messages could be related to the dependency-causing code construct. With a strict accuracy level in mind, the number of missed dependencies would have been much higher.

- In case of inner class related dependencies, we scored a dependency also to be detected if it was reported as a dependency to the outer class instead of to the inner class.
- In case of inheritance related dependencies, we scored a dependency also to be detected if it was reported as a dependency to a sub class instead of the super class that actually implemented a depended-upon variable or method.
- In case of dependency messages with a non-optimal accuracy, we scored all dependencies to be detected that could be related to the dependency message. For instance, if a tool reported one dependency to class X of type declaration or access at line Y, while in the source code a declaration construct and a type cast construct were present, both were scored to be detected. Similarly, if a tool reported one dependency to class X of type access in method Z, while in the source code of the method five of these access construct were present, all five were scored to be detected.

Method: Accuracy of Reported Violation and Dependency Messages

The objective of this test was to identify differences in quantity and exactness of the reported violation and dependency messages. For each tool, we performed the following steps:

- Registration of two rules: 1) package accessories is not allowed to use package freemind; and 2) package plugins is not allowed to use package freemind.
- Activation of the conformance check.
- Study of the reported violation and dependency messages.
- Scoring of the number and exactness of the messages.

5.5.2 Findings: Accuracy of Dependency Detection

The test results of the FreeMind tests concerning the accuracy of dependency detection are presented below. All ten tools provided at least information in their violation messages on the depended-upon to-classes per from class. Therefore, the results of the reported depended-upon classes per tool are presented at first. Next, the results of the test at the level of the 109 identified dependency constructs are presented. Five tools are included in these results, since only these tools provided detailed enough information in their violation messages or dependency messages. Finally, examples are provided of code constructs that caused hard to detect dependencies.

Detected Depended-Upon Classes

Class ScriptingEngine depends-upon seventeen classes, of which most are shown in the freemind package in Figure 5.3. Two of these classes contain inner classes, which are also used by ScriptingEngine, namely OptionalDontShowMeAgainDialogue and Tools. In our test, we expected that

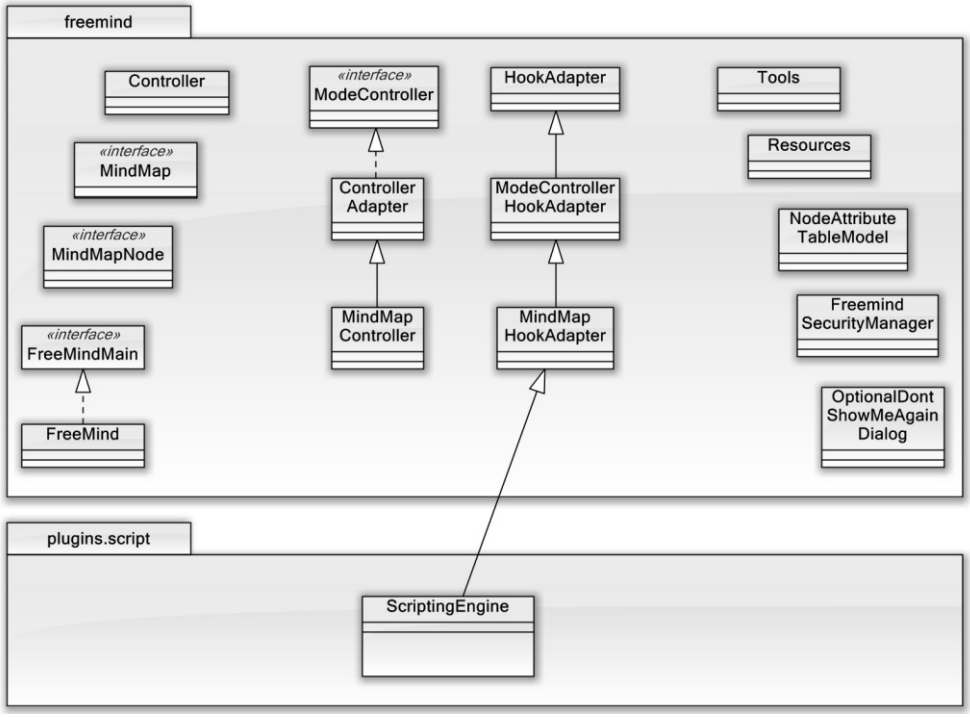

Figure 5.3: Class ScriptingEngine and its depended-upon classes in package freemind.

usage of the seventeen depended-upon classes would be reported as violations. Please note that Figure 5.3 provides a simplified view. There are many more classes in package freemind, and the shown classes are in reality included in different subpackages of freemind. Furthermore, for reasons of readability, we have drawn no dependency arrows in the diagram, only UML inheritance relations.

Several inheritance structures are shown in the figure. For example, ScriptingEngine inherits from three superclasses in package freemind. In our test, we expected that usage of these classes would be reported as violations; especially in case of a call of method or in case of access of an attribute inherited from one of these classes. In these cases, actual usage takes place of the super class that implements the method or variable.

Table 5.6 shows for each of the tested tool, which depended-upon classes in package freemind were reported in violation reports or other views. Furthermore, it shows per class the number of related dependency-causing constructs. The bottom row in the table shows that JITTAC was the only tool that reported usage of all seventeen classes. Dependometer, SAVE, and Sonargraph reported usage of fifteen classes, a sensitivity of 88 percent. Macker, Sonar ARE, and Structure101 reported usage of fourteen classes, a sensitivity of 82 percent. Finally, Conquat, dTangler, and Lattix reported usage of twelve classes, a sensitivity of 71 percent. On average, 82 percent of the classes was detected.

All not reported classes (by all tools) were of one of the following types:

- Super class (ControllerAdapter, Hookadapter), of which methods are used via inheritance;
- Inner class (OptionalDontShowMeAgainDialogue. StandardPropertyHandler, Tools.BooleanHolder), which may be used in various ways: Import, Declaration, Access, Call;
- Normal class (FreeMind), of which only static constant variables are accessed.

Table 5.6: Freemind test, Detected (1) and not detected (0) depended-upon classes

Reported Classes	**Nr of dep. constructs**	**ConQAT**	**Dependometer**	**dTangler**	**JITTAC**	**Lattix**	**Macker**	**SAVE**	**Sonar ARE**	**Sonargraph**	**Structure 101**	
Controller	1	1	1	1	1	1	1	1	1	1	1	10
ControllerAdapter	5	0	0	0	1	0	0	1	0	0	0	2
FreeMind	12	0	1	0	1	0	0	1	0	1	0	4
FreeMindMain	16	1	1	1	1	1	1	1	1	1	1	10
FreeMindSecurityManager	5	1	1	1	1	1	1	1	1	1	1	10
HookAdapter	6	0	0	0	1	0	0	1	0	0	0	2
MindMap	1	1	1	1	1	1	1	1	1	1	1	10
MindMapController	6	1	1	1	1	1	1	1	1	1	1	10
MindMapHookAdapter	5	1	1	1	1	1	1	1	1	1	1	10
MindMapNode	17	1	1	1	1	1	1	1	1	1	1	10
ModeController	2	1	1	1	1	1	1	1	1	1	1	10
NodeAttributeTableModel	6	1	1	1	1	1	1	1	1	1	1	10
OptionalDontShowMeAgainDialog	5	1	1	1	1	1	1	1	1	1	1	10
- StandardPropertyHandler (inner)	1	0	1	0	1	0	1	0	1	1	1	6
Resources	2	1	1	1	1	1	1	1	1	1	1	10
Tools	6	1	1	1	1	1	1	1	1	1	1	10
- BooleanHolder (inner)	13	0	1	0	1	0	1	0	1	1	1	6
Number of reported classes		12	15	12	17	12	14	15	14	15	14	
Sensitivity (in%) (average = 82)		71	88	71	100	71	82	88	82	88	82	

Detected Dependencies
Table 5.7 shows for each of the five in the detailed test included tools, how many dependencies per dependency type were reported to classes in package freemind. All five tools detected all the dependencies of the following dependency types: 1) method call, class method; 2) method call, interface method; 3) inheritance, extends class.

Dependencies of the other dependency types, which were not reported by one or more tools, are per type discussed below.

- Import, class import: Lattix, Structure101, and Sonargraph missed all 10 dependencies. SAVE missed one, because of a not-recognized inner class.
- Declaration, local variable: SAVE missed all six dependencies (in contrast to the benchmark test), probably because off complex initialization statements at the same line.
- Declaration, parameter: SAVE missed three out of seven dependencies (because of a not detected inner class), while Sonargraph missed one.
- Declaration, type cast: SAVE missed all two dependencies (as in the benchmark test).
- Call, instance method: JITTAC missed two dependencies, probably because these were located within an anonymous class.
- Call, instance, inherited: Lattix missed eight out of fourteen dependencies, Sonargraph also missed eight, and Structure101 missed all fourteen (in contrast to the benchmark test), all in inheritance trees up to four levels.
- Call, constructor: SAVE missed two dependencies out of three: two constructor invocations of inner classes. It detected an invocation of the constructor of a normal class (as in the benchmark test).
- Call, inner class method: SAVE missed all two instance method invocations (as in the benchmark test).
- Access, constant: Lattix, Structure101 and SAVE missed all twelve dependencies (as in the benchmark test).
- Access, object reference: JITTAC and Lattix missed all sixteen dependencies (as in the benchmark test); 15 caused by variables passed as parameter value (or argument) and one caused by a variable used within an if statement. SAVE missed six, because of not detected inner classes.

Remind, we scored mildly, as explained in the method sub section of this test.

Table 5.7: Freemind test – Reported dependencies per dependency type

Dependency type (number of constructs)		**JITTAC**	**Lattix**	**SAVE**	**Sonargraph**	**Structure 101**
Import						
Class import	(10)	10	0	9	0	0
Declaration						
Local variable	(6)	6	6	0	6	6
Parameter	(7)	7	7	4	6	7
Type cast	(2)	2	2	0	2	2
Call						
Instance method	(11)	9	11	11	11	11
Instance method-inherited	(14)	14	6	14	6	0
Class method	(6)	6	6	6	6	6
Constructor	(3)	3	3	1	3	3
Inner class method (instance)	(2)	2	2	0	2	2
Interface method	(19)	19	19	19	19	19
Access						
Constant variable	(12)	12	0	0	12	0
Object reference	(16)	0	0	10	16	16
Inheritance						
Extends class	(1)	1	1	1	1	1
Detected	(109)	91	63	75	90	73
Sensitivity (in %)	(average = 72)	83	58	69	83	67

Examples of Code Constructs

Several examples of code constructs that caused dependencies that were hard-to-detect in the FreeMind test, are provided in Table 5.8. The first column shows the type of the dependency with, if needed, some added details. The second column shows the example code. The text that causes a dependency is shown in italic.

Table 5.8: Examples of code constructs within ScriptingEngine

Dependency type	**Example code**
Import	
Class import (inner class)	import *freemind.main.Tools.BooleanHolder*;
Declaration	
Local variable	MindMapNode *node* = getMindMapController().getMap().getRootNode();
Call	

Instance method
(within an anonymous class)
	final GroovyShell shell = new GroovyShell(binding) {
	public Object evaluate(..., ...) throws ... {
	try {
	securityManager.setFinalSecurityManager(...);
	}
	}
	};

Inherited method (of MindMapHookAdapter)	MindMapNode node = *getMindMapController()*.getMap().getRootNode();
Inherited method (of ControllerAdapter)	MindMapNode node = getMindMapController().*getMap()*.getRootNode();
Constructor (of inner class)	BooleanHolder bh = *new BooleanHolder*(false);
Inner class method	*bh*.setValue(true);
Access	
Constant class variable (passed as parameter value)	String executeWithoutAsking = frame.getProperty(FreeMind.*RESOURCES_SIGNED_SCRIPT_ARE_TRUSTED)*
Object reference (passed as parameter value)	performScriptOperation(element, *bh*);

5.5.3 Results: Accuracy of Reported Violation and Dependency Messages

Two functional types of messages could be distinguished: violation messages and dependency messages. *Violation messages* report on inconsistencies between the implemented architecture and the defined architecture, with a class at the lowest level of granularity. The second type, *dependency messages* provide information about the dependencies, like the dependency type and the location of the dependency-causing code constructs in the program code. Findings regarding these two types are presented in the following subsections.

To illustrate the difference between the two types of messages, we have included practical examples from the FreeMind test with Structure101. Three examples show messages at three different levels of abstraction. Figure 5.4a shows a graphic with a violation message: a violating relation (dotted) from class ScriptingEngine to package freemind. After selection of the relation, the tool shows the message "ScriptingEngine uses freemind [55]". Figure 5.4b shows violation messages within the view that specifies the violating relation; it lists fourteen depended-upon classes and interfaces within package freemind. Figure 5.4c shows examples of dependency messages. Structure101 reported 55 instances of dependencies from ScriptingEngine to freemind. These dependencies were specified in a separate view, which listed for each dependency: the from-class (ScriptingEngine), from-method (e.g., evaluate), dependency type (e.g., extends), to-class (e.g., freemind.main.FreeMindSecurityManager), and to-method (e.g., setFinalSecurityManager).

Violation Messages

Violation messages indicate where the implemented architecture deviates from the intended architecture. The tested tools differ considerably in the way violations are reported, for instance by means of colors in a Dependency Structure Matrix (DSM), additional symbols or line styles in diagrams, textual reports, or indicated code lines in a code viewer. Most tools offer more than one way to report violations, especially the commercial tools.

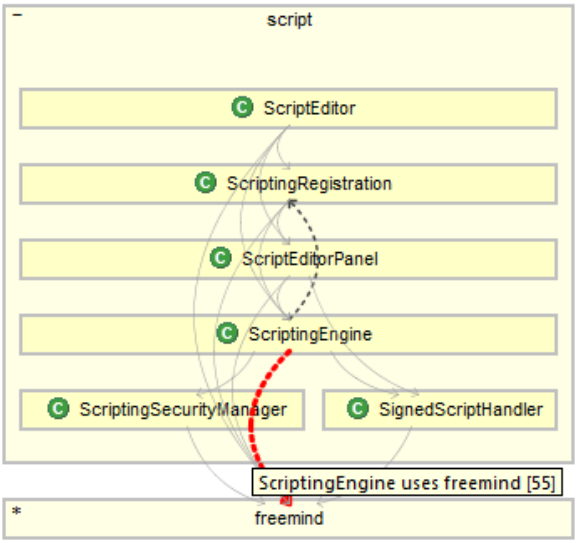

Figure 5.4a: The package structure of Freemind plugins.script, as depicted by Structure101, with one violating dependency relation (red) from class ScriptingEngine to package freemind.

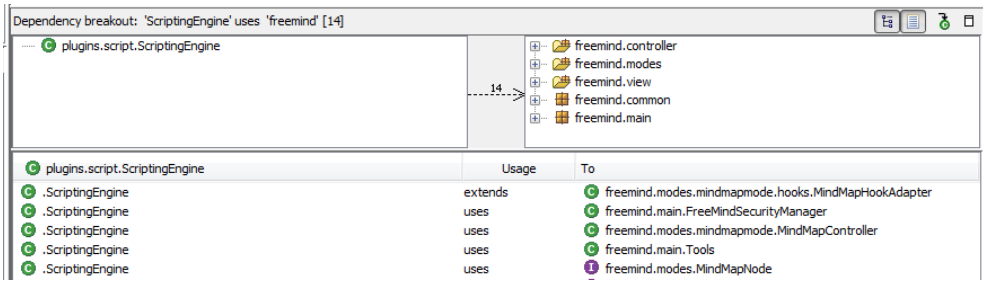

Figure 5.4b: Specification of the violating relation; 14 classes and interfaces are used.

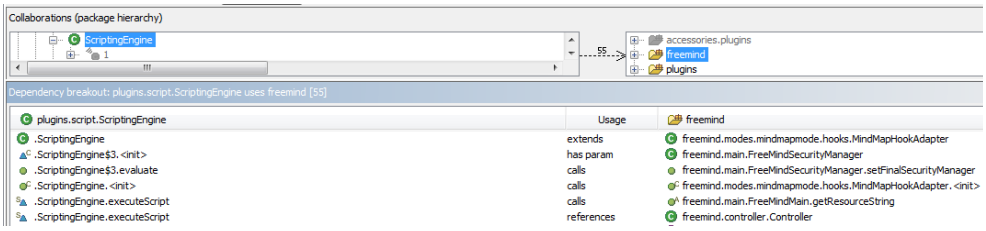

Figure 5.4c: Detailed specification of dependencies from ScriptingEngine to freemind.

Observations regarding the accuracy of violation messages are described below.

- Reported violations versus reported dependencies
  No cases were noticed, in which a tool reported a dependency to a class, but failed to report a violation for this dependency. Since this is also true for the benchmark test, Table 5.4 and 5.5 do not only show the true positive violations, but also the false negative violations per tool and per dependency type. However, one exception applies: SAVE reported correct violations for classes containing violating *direct* dependencies, even when the specific dependency of the test case was not detected. The tool was able to do this, based on detected import statements. SAVE did not have this advantage in case of *indirect* dependencies, since no import statement was included in these cases, and in case of an inner class.
- Exactness of the violation messages
  To show where a violation is present in the modular architecture, seven of the ten tools (see Table 5.9) include violation messages in graphical overviews. Table 5.9 shows also that all tools were able to report the from-class and to-class, generally in text-based violation messages. However, management information to indicate the severity of the violation of a rule,

Table 5.9: Exactness of violation and dependency messages
(√= Included in Message).

	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure101
Violation message										
Graphical overview	√		√	√	√		√		√	√
Class from - Class to	√	√	√	√	√	√	√	√	√	√
Dependency message										
Class from - Class to	√	√	√	√	√	√	√	√	√	√
Dependency type					√		√		√	√
Method from – Method to				√			√			√
Line				√[1]	√			√[1]	√	
Position within the line				√[1]	√[1]					

[1]An indication of the line (and position of the dependency construct within the line) is not provided in a report, but in a code viewer or IDE plug-in.

like the actual number and/or strength of the underlying dependencies, is less frequently included in violation reports, while this is meaningful information. As a positive example, Structure101 shows, in Figure 5.4a, the number of corresponding dependency messages when a relation is selected. Furthermore, it is available in JITTAC diagrams, where the number of dependencies is shown per dependency arrow, and in SAVE diagrams, though less accurate, where the thickness of a line indicates the number of dependencies.

- Number of reported messages
  Because of different capabilities of the tools and different choices made by the developers, the tools report varying numbers of violation messages at the level of from-class, to-class. This is illustrated in Table 5.10, which holds the numbers of violation messages per tool during the FreeMind test (except for JITTAC and SAVE; see table footnote). The reported violations against the rule that the freemind package should not be used, are shown for package accessories, for package plugins, and for class plugins.script.ScriptingEngine. Several tools report more violations than depended-upon classes in their violation report. In these cases, a separate message is created for each combination of from-class, to-class, and dependency type.

Dependency Messages

A dependency message enables developers to resolve a violation efficiently. To do so, detailed information is needed to trace the dependencies in the code. Six tools provide this information in separate reports or views: JITTAC, Lattix, SAVE, Sonar ARE, Sonargraph Architect, Structure101. Our observations regarding the exactness of the dependency messages are described below.

Exactness of the Reported Location of a Dependency

The tools differ in exactness of the reported location of a dependency, as shown in Table 5.9. At the highest level of accuracy, a tool indicates a dependency-causing construct within a line of code, even when several dependency-causing constructs are included in the same line. Only two tools were able to do this: JITTAC, and Lattix. Both tools highlight the violating code constructs in the source code within an IDE's code editor. Table 5.9 shows that these two tools provide the following information in dependency messages: class-from, line, and position within line. However, Lattix did not always appoint the line and code construct correctly.

Two other tools with code viewers, Sonar ARE and Sonargraph Architect, indicated the line correctly, but not the position within the line. Sonar ARE's

usability was restricted by the fact that per from-class, it indicated only the first instance of a violating usage of a depended-upon class. Following usages of the to-class were not indicated.

Several tools were providing reports as well. Sonargraph Architect was providing the most detailed report with from-class, to-class, dependency type, and a correct line number. Lattix was providing an information view with dependency types and line numbers of the dependencies in the code, but here also, it did not always specify the correct line number in the source code. SAVE and Structure101 were providing reports, which indicated the method including the violating dependency in the from-class and, in case of method calls, also the method of the to-class.

The practical implications of the different approaches became clear during the FreeMind test, in which reported violations needed to be traced to 109 constructs in the program code. At first, we tried to do this based on the dependency messages in the reports or in the dependency browsers of the tools. Sonargraph Architect provided a very useful report, which made it easy to trace the dependencies in the code. It contained all the detected dependencies with type and line number. The

Table 5.10: Reported numbers of violation and dependency messages within the Freemind test.

	ConQAT	Dependometer	dTangler	JITTAC[1]	Lattix	Macker	SAVE[1]	Sonar ARE	Sonargraph	Structure101
accessories → freemind										
Violations	228	435	282	1739	288	386	1332	378	362	308
plugins → freemind										
Violations	54	100	63	348	65	87	229	79	75	71
ScriptingEngine → freemind										
Violations	12	18	12	97	25	16	54	15	15	14
Dependencies				97	25		54		121	55

[1] JITTAC and SAVE provide no aggregated violation report with messages at the level of from-class, to-class. Instead, the table shows the number of reported dependency messages.

reports of SAVE and Structure101 required much more analysis and interpretation, with risk of misinterpretation in complex situations. In part, because one dependency message may abstract several dependency-causing code constructs. In concrete terms: SAVE and Structure101 reported respectively 54 and 55 messages, but these covered respectively 75 and 73 dependencies in the code. Since Lattix's reports proved to be too inaccurate for our use, and because JITTAC did not provide a report or dependency browser, we used the messages provided in code viewers of these tools. In case of Lattix, we combined different reports with the code viewer to circumvent incorrect line numbers and positions. Finally, since Sonar ARE's support for this test was too restrictive, we did not include the tool in this part of the FreeMind test.

Exactness of the Reported Dependency Type

Only four tools provide a dependency type (as shown in Table 5.9), which differentiates between different types of usage, like declare, access, or call. The tools differ in the exactness of the reported dependency type: the numbers of dependency types vary per tool, and the names of these types vary as well. Consequently, different tools label a dependency type in our classification in several ways. For example, a dependency of type "Call constructor" in our classification was reported by Lattix as "Construct with Arguments", by SAVE as "ACCESS", by Sonargraph as "Uses new", and by Structure101 as "calls". Some types used by the tools are very specific, while others cover many forms of code constructs. Even if two tools use the same type-name, like access, they may cover different dependency types within our classification.

5.6 Frequency of Hard-To-Detect Dependency Types

The results of the benchmark test and FreeMind test have shown that certain types of depended-upon classes and ten types of dependencies are not reported at all by some tools, or are reported inaccurately. To address the relevance of these findings, we have measured the number of dependencies per dependency type in five open source systems. The method and the results of this experiment are described below.

5.6.1 Method

To measure the numbers of dependencies per dependency type distinguished in the benchmark test, we needed a tool that was able to detect and report all the types of dependencies in these tests. Since no tested tool was able to detect dependencies of all these types, we improved and extended a tool, HUSACCT, which we had developed in a line of research that focused on ACC support for rich sets of module and rule types (Pruijt et al. 2014). We improved the dependency analysis process in HUSACCT version 4.0 up to the point, and beyond, where all dependencies in the benchmark test and FreeMind test were detected and reported, without false positives. Since a considerable part of the not-reported dependencies in the benchmark test and FreeMind test were related to inheritance and inner class constructs, we extended the analysis process and data model to detect and store these characteristics per dependency. Furthermore, we extended the dependency report with a statistics sheet, which presents the numbers of dependencies in different ways: total, direct, indirect, total per type, total of inheritance related dependencies, total of inner class related dependencies, et cetera. To enable the reproduction of the experiment by other researchers, the improvements and extensions were included in version 4.1, which is downloadable via http://husacct.github.io/HUSACCT/.

Next, we selected five open source subject systems and downloaded their source code. We used the following selection guidelines. First, the systems had to be written in Java, since our benchmark test and FreeMind test were also Java-based. Second, FreeMind was included, since it is interesting to compare its analysis results in the FreeMind test with those of other subject systems. Third, four other systems were selected, because they were used in published scientific experiments of other authors, but also because of their notoriety.

The five systems, their version, download address and size are shown in Table 5.11. The source of all the systems was downloaded on February 10, 2015 (except FreeMind, which was downloaded already in 2012 from another web address; the current location of the source is included in the table). An impression of the size per system is provided in kilo lines of code (KLOC). The given numbers

show the lines of code (including comments and blank lines) in all the files with extension ".java", as measured by HUSACCT.

Finally, for each system the source code was analyzed with HUSACCT and a dependency report was generated. The numbers of dependencies per dependency type per system were included in a spreadsheet and averages were calculated, per system, and over the systems. These final results are presented in the next sub section. The reported numbers of dependencies: a) include internal dependencies and dependencies on external systems (library objects); b) exclude dependencies from a class to itself; c) exclude most dependencies on primitive types.

5.6.2 Results

Table 5.12 shows the numbers of dependencies per dependency type and per system. The ten dependency types that proved *hard-to-detect* in our tests are included in the table; they are shown in italics. The results are presented in three groups, visible in the first column, namely: 1) All dependencies; 2) Inheritance related dependencies; and 3) Inner class related dependencies. Per group and per dependency type, the numbers of reported dependencies are shown per subject system, while the last column shows the average percentage of the dependency type over the four subject systems. The average percentage of a dependency type is calculated as the average for this type of the four subject system specific percentages (not shown in the table).

The first group shows the numbers of all reported dependencies. The first row within this group shows that the total number of dependencies increases with the size of the subject system, as can be expected. Thereafter, two subdivisions are shown; one for direct versus indirect dependencies, and another for the six main types (Import, Declaration, Call, Access, Inheritance, Annotation). The numbers show that on average 84 percent of the dependencies is direct, while 16 percent is indirect in these subject systems. Furthermore, that Import statements cause ten percent of the dependencies, while Declaration, Call, Access, Inheritance, and Annotation statements caused respectively 20, 39, 26, 3, and 2 percent.

Table 5.11: Subject systems used in the experiment

System	Download address	Size (KLOC)
Ant 1.9.4	http://archive.apache.org/dist/ant/source/	267
Findbugs 3.0.0	https://code.google.com/p/findbugs/source/browse/?name=3.0.0	327
Freemind 0.9.0	http://sourceforge.net/projects/freemind/files/freemind/0.9.0/	87
Hibernate 4.2.4	https://github.com/hibernate/hibernate-orm/releases/tag/4.2.4.Final	713
Struts 2.3.20	http://struts.apache.org/download.cgi#struts2320	277

The second group shows the numbers of inheritance related dependencies, which are caused by: 1) Access of an inherited variable; 2) Call of an inherited method; and 3) Inheritance by means of an extends or implements statement. The numbers show that on average twelve percent of the dependencies is inheritance related, of which three percent of type access, six percent of type call, and three percent of type inheritance.

The third group shows the numbers of inner class related dependencies. A dependency was marked as such, if the from-class or to-class is an inner class (or if both classes are). The numbers show that on average nine percent of the dependencies is inner class related.

Frequency of Hard-To-Detect Types of Dependency

Ten dependency types, shown in italics in Table 5.12, were hard-to-detect by several tools in our tests. The hard-to-detect types in the first group of Table 5.12 represent 39 percent of all dependencies in the four systems: Import (10 percent),

Table 5.12: Number of dependencies per dependency type

Dependencies	Type	Ant	Findbugs	Hibern.	Struts	%
All	All	88,943	128,876	401,356	122,877	100
	- Direct	76,350	115,070	319,530	102,332	84
	- Indirect	12,593	13,806	81,826	20,545	16
	Import	8,422	15,988	39,670	12,528	10
	Declaration	18,282	28,621	69,372	22,764	20
	- Local var.	5,076	10,167	22,976	8,316	7
	Call	37,208	43,271	155,051	51,874	39
	Access	20,388	35,276	115,224	29,804	26
	- Constant variable	1,923	1,695	4,589	920	1
	- Object ref. Direct	12,233	21,541	54,797	16,223	14
	- Object ref. Indirect	3,047	5,675	28,218	5,244	5
	Inheritance	2,368	3,066	8,610	4,665	3
	Annotation	2,275	2,654	13,429	1,242	2
Inheritance related	All	10,291	8,731	47,608	19,219	12
	Access	1,961	1,956	19,200	5,043	3
	- Inh. var. Indirect	1,008	1,252	10,941	3,294	2
	Call	5,962	3,709	19,798	9,511	6
	- Inh. meth. Indirect	4,657	2,861	17,186	5,431	4
	Inheritance	2,368	3,066	8,610	4,665	3
	- Indirect	1,205	1,641	3,839	2,779	2
Inner class rel.	*All*	10,739	14,283	16,650	12,343	9

Declaration, local variable (7 percent), Access, constant variable (1 percent), Access, object reference, direct (14 percent), Access, object reference, indirect (5 percent), and Annotation (2 percent).

The hard-to-detect types in the second group, inheritance related dependencies, total to eight percent on average of the dependencies in the four systems: Access of an inherited variable, indirect, (2 percent), Call of an inherited method, indirect (4 percent), and Inheritance, indirect (2 percent). The total of eight percent may be added to the total of hard-to-detect dependencies in the first group, which makes 47% of potentially hard-to-detect dependencies, since there is no overlap between the types of hard-to-detect dependencies in the first and second group.

The third group concerns inner class related dependencies, of which all instances in our test were hard to detect by several tools. Nine percent of all dependencies fall within this group. However, this number may not be added to the sum of the hard-to-detect dependencies of the other groups, since there may be an overlap.

5.6.3 Comparison Results of FreeMind Test

Finally, we compared the average analysis result of the four subject systems in Table 5.12 with the results of the FreeMind system as a whole, and with the class plugins.script.ScriptingEngine, on which the FreeMind test focused. Table 5.13 shows that the distribution of the 44,146 dependencies in the FreeMind system over the dependency types differs only a little from the average distribution in the reference systems, the other four subject systems. Main difference is that the FreeMind system contains twelve percent inner class related dependencies, while the reference systems on average contain nine percent inner class related dependencies.

The dependencies shown for class plugins.script.ScriptingEngine are the dependencies included in the FreeMind test, so limited to dependencies to classes and interfaces in the package freemind. The numbers in Table 5.13 show that ScriptingEngine contains relatively more indirect dependencies to package freemind, more call dependencies, and more inheritance related dependencies; especially more calls of inherited methods. On the other hand, the class contains relatively less declaration dependencies, and no annotation dependencies nor indirect dependencies of type access of an inherited variable. In total, ScriptingEngine contains relatively more hard-to-detect dependencies: 41 percent in group one, 15 percent in group two, and 11 percent in group 3, compared to respectively 39, 8, and 9 percent.

HUSACCT reported dependencies for all 109 dependency-causing constructs in class ScriptingEngine to package freemind. However, HUSACCT reported 126

Table 5.13: Dependency types of ScriptingEngine compared to Freemind and other systems

Dependencies	Type	Scripting Engine	Scripting Engine %	Freemind %	Reference Systems %
All	All	126	100	100	100
	- Direct	92	73	86	84
	- Indirect	34	27	15	16
	Import	10	8	11	10
	Declaration	15	12	21	20
	- Local var.	6	5	7	7
	Call	61	48	40	39
	Access	35	28	25	26
	- Constant v.	12	9	1	1
	- Object ref. Direct	16	13	15	14
	- Object ref. Indirect	7	6	4	5
	Inheritance	5	4	3	3
	Annotation	0	0	0	2
Inheritance related	All	27	21	10	12
	Access	2	2	2	3
	- Inh. var. Indirect	0	0	1	2
	Call	20	16	5	6
	- Inh. meth. Indirect	15	12	4	4
	Inheritance	5	4	3	3
	- Indirect	4	3	1	2
Inner class rel.	*All*	14	11	12	9

dependencies, 17 more, since one construct may cause more than one dependency. The extra dependencies are of the following types: seven instances of Access, object reference, return value (indirect), six instances of Call, instance method, inherited (four direct, two indirect), and four instances of Inheritance, indirect. For example, construct "extends MindMapHookAdapter" causes not only a direct inheritance dependency to class MindMapHookAdapter, but also four extra indirect inheritance dependencies to classes and interfaces higher up in the inheritance hierarchy.

5.7 Discussion

In this section, we discuss the key findings, answer the research questions, and discuss identified challenges and their implications.

5.7.1 Key Findings

In our opinion, all tested tools provide useful functionality to perform an architecture compliance check. However, our tests show that all ten tools could improve the accuracy regarding dependency and violation reporting, though in varying degrees. A summarizing overview of the findings of our tests is provided in Table 5.14, which shows a relative comparison of the tools with respect to the tested characteristics. The subsections below elaborate on these findings and answers the research questions.

Although this study includes a tool test regarding ACC support, we do not advise on a "best" tool. To remain objective, we refrained from this. Accuracy is only one of several qualities that should be considered in the course of a selection process of an ACC-tool. Some tools offer only a limited set of functionality, while others provide a rich set as shown in Table 5.1, especially the commercial tools.

Accuracy of Dependency Detection and Violation Reporting

Our study shows that all ten tools were able to detect dependencies established by basic constructs, like method calls and type declaration. However, the test results show also that significant numbers of dependencies were not reported, even by the best scoring tool. Consequently, the answer to research question RQ1 (Do ACC tools find all the dependencies between modules in the software?) is negative. Numerous false negatives were identified, so all tools may improve on the sensitivity regarding dependency detection. The answer to RQ2 is also negative, since we found no differences in the sets of reported dependencies and reported violations. If a tool was able to detect a dependency, then it was also able to report the dependency if it violated an architectural rule. With regard to false positives, the tested tools performed well; no tool reported false positives. Consequently, the answer to research question RQ3 is negative.

The benchmark test showed that no tool in the test was able to detect all included dependency types, although several tools performed well. On the average, the ten tools detected 77 percent of the dependency types in the test-software: 83 percent of the 25 direct types and 60 percent of the 9 indirect types. The ten tools differ considerably in their ability to detect all types of dependencies included in our test. JITTAC and Structure101 detected the most direct and indirect dependency types; both 31 out of 34 types (91 percent). On the other side, ConQAT and SAVE detected a total of respectively 21 and 18 dependency types

(62 and 53 percent). Table 5.14 summarizes the results, based on the following scales: Direct dependencies * = 0-79%, ** = 80-89%, *** = 90-100%; Indirect dependencies: * = 0-59%, ** = 60-79%, *** = 80-100%.

The FreeMind test delivered results regarding the accuracy of dependency detection at the level of depended-upon classes and at the more detailed level of dependency constructs within the code. First, all tools were able to report violations at the level of "from-class makes use of to-class". However, only one of the ten tools reported usage of all seventeen classes used by class ScriptingEngine, while the least well-performing tools reported 12 classes only. On average, the ten tools were able to report 82% of the 17 depended-upon classes. Table 5.14 summarizes the results, based on the following scales: Detected classes * = 0-79%, ** = 80-89%, *** = 90-100%.

Second, none of the tools was able to detect dependencies for all 109 constructs within class ScriptingEngine to package "freemind". On the average, 78 of 109 dependencies (72 percent) were reported. However, the five tools within this test (the other five tools did not report detailed enough information at the level of dependencies) differed considerably in their performance. JITTAC and Sonargraph performed relatively well and reported respectively 91 dependencies (83.4 percent) and 90 dependencies (82.6 percent), while SAVE reported 75 dependencies (69 percent), Structure101 73 (67 percent), and Lattix 63 (58 percent). These numbers will not be higher for the other tools, as far as we were able to ascertain, based on the reported depended-upon classes in the violation reports of these tools.

Table 5.14: Relative comparison of the tools on the tested characteristics (the scales are explained in the related subsections)

	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure101
Accuracy dependency detection										
Benchmark: Direct dependencies	*	***	**	***	**	**	*	**	***	***
Benchmark: Indirect dependencies	*	**	*	**	*	*	*	*	*	***
Freemind: Detected classes	*	**	*	***	*	**	**	**	**	**
Freemind: Detected dependencies				***	*		**		***	**
Accuracy dependency messages										
Graphical overview of violations	**		**	**	**		**		**	**
Exactness of dependency messages	*	*	*	***	**	*	**	**	***	**

Table 5.14 summarizes the results, based on the following scales:
Detected dependencies: * = 0-59%, ** = 60-79%, *** = 80-100%.

Exactness of Violation and Dependency Messages
The FreeMind test revealed that the ten tools differ considerably in the way violations and dependencies are reported, how many messages are reported, and how much information is reported in the messages. Consequently, the answer to research question RQ4 is diverse, a few ACC tools report the type and location of violations and dependencies quite exactly, but most tools not.

All tools, except Dependometer, Macker and Sonar ARE, provide diagrams in which violations are shown at the level of packages and classes. Some of these tools even provide an indication of the quantity of underlying dependencies, by means of a number or by the thickness of a line. In our opinion, such an indication of the severity of a violation is relevant information for architects and management; information that also should be included in violation reports. Table 5.14 summarizes the results. In case graphical support is provided, two asterisks are shown.

We regard the exactness of a dependency message to be high, if the message helps to locate the dependency causing code construct accurately in the source code. Four tools (ConQAT, Dependometer, dTangler and Macker) only provide information on the from-class and the used to-class. The other tools provide more information: SAVE and Structure101 provide also the names of the invoking method at the from-side and the invoked method at the to-side, while JITTAC, Lattix, Sonar ARE and Sonargraph Architect indicate the line number, and JITTAC and Latix even the position of the dependency-causing construct in the line. Table 5.14 summarizes the results. One asterisk is shown, in case only from-class, to-class information is provided. Two asterisks are shown, in case also method from, method to information is provided. Three asterisks are shown, in case the line number is provided, and/or the position of the dependency causing construct in a line. For reason of usability issues, described in Section 5.5, we have valued Lattix and Sonar ARE with two instead of three asterisks.

Hard-To-Detect Dependency Types and their Frequency in Open Source Systems
The answer to research question RQ5 is positive: yes, there are hard-to-detect types of dependencies. We identified ten dependency types of which several tools failed to report instances of dependencies. Analysis of the number of dependencies per type in open source systems has yielded interesting data. The average distribution of dependencies in four reference systems over the six main types (import, declaration, call, access, inheritance, annotation) is respectively 10, 20, 39, 26, 3,

and 2 percent. Below is a summary of the findings related to hard-to-detect types of dependencies.

- The ten hard-to-detect types of dependencies in our test account for at least 47 percent of the dependencies in the reference systems. Inner class related dependencies, also hard-to-detect, are not included in this percentage, since there may be an overlap with the already included dependencies.
- A considerable fraction of the dependencies within the reference systems is inheritance related, namely twelve percent on average, while 3% of the dependencies is inheritance related and indirect.
- A considerable fraction of the dependencies within the reference systems is inner class related, namely nine percent on average.
- A considerable fraction of the dependencies within the reference systems is indirect, namely 16 percent on average. In the benchmark test, only 60 percent of the indirect dependencies were detected, on average.

5.7.2 Challenges in Dependency Detection

Based on the results of the benchmark test and FreeMind test, we have identified challenges in dependency detection. Analysis of the most common shortcomings in dependency detection revealed the challenges, which are discussed below.

C1: *Report dependencies accurately in case of inheritance structures*
The test results show that inheritance structures frequently hamper the accuracy of dependency detection. This finding is relevant, since our analysis of four reference systems showed that twelve percent of all reported dependencies were inheritance related. Furthermore, three hard-to-detect types of dependency are inheritance related. Together, these three types account for 8 percent of all the reported dependencies.

The results of the benchmark test show that only three of the ten tools were reporting a dependency on the super class in case of a call of an inherited method, or in case of access of an inherited variable. These three tools reported a dependency of these types as a dependency to the super class where the method is actually implemented, while the other seven tools reported a dependency to the addressed subclass, but not to the super class where the method was implemented. Moreover, the results of the FreeMind test show that many method calls of inherited instance methods were not detected at all (43 percent, on average); neither as dependency on the addressed subclass, nor as dependency on the super class where the method is actually implemented.

The relevance for ACC is considerable. In case of a compliance check, the seven tools will fail to report a violation if the used subclass is part of an allowed-

to-use module, while the super class that has implemented the called method or accessed variable, is part of a not-allowed-to-use module. In such cases, a strong dependency stays unnoticed.

Finally, no tool reported indirect inheritance dependencies. The relevance for ACC is considerable, since no tool reported a violation in case the super class of the from-class was part of an allowed-to-use module, while the super class of this parent super class in the inheritance hierarchy was part of a not-allowed-to-use module. This appears as inconsistent behavior of the tools, since all tools reported a violation in case the first super class of the from-class was part of a not-allowed-to-use module. In our opinion, a violating indirect inheritance relation should be reported, since it marks a strong dependency, and changes may have substantial consequences.

C2. *Report dependencies accurately in case of inner classes*
The test results show that inner classes also may hamper the accuracy of dependency detection. This finding is relevant, since our analysis of four reference systems showed that nine percent of all reported dependencies were inner class related.

The results of the FreeMind test show that four tools reported no dependency at all on inner classes, while fourteen violating dependencies on inner classes were present in the code of class ScriptingEngine. In the majority of these cases, a dependency to the outer class instead to the inner class is reported, with as consequence a diminished traceability to the related code constructs in the source code. Furthermore, dependencies between inner classes of the same outer class will not be reported.

C3. *Report relevant object references*
The results of the benchmark test show that seven tools had problems with the detection of dependencies of type “Variable access, object reference”. Dependencies of this type are frequently included in the code as parameter values (arguments), or reference variables within if clauses.

Our analysis of four reference systems showed that fourteen percent of all reported dependencies were of this type.

The results of the FreeMind test show the practical implications: two of the five tools in this test missed all sixteen dependencies of this type. These sixteen missed object references represented 15 percent of the 109 dependency causing constructs in class ScriptingEngine.

In our opinion, it is a good practice to filter out object and type references, which precede method calls and variable access. Most tools do, HUSACCT too, since a dependency message for such a reference doubles with the dependency on

the same type for the call or access. However, a reference needs to be reported in case of standalone references, e.g. when an object is passed as a parameter value.

C4. *Report information that is missing in compiled files*
We encountered several situations where tools failed to report dependencies accurately, just because information in source files is removed from the compiled files. Tools that analyze compiled files only, were not able to report dependencies of three dependency types: a) dependencies caused by import statements; b) dependencies caused by declaration statements of not initialized local variables; and c) dependencies caused by access of constant variables (instance and class).

Import dependencies were reported by only two tools in the benchmark test. In our opinion, import statements should be reported, since they cause coupling, although weak. The analysis results of the number of dependencies per dependency type in five open source systems show a high frequency of import dependencies; nine percent of all reported dependencies were caused by import statements.

Moreover, reporting import dependencies enhances the accuracy of violation reporting in these cases where a tool fails to report dependencies of specific types. In a number of situations in the benchmark test, SAVE missed a dependency of a specific type, but reported a correct violation message at class level, merely based on the import statement.

Finally, the exactness of the messages with respect to the line number is diminished if a tool does not make use of source code. For example, in the FreeMind test, the line numbers of dependencies reported by Lattix (which makes use of compiled files only) were by far not as accurate as the line numbers reported by Sonargraph (which makes use of source files, in addition to compiled files).

5.8 Threats to Validity

To discuss the validity of the results of our laboratory experiments, we make use of the four validity threats, as described and defined by Wohlin et al. (2012).

5.8.1 Construct Validity

Construct validity is concerned with the relation between theory and observation. The experiment should be suitable to answer the research question, which in our case means that the experiments should be suitable to answer the main question "How accurate do ACC-tools report dependencies and violations against dependency rules?" We have ensured the construct validity in several ways. First, by starting with an inventory of dependency types in object oriented code, on which we have based the test cases of the benchmark test. The set of dependency types included in our benchmark test is no random set, but a carefully chosen set of

34 types; large enough to assess the sensitivity of the tools. However, we do not claim that our classification of dependency types is complete, since dependencies may be established by many different types of code constructs in object oriented programs.

Second, we have taken care that the code constructs in each test case of the benchmark test are specific for the related dependency type. Furthermore, we have taken care that false negatives could not be caused by code constructs that were not specific to the related dependency type.

Third, to answer research questions RQ1 – RQ3, we have scored for each test case not only the ability of a tool to report a dependency, but also the ability to report a violation.

Fourth, we have complemented the benchmark test with the FreeMind test, and we have cross- checked the results of both tests, as described under the results of the FreeMind test.

5.8.2 Internal Validity

Internal validity is threatened, if certain influences have affected the results, without the researcher's knowledge. In the context of our experiments, the internal validity may be threatened by the inclusion of problematic code constructs in the benchmark test, or by the use of very uncommon source code in the FreeMind test.

In our opinion, the strict design of our custom-made benchmark test strengthens the internal validity. In this test, each test case is specific for one dependency type, and each test case has one separate from-class in the test code. This approach proved to be valuable, especially to test tools that provide only messages with a low level of exactness, thus with no more information than the from-class and to-class. Another aspect in favor of the internal validity of the benchmark test is that all test cases were detected by at least one tool, except in case of indirect "Inheritance, extends-implements variations" (although two of the three cases represented quite common situations).

The FreeMind test adds to the internal validity, since it contains several code variations not included in the benchmark test. We used it to validate and extend the benchmark test. The FreeMind test showed in several cases that a tool might fail to detect a dependency in a complex real life application, while it is able to detect a dependency of the same type in a simpler situation within the same application, or in the relatively simple code of the benchmark test application.

5.8.3 Conclusion Validity

Conclusion validity reflects on the relationship between the treatment and the outcome of the experiment. In favor of conclusion validity, no statistical operations

were needed to interpret the results of our experiments. The research questions could straightforwardly be answered, based on the results of the benchmark test and FreeMind test.

To secure the validity of the identified challenges in dependency detection, we have substantiated each challenge by means of data and arguments.

5.8.4 External Validity

External validity reflects the extent to which the experiment results may be generalized. Since we did not work with a randomized selection of tools, our study can be characterized as a quasi-experiment, according to Wohlin et al. (2012). Consequently, our findings may not be generalized to other tools, even though we tested ten tools in a small market. Also, be aware that our findings may not be generalized to newer versions of the tested tools; the performance of the tools may improve. Furthermore, our findings are limited to Java code analysis and should not be generalized to tools that analyse code of other programming languages.

5.8.5 Comparison of the Frequency of Dependencies per Type

In favor of the internal, external and conclusion validity, we have compared the frequency of dependencies per type in the FreeMind system and its class ScriptingEngine to the average of four reference systems. In our opinion, this comparison confirms that the FreeMind system and its class ScriptingEngine are suitable for research on the accuracy of dependency detection. The system as a whole contains dependencies of many different types and in proportion to the average percentages of the reference systems. The same applies for class ScriptingEngine, although one should keep in mind that this class contains ten percent more hard-to-detect dependencies than the reference systems on average. Even so, the wide variety of dependency-causing constructs, including a set of complex constructs, makes it an appropriate subject class for the test.

However, limitations apply to the validity of the analysis results of the four reference systems and FreeMind. We cannot guarantee that all dependencies in these systems are reported and that all dependencies of a type are reported. We have ensured the validity of the analysis results by upgrading HUSACCT to the level (and beyond) that all test cases in the benchmark test and all dependencies in ScriptingEngine were reported. However, since many code variations are possible, some variations may not be reported. Furthermore, deficiencies may be present in HUSACCT itself or in the included open source (ANTLR based) lexer and parser functionality. For instance, a small percentage of the classes is skipped by the parser, because of unexpected (and often erroneous) code in these classes.

5.9 Related Work

Callo Arias et al. (2011) state that dependency analysis approaches that identify structural dependencies have a high degree of accuracy. Our research outcome shows that it is appropriate to be aware of the limitations of the tools used. Practitioners and academics rely on tools for their work. It is not hard to get impressed by the output of these tools, but it is hard to get an impression of what is missing in the output of a tool. Our study demonstrates that the tested tools will not always provide a 100 percent accurate output. Other comparative tool studies also show that static analysis tools and techniques are not always accurate. For instance, Sutton and Maletic (2007)compared four tools that reverse engineer C++ source code into UML models. The numbers of recovered classes and relationships differed by about 20 percent and much more for attributes, operations, and generalizations.

Rutar et al. (2004) compared five bug finding tools for Java, and they reported false positives, false negatives, redundant warnings and only 15-33 percent overlap between the tools. Compared to the set of bug finding tools, the ACC-tools in our test perform better, with no false positives and no redundancy, but with differences in output and quite a number of false negatives.

According to Binkley (2007), source code analysis is impeded by the complexities of modern programming languages. Barowski and Cross (2002) pay special attention to dependencies on virtual members and on synthetic methods in their paper on the extraction and use of class dependency information for Java. Our study confirms that their special attention is justified, since these types of dependencies (on super classes and inner classes) are involved in many unreported dependencies and violations.

Another topic in their paper is source file versus class file based dependency extraction, and they describe some differences between both forms. For their own tool, they choose for class file based extraction. We do not object to this choice, but we advise, based on our study, to include source code too in the analysis of ACC-tools. In order to optimize the accuracy of the tool with respect to import statements, constant variables, local variables, and the exact position of a dependency-causing construct in the source code.

The FreeMind system has been used in several other scientific studies. We compared our analysis results with these studies, but we did not find overlap, except for comparison on the number of packages and classes of FreeMind version 0.9.0. Emanuel and Surjawan (2012) analyzed all versions of FreeMind to illustrate the use of their revised Modularity Index. Their counts of version 0.9.0 match our counts quite closely. We counted 58 packages and 853 classes (including inner classes, but excluding anonymous classes), which is around ten percent more than

their counts. The difference may be explained by the fact that Emanuel and Surjawan analyzed compiled code, while we analyzed source code with inclusion of test files. Zoller and Schmolitzky (2012) used Freemind 0.9.0 also in a study, however they counted only 445 classes. On the other side, Arlt et al. counted 1,362 classes (2012), much more than reported in the other studies. Summarizing, we noted large difference in class counts, while the counted numbers of NCLOC differed not more than 25 percent between the three studies.

With respect to our analysis data of the four reference systems, we found some interesting studies. Tempero et al. focused on the use of inheritance in Java systems in two empirical studies (Tempero et al. 2008), (Tempero et al. 2013) of more than 90 open-source systems. They found high levels of use of inheritance, with about three out of four types being defined using inheritance. Furthermore, they found that the inheritance structures are used actively, also for what we typified as access of an inherited variable or call of an inherited method. In line with their research, our study has revealed a high percentage of inheritance related dependencies in the four reference systems. On average, nine percent of the dependencies are caused by access of inherited variables and calls of inherited methods.

Tempero conducted another large study (Tempero 2009) to investigate whether the advice is followed to avoid non-private fields. It is good practice to prevent usage of attributes of other classes, since it compromises encapsulation (Wirfs-Brock and Wilkerson 1989). The results of his study indicate that it is not uncommon (albeit not that terribly common) to declare non-private fields. In line with his findings, our study shows that access of an attribute of another class accounts for about five percent of the dependencies in the four reference systems.

Dyer et al. (2013) conducted a large-scale empirical study on the adoption of Java language features. With respect to annotations, these results showed that annotations were among the most used new features of the last three Java versions. However, they noted a relative lack of custom annotations. In line with their work, our study showed that annotations with a reference to another type (internal or external) accounted for two percent of the dependencies in the four reference systems.

5.10 Conclusion

Architecture compliance checking (ACC) relies on the support of tools to define modules and rules, analyze the code, check the compliance, and report violations to the rules. In this study, we have investigated to which extent static ACC-tools report dependencies and violations accurately. We classified dependency types, prepared a benchmark test, and tested ten tools based on this benchmark test. In addition, we have tested these tools based on the program code of open source system FreeMind, which we used to test the ability of the tools to report all depended-upon classes, all dependency-causing constructs, and all information needed to locate dependency-causing constructs in the source code.

5.10.1 Answers to the Research Questions

We started our study with the following question in mind: How accurate do ACC-tools report dependencies and violations against dependency rules? In the Introduction, this main question was decomposed into four research questions, which are answered as follows:

RQ1: *Do ACC tools find all the dependencies between modules in the software (no false negatives)?*

No, the ten tools detected on average 77 percent of the dependency types in the benchmark test; 83 percent of the 25 direct types, and 60 percent of the 9 indirect types. Furthermore, the tools detected on average 72 percent of the 109 constructs with dependencies in a class of FreeMind. All ten tools were able to detect dependencies established by basic constructs, like method calls and type declaration. However, our study showed also that relevant numbers of violating dependency constructs were not reported. For example, in the FreeMind test, the tool with the lowest scores missed 46 out of 109 constructs, while even the best scoring tool missed 18 constructs. Consequently, all tools may improve the accuracy of dependency detection.

The tools differ considerably in their ability to detect all types of dependencies. For instance, in the benchmark test, JITTAC and Structure101 detected 91 percent of all *dependency types*, while ConQAT and SAVE detected respectively 62 and 53 percent. In the FreeMind test, JITTAC and Sonargraph Architect reported dependency messages for respectively 83 percent of the 109 violating code constructs, while Structure101 and Lattix detected 67 and 58 percent.

RQ2: *Do ACC tools report all the violating dependencies in the software (no false negatives)?*

No, since no cases were noticed during the tests, where a tool detected a dependency, but failed to report it in case it violated an architectural rule. Consequently, the answer to the previous research question is here valid too.

RQ3: *Do ACC tools report non-violating dependencies as violations (false positives)?*

No, during the benchmark test no false positives were detected. No tool interpreted allowed dependencies in the program code as violating dependencies. In addition, nearly no errors in the violation messages were identified during the tests; only a few messages contained incorrect information.

RQ4: *Do ACC tools report the exact type and location of violations and dependencies?*

The answer to this research question is diverse. Only four tools provide a dependency type that differentiates between types of usage, like declare, access, or call. The number and names of dependency types vary per tool. Furthermore, only a few tools report the location of dependencies exactly. All tools report violations to dependency rules at the level of from-class, to-class, but at this level of abstraction, one message may represent several dependencies. Six tools also provide dependency details in reports or IDE plug-ins, but not always precisely enough to localize dependencies discretely.

RQ5: *Are there types of dependencies, which proved hard-to-detect by several tools?*

Yes, based on our tests, we identified ten hard-to-detect types of dependencies, which were each missed by several tools. To substantiate the relevance of our findings, we performed an analysis of the number of dependencies per dependency type in five open source systems. The analysis results revealed that the hard-to-detect types of dependencies account for at least 47 percent of the dependencies in the reference systems.

5.10.2 Challenges

Since significant percentages of false negatives were revealed per tool during the benchmark test and FreeMind test, we have analyzed the test results in detail. As outcome, we have identified and described four challenges in dependency detection. In summary, the test results revealed that the most common shortcomings in dependency detection, encountered in our study, have to do with: 1) inheritance structures; 2) inner classes; 3) object references; 4) missing information in compiled files.

Our tests have shown that inheritance structures and inner classes hamper the accuracy of violation reporting in many cases. A dependency caused by usage of inherited methods or variables is often not reported, and if reported, than mostly as dependency on the accessed subclass only and not on the super class that implements the method or variable. Furthermore, usage of an inner class is frequently not reported at all, and if reported, it is most often reported as a dependency on the outer class instead of the inner class, which diminishes the traceability in the source code.

5.10.3 Future Work

Benchmark test are relevant to advance the state of the arts of tools. We have developed and applied initial tests to benchmark tools on the accuracy of dependency detection and reporting. The testware of our benchmark test and FreeMind test is available on request. Future work can be based on these tests and be aimed at the development and application of comprehensive benchmark tests for a wide variety of tools and a wide variety of programming languages.

Research on the performance and improvement of dependency analysis is relevant for practitioners and academics, since dependency analysis supplies the data not only for ACC, but also for architecture reconstruction, metrics, and architecture restructuring advice.

Chapter 6

A Typology Based Approach to Assign Responsibilities to Software Layers

In software architecture, the Layers pattern is commonly used. When this pattern is applied, the responsibilities of a software system are divided over a number of layers and the dependencies between the layers are limited. This may result in benefits like improved analyzability, reusability and portability of the system. However, many layered architectures are poorly designed and documented. This chapter proposes a typology and a related approach to assign responsibilities to software layers. The Typology of Software Layer Responsibility (TSLR) gives an overview of responsibility types in the software of business information systems; it specifies and exemplifies these responsibilities and provides unambiguous naming. A complementary instrument, the Responsibility Trace Table (RTT), provides an overview of the TSLR-responsibilities assigned to the layers of a case-specific layered design. The instruments aid the design, documentation and review of layered software architectures. The application of the TSLR and RTT is demonstrated in three cases.

6.1 Introduction

The Layers pattern, or Layered style, is one of the most common patterns used in software architecture (Clements et al. 2010, Harrison & Avgeriou, 2008). The concept of layering can be traced back to the works by Dijkstra (1968) and Parnas (1972). Buschmann et al. described the Layers pattern extensively (1996). Avgeriou and Zdun (2005) have shown that layers are also described as patterns or styles by many other authors. For a definition we cite Larman (2005) who summarized the essential ideas of the Layers pattern as: *"Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns such that the 'lower' layers are low-level and general services, and the higher layers are more application specific. Collaboration and coupling is from higher to lower layers; lower-to-higher layer*

coupling is avoided". An example of a layered design, shown in Figure 6.1, represents a strict layered design in which the following usage rules are respected: usage relationships are from top to bottom; neither back call nor skip call usage is allowed.

Applying the Layers pattern will improve software qualities, like analyzability, reusability and portability of the system, but may also impose liabilities like a lower efficiency, or more rework when a change affects several layers. When a layered view of architecture is drawn up, a number of design decisions must be taken, preferably explicit and documented (Kruchten et al. 2006a). For example, the following design questions should be answered:

- Which layers are distinguished?
- Which types of responsibility are assigned to each layer?
- Which usage relationships between the layers are allowed?

Unfortunately, the layered designs are often poorly defined and many violate the principles for which layers are designed (Clements & Nord 2000, Clements et al. 2010). In practice and in student projects, we encounter many layered designs describing or showing only the layers and the names of the layers, without a specification of the contents, the communication rules, and a justification. Such an architectural product gives little guidance to the developers. For instance, another layered model with three layers (Presentation, Domain, and Technology), could represent the same-layered design, as the one shown in Figure 6.1, but it could be substantially different as well. The names of the layers do not clarify the exact responsibilities of the layers, e.g. where the control of the task is located, or where the status is maintained. Therefore, a specification of the responsibilities of the layers is needed.

We aimed our research on the analysis of the causes of these problems and on the provision of instruments to design layered architectures of high quality. One cause of the problems is described by Clements et al. (2010): "The layered view of architecture, shown with a layer diagram, often is poorly defined, and so often misunderstood". In addition, we hypothesize another cause, namely that the terminology regarding layered designs is not clear and sometimes downright

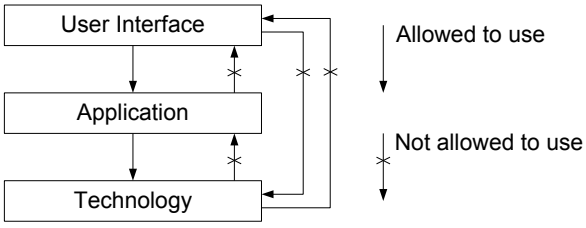

Figure 6.1: Example of a strict layered design

contradictory. A uniform classification is lacking; different authors use varying and sometimes conflicting terms for layers, types of logic and types of responsibilities. A good example is the popular concept "application logic" or "application layer". Application logic is interpreted substantially diverse by different authors. Larman (2005) describes it as: “handles presentation layer requests; workflow; session state; window/page transition; …”, whereas Erl (2008) defines it as: “an automated implementation of business logic …” Even, the concept of "domain layer" has different meanings in different layered designs.

The starting point of our investigation was the observation that to answer the design question “*Which types of responsibility are assigned to each layer?”,* a uniform classification for the naming and characterization of types of responsibilities in software layers could be useful. This perception resulted in the following research questions, which were leading in our study: 1) What types of responsibilities are distinguished in layered architectures; 2) How can these types of responsibility be named and defined unambiguously; 3) How can a typology of responsibilities be applied in practice?

To answer these research questions, we studied leading literature about software layers to get an overview of common types of responsibilities and the names given to them. Based on this literature, we constructed the Typology of Software Layer Responsibility (TSLR). The TSLR gives an overview of the different types of responsibility, gives them unambiguous names, specifies them and exemplifies them. To enhance the application of the TSLR in practice, we designed the Responsibility Trace Table (RTT). An RTT shows the assignment of TSLR-responsibilities to the layers of a case-specific architecture, without the need for extensive textual descriptions. The TSLR and RTT may be used to design and document a particular layered design and to assess the quality of existing layered designs. Finally, to evaluate and improve the typology and its related trace table, the instruments were reviewed by experts in the field of software architecture, applied in case studies, and used in training courses for bachelor students.

In this chapter, we propose our typology in Section 6.2, and an approach to apply the TSLR for different practical purposes in Section 6.3. Next, Section 6.4 illustrates the application of the TSLR and RTT by means of three cases. Thereafter, Section 6.5 discusses our research approach, the artifacts and the limitations, and Section 6.6 presents the conclusions and an outlook to future work.

6.2 Typology of Software Layer Responsibility

The Typology of Software Layer Responsibility (TSLR) provides an inventory of distinct types of responsibility commonly found in the software of business information systems. The TSLR identifies and orders responsibilities, gives them unambiguous names and specifies them. The included responsibilities are distilled from leading literature on layers in the field of software architecture.

The TSLR consists of a classification schema and a textual specification of the responsibilities. The TSLR is not intended as a layered design itself, but is intended to be used as a reference, when a system's layered design is drawn up or reviewed. For example, a software architect may specify the responsibilities of a system specific layered design in terms of the TSLR responsibility types, which saves work, and check his design on completeness, which may contribute to the quality of the layered architecture. Furthermore, the TSLR is intended to help software engineers to determine the responsibility type of a concrete fragment of functionality within a system. An important competence, since functionalities have to be mapped to design units with appropriate responsibilities, in conformance of the software architecture, and consecutively to the software units that implement these design units.

6.2.1 Responsibility

Responsibility in the context of software architecture is defined by Clements et al. (2010) as "a general statement about an architecture element and what it is expected to contribute to the architecture. This includes the actions that it performs, the knowledge it maintains, the decisions it makes or the role it plays in achieving the system's overall quality attributes or functionality". We adopt this definition and the notion, from the same source, that a *layer* (as all software architecture modules) is characterized by its set of responsibilities.

The TSLR provides an overview of *logical* responsibility types. Logical means here "free of implementation choices", since the typology is not intended for a specific platform, orientation or deployment strategy, like in tier models. The layered style is a modular style (Clements et al. 2010) and does not focus on the runtime behavior, or the allocation of software components. The TSLR is primarily aimed to be useful in the context of business information systems, since most literature used as basis for the typology is focused on this type of systems. The typology may also be useful for embedded systems, control systems, games, et cetera, but extensions may be necessary.

6.2.2 Classification Schema

The classification schema, shown in Figure 6.2, provides an overview of the responsibility types distinguished within the TSLR. The responsibility types are specified in the next sub-section. Three levels of abstraction are distinguished within the classification schema:

6. Top level, where Software System responsibility represents all the logical responsibilities of a software system.
7. Intermediate level, where compound responsibility types reside: the five main types of responsibility: Consumer Interface responsibility, Task Specific responsibility, Domain Generic responsibility, Infrastructure Abstraction responsibility and Infrastructure responsibility. The main types of responsibility are distinguished on the basis of potential reuse.
8. Bottom level, where sub-responsibilities represent singular types of responsibility. The TSLR distinguishes thirteen sub-responsibilities within the five main types of responsibility.

6.2.3 Specification of the Responsibility Types

The main types of responsibility are specified below, together with their sub-responsibility types. For each main type of responsibility, design criteria are specified to provide guidance, when the TSLR is used in practice, to identify the responsibility type of a concrete fragment of functionality. The design criteria focus on reusability, since the primary criterion to differentiate between the different main types of responsibility is reuse.

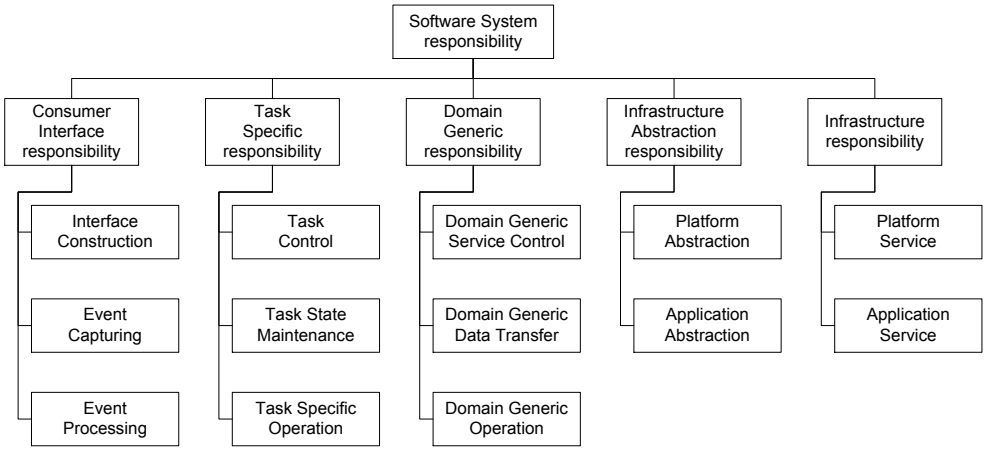

Figure 6.2: TSLR classification schema

Consumer Interface Responsibility
Description: Consumer Interface responsibility takes care of establishing and maintaining communication with a consumer of a system service in a manner appropriate to the task of the consumer. The consumer can be an end user communicating via a user interface, but also an automated client system communicating via a service interface. Consequently, interface and events may have different forms.

Design criteria: Responsibility is...
- Included, when it is specific to the interface of a task.
- Excluded, when it is reusable across different interfaces, which support the same task.

Table 6.1: Sub-responsibilities of Consumer Interface responsibility

Sub-responsibility	Description	Examples
Interface Construction	Provide an interface to the consumer with information and/or control appropriate to the task of the consumer.	• GUI building • Report layout and presentation • Speech interface • Service interface
Event Capturing	Capture events from the consumer.	• Recognizing input from consumer: data, control, speech • Knowledge about when an event is captured
Event Processing	Process events from the consumer as far as it concerns Consumer Interface responsibility.	• Deciding what to do with the input (data, speech ...) • Format check on input data • Delegation

Task Specific Responsibility
Description: Task Specific responsibility takes care of the coordination of the task, the maintenance of the task state and the execution of functionality specific to the task. A task is a unit of work, to be performed as a whole, which provides the consumer with a result of value. A task is generally, in terms of Cockburn (1997), at the user-goal level. Task Specific responsibility is potentially reusable across different interfaces (e.g. on different platforms) of the same task.

Design criteria: Responsibility is...
- Excluded, when it is specific to a task interface.
- Included, when it is specific to a task.
- Excluded when it is potentially broadly reusable.

Table 6.2: Sub-responsibilities of Task Specific responsibility

Sub-responsibility	Description	Examples
Task Control	Coordinate the task. Decide what needs to happen when an event takes place.	• Workflow, orchestration, page flow • Control of task specific sub-responsibilities • Delegation
Task State Maintenance	Track and maintain the task state.	• Tracking which data is selected, inserted, or changed • Transaction state control
Task Specific Operation	Perform actions and transformations exclusive to the task.	• Conversion of data • Task specific constraints • Task specific transformations and computations, like calculating report totals, joining data, sorting data

Domain Generic Responsibility

Description: Domain Generic responsibility is responsible for the coordination and the execution of functionality dealing with concepts, information and rules of the business. Domain Generic responsibility has to do purely with the problem domain and is potentially reusable across different tasks.

Design criteria: Responsibility is...

- Excluded, when it is specific to a task.
- Included, when it is specific to the business.
- Excluded when it has knowledge of infrastructure that has to be abstracted.
- Excluded, when it is reusable across different business applications.

Table 6.3: Sub-responsibilities of Domain Generic responsibility

Sub-responsibility	Description	Examples
Domain Generic Control	Coordinate the activities needed to handle requests.	• Control of domain generic sub-responsibilities • Delegation
Domain Generic Data Transfer	Retrieve and store data.	• Selecting and sorting data • Storing new or changed data
Domain Generic Operation	Execute domain generic actions and transformations.	• Generic constraints • Generic transformation rules • Maintaining entity state

Infrastructure Abstraction Responsibility
Description: Infrastructure Abstraction responsibility is responsible for the translation of infrastructure independent requests into requests dependent on the infrastructure. Infrastructure Abstraction responsibility is separated from other responsibility types, when needed to meet quality requirements like portability, analyzability, and reusability.

Design criteria: Responsibility is...
- Excluded, when it is specific to a domain or task.
- Included, when it has knowledge of infrastructure that has to be abstracted.
- Excluded, when it is part of an infrastructure platform or infrastructure application.

Table 6.4: Sub-responsibilities of Infrastructure Abstraction responsibility

Sub-responsibility	Description	Examples
Platform Abstraction	Encapsulate functionality dependent on an application platform element.	• Adapter to a specific database • Functionality formatted to make use of a specific object relational mapping framework • Adapter a specific security framework
Application Abstraction	Encapsulate functionality dependent on an infrastructure application.	• Adapter to a specific electronic mail client • Adapter to a specific document editor

Infrastructure Responsibility
Description: Infrastructure responsibility is responsible for broadly reusable functionality, non-specific to the business. It may be bought, but also self-built, e.g. utilities. Since there are a huge number of infrastructural services, the TSLR connects here with the TOGAF Technical Reference Model (TRM) (The Open Group 2009). The TRM defines and exemplifies the concepts Infrastructure Application and Application Platform. Furthermore, it provides a typology of the services of the Application Platform.

Design criteria: Responsibility is...
- Excluded, when it is specific to a business application.
- Included, when it is reusable across different applications and/or businesses.

Table 6.5: Sub-responsibilities of Infrastructure responsibility

Sub-responsibility	Description	Examples
Platform Service	Provide generic application support (by the technology components of hardware and software).	• Data interchange service • Data management service (DBMS, OODBMS, ORB …) • Network service • Operation System Service • Software engineering service (Programming language …) • Security service (Identification, Authentication …)
Application Service	Provide general-purpose business functionality (by Commercial Off-The-Shelf software).	• Electronic mail client • Document editing and presentation • Spreadsheets • Workflow management

6.2.4 Justification of the TSLR and Related Work

Founding Literature

The responsibility types in the TSLR are distilled from leading literature in the field of software architecture and layers. We performed a literature study based on the search strings "software", "layer", "architecture", "responsibility" and their synonyms in various combinations. We found the most valuable sources to be books, well-established in the software architecture community; an experience matching with a systematic literature review conducted by Savolainen and Myllärniemi (2009).

The first category of sources consulted in the course of this investigation describes the basics of the Layers pattern and guidelines to design a layered architecture (e.g., Bass et al. 2012, Buschmann et al. 1996, Clements et al. 2010, Evans 2004, Fowler et al. 2003, Larman 2005, Shaw & D. Garlan 1996). Authors often refer to these sources, when the subject of layers is addressed. Evans (2004), Fowler et al. (2003) and Larman (2005) extensively describe layered designs suitable for business systems. Evans distinguishes four layers, Fowler three layers, while Larman specifies six common layers in an information system's logical architecture. The second category of sources describe a specific layered design, useful within the scope of this study (e.g., Allen & Frost 1997, Gorton 2006, MSDN 2009, Snoeck et al. 2000, The Open Group 2009). The third category of sources discusses layered designs for service oriented architectures (e.g., Erl 2008, Krafzig et al. 2005, Lankhorst 2009, Winter & Fischer 2007). Service layers cannot be compared straightforwardly with software layers, since they do not focus on the

internal structuring of an application, but distinguish services at different levels of abstraction.

Design Decisions with Regard to the Main Types of Responsibility
The names of the main types of responsibility within the TSLR are intended to be as clear and unambiguous as possible. Terms used in the founding literature, like application logic, business logic and even domain do not make clear what is meant by them, and the terms are used for quite different responsibilities (Larman 2005). An interesting example of different definitions of “domain” and “business logic” by two authors in the same book makes clear that the responsibilities can differ considerably. Fowler describes domain as "logic that is the real point of the system”, “also referred to as business logic" (Fowler et al. 2003, p. 20). On the other hand, Stafford divides "business logic into two kinds: domain logic, having to do purely with the problem domain (such as strategies for calculating revenue recognition on a contract), and application logic, having to do with application responsibilities"(Fowler et al. 2003, p. 134).

To prevent confusion, new, semantic-rich names are chosen within the TSLR. *Consumer Interface responsibility* is selected as name, because it is a semantically rich name. Furthermore, it is not frequently used and burdened, like Presentation and User Interface, which are often used as names for layers not only given Consumer Interface responsibilities, but also given task specific responsibilities.

Task Specific responsibility is business logic, exclusive for a task and not commonly reusable. It maps to the second kind of business logic in Stafford's description. The term "Task Specific responsibility" is derived from the term task-centric service as used by Erl (2008).

Domain Generic responsibility within the TSLR maps to the first kind of business logic in Stafford's description. The term is chosen to make clear that this type of business logic is broadly reusable within the software system, contrary to Task Specific responsibility. The distinction between Task Specific responsibility and Domain Generic responsibility is also described by Alan and Frost (Allen and Frost 1997) as the distinction between user and business service, and by Larman (2005) as the distinction between the application layer and domain layer.

The distinction between Infrastructure Abstraction responsibility and Infrastructure responsibility also requires some explanation. Evans (2004) distinguishes one Infrastructure layer only. On the other hand, Larman (2005) distinguishes three layers containing broadly reusable logic (Business Infrastructure, Technical Services, Foundation), but the criteria used, are a bit vague. Within the TSLR, the *Infrastructure Abstraction responsibility* is specific to

the business application, while *Infrastructure responsibility* is not specific and enables reuse across different business applications.

TSLR Meta-Model

The meta-model of the TSLR, as shown in Figure 6.3, matches the structure of the proposed typology. The meta-model allows for possible future extensions in width and depth and may be used to provide tool support. The composite pattern (Gamma et al. 1995) is used to allow extensions in the hierarchical structure. Each *Responsibility* in the typology has a name and description. *CompoundResponsibilities* represent the main types of responsibility, each composed of a set of SingularResponsibilities, and *designCriteria* help to determine which responsibilities are included or excluded. *SingularResponsibilities* represent specific responsibilities in our typology, and they are illustrated by *examples*.

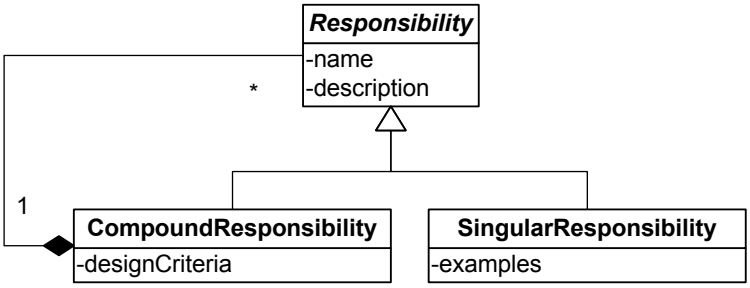

Figure 6.3: TSLR meta-model

For reasons of comprehensibility, *designCriteria* and *examples* are modeled as attributes, although they may contain multiple values.

6.3 Approach to apply the TSLR with the Responsibility Trace Table

The TSLR is intended to aid the design, documentation and review of layered software architectures. The Responsibility Trace Table (RTT) enhances the application of the TSLR in practice. In this section, the RTT is introduced and exemplified at first. Next, the application areas of the TSLR and RTT are discussed.

6.3.1 Responsibility Trace Table

An RTT shows the assignment of the TSLR-responsibilities to the software layers of a case-specific layered design. The main types of responsibility with their sub-responsibilities are represented as columns and the software layers as rows. An X within an intersecting cell of a TSLR responsibility and a layer shows the assignment of the TSLR-responsibility to the layer. The advantage of the trace table is that it provides an overview of the responsibilities of the software layers, without the need for extensive textual descriptions, since the responsibilities are defined within the TSLR. An example of an RTT is included as Table 6.6. It shows the responsibilities of the three principal layers as described by Fowler et al. (2003) and shown in Figure 6.4. The analysis of this layered design is discussed in the next section.

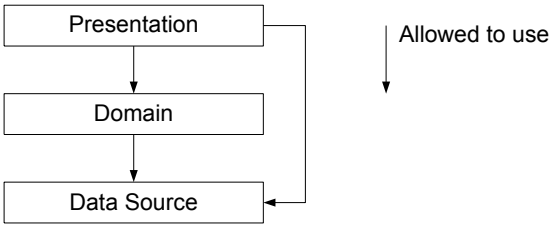

Figure 6.4: Layered design based on the three principal layers(Fowler et al. 2003)

Table 6.6: Responsibility Trace Table linking Fowler's three principal layers to the responsibilities defined by the TSLR

Main Type of Responsibility →	**Consumer Interface**			**Task Specific**			**Domain Generic**			**Infrastructure Abstraction**		**Infrastructure**	
TSLR Responsibility → / **Software Layer ↓**	Interface Construction	Event Capturing	Event Processing	Task Control	Task State Maintenance	TS Operation	DG Service Control	DG Data Transfer	DG Operation	Platform Abstraction	Application Abstraction	Platform Service	Application Service
Presentation	X	X	X										
Domain							X	X	X				
Data Source										X	X		

6.3.2 Application areas

Design of Software Layers
During the design of layered software architectures, a number of design decisions, described before in the Introduction section, have to be taken. The TSLR and RTT aid the decision and documentation regarding the design question: *Which types of responsibility are assigned to each layer?*

The TSLR gives a complete overview of the assigned responsibilities per layer and can be used to consider alternatives and to decide on clear-cut separations of concerns per layer. The RTT shows the assignment of the TSLR-responsibilities to the application-specific layers, and this overview supports reasoning about the architecture. Finally, the documentation of the responsibilities of the layers of a system specific layered design may be prepared by the combined use of the TSLR and RTT. An RTT makes it easy to an architect to complement the graphical representation of the layered design with a definition of the responsibilities of the layers, without much documentation.

Analysis of Layered Designs
Another application area is the analysis of existing or proposed layered designs. TSLR and RTT are useful to gain a clear insight into the division of the responsibilities over the software layers. The RTT is very useful here, since it shows omissions and redundancies in the assignment of responsibilities to the layers within a software architecture. This helps to evaluate the quality of the layered design and the effectiveness in achieving the quality requirements.

Training
Finally, the TSLR and RTT may be helpful in the training of students, software engineers and architects. We used the TSLR and RTT in software architecture courses for third year bachelor students Computer Science. Drawing up or implementing a layered design requires knowledge of the different types of responsibilities. We use the TSLR and some assignments to let the students acquire this knowledge. Furthermore, we discuss the suitability of several layered designs to meet specified quality requirements, and we discuss proposals for layers in student projects. The visual character of the TSLR's classification schema, and the overview provided by an RTT, support the explanation and discussion of different design alternatives regarding the assignment of responsibilities to layers.

6.4 Applications

Three cases are described below to illustrate the practical use of the TSLR and RTT.

6.4.1 Fowler's Three Principal Layers

To demonstrate the applicability of the TSLR and RTT as supporting tools for the *analysis of an existing layered design*, we use Fowler's "Three Principal Layers". This layered design, discussed in the previous section and shown in Figure 6.4, serves well for this purpose, since it is extensively described (Fowler et al. 2003, pg. 19-22) and well known. The translation of the description of the three layers into TSLR responsibilities was fairly easy. The resulting Responsibility Trace Table, shown in Table 6.6, provides a good overview of the responsibilities per layer. Two observations are interesting to discuss.

The first observation is that the Task Specific responsibility is not assigned to a layer, which should be regarded as an omission in the definition of a layered design. The description of the layers makes clear that the Presentation layer and Domain layer include all responsibilities of respectively Consumer Interface responsibility and Domain Generic responsibility from the TSLR. However, the definitions of Presentation and Domain do not make clear where the Task Specific responsibility is allocated. In later chapters, it appears that Task Specific responsibility may be included in both layers, Presentation and Domain, depending on the pattern chosen. The Application Controller Pattern assigns Task Specific responsibility to the Presentation layer. In terms of the TSLR, an Application Controller contains Task Specific responsibility, since its two main responsibilities are defined as "deciding which domain logic to run", and "deciding the view with which display the response". The Service Layer Pattern is used to organize the Domain and assigns Task Specific responsibility to the Domain layer. A service layer "encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations". In terms of the TSLR a services layer contains Task Specific responsibility, especially since it "typically includes logic that's particular to a single use case".

The second observation is that the name of the third Principle Layer, the Data Source layer, does represent only a part of its contents. A more general name should enhance the interpretability of this layer, since it is not only responsible for the communication with data sources in the infrastructure, but also for the communication with the rest of the infrastructure, like transaction monitors, other applications, and messaging systems.

6.4.2 Layered Design of HUSACCT

To demonstrate the applicability of the TSLR and RTT as supporting tools for the *design of a layered architecture*, we use the case of the development of HUSACCT (HU University Software Architecture Compliance Checking Tool). We have been working on HUSACCT for several years during a specialization semester "Advanced Software Engineering" for third year bachelor students. HUSACCT can be used to: 1) Define the intended modular architecture: layers, subsystems, components, external systems, and rules constraining their properties and relations (Pruijt et al. 2013a); 2) Analyze the implemented architecture embedded in the source code (Java, C#); and 3) Validate the compliance between intended and implemented architecture.

Based on the requirements, the layers model was drawn up, shown in Figure 6.5, and the responsibilities per layer were specified in a RTT, shown in Table 6.7. The layered design of HUSACT, combined with a domain model and a logical component model, served well to address the key requirement, divide the work over six teams, and identify and specify the required communication between the system's components.

The Task layer includes two main types of responsibility from the TSLR: Consumer Interface responsibility, and Task Specific responsibility. The restrictions of two strictly separated layers for these responsibilities seemed to impose more cons than pros. For reasons of analyzability, separate packages were created for these types of responsibility within the Task Layer, but the communication between these packages is not restricted by the default rules of a layered design.

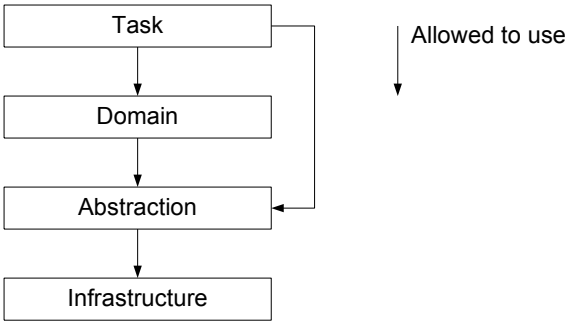

Figure 6.5: Layered design of HUSACCT

Table 6.7: Responsibility Trace Table of HUSACCT's Layered design

Main Type of Responsibility →	**Consumer Interface**			**Task Specific**			**Domain Generic**			**Infrastructure Abstraction**		**Infrastructure**	
TSLR Responsibility → / **Software Layer ↓**	Interface Construction	Event Capturing	Event Processing	Task Control	Task State Maintenance	TS Operation	DG Service Control	DG Data Transfer	DG Operation	Platform Abstraction	Application Abstraction	Platform Service	Application Service
Task	X	X	X	X	X	X							
Domain							X	X	X				
Abstraction										X			
Infrastructure												X	

The abstraction layer was introduced to implement the analysis of source code as programming language independent as possible; since an important requirement was that the tool should be expandable with regard to the analysis of other programming languages. Since certain processes within the Task layer need direct access to abstracted infrastructural services, a skip call is allowed from the Task layer to these services of the Abstraction Layer. The layered design of HUSACCT illustrates that the number and names of system specific layers do not have to match the TSLR's types of responsibility.

6.4.3 Layered Architecture of a Large Software System

To demonstrate the applicability of the TSLR and RTT for large systems, we use a case of a complex, governmental administration system, aimed at a user base of approximately 6000 end users and distributed across 75 different physical locations. The system's layering schema, part of the well-documented software architecture, is shown in Figure 6.6. The layers and their constituting components are described concisely below. The mapping of the layers and their components to the TSLR responsibilities was done in retrospect, and the result is visible in the Responsibility Trace Table, shown in Table 6.8.

The system's architecture was based primarily on the Microsoft .Net reference architecture for .Net version 1.1. However, it deviated from Microsoft practices in the following manner: a) the system made heavily use of object orientation conform the domain model pattern (Fowler et al. 2003) to handle the large quantity of business rules in a classic OO style; b) the system was split up in a Smart Client application and a Business Domain Server application.

The Smart Client is the implementation of the *User Interface layer*, responsible for capturing input and calling the Business Domain Server through web services. It was designed to be user-friendly while having only a minimal amount of business knowledge. The User Interface layer contains five types of components. User Interface Components (1) are responsible for showing data to the users, for collecting and syntactical validating data entered by the user and for interpreting events. The UI Process Components (2) are responsible for coordination of the user process and management of the process state. Client business entities (3) are the "less intelligent" cousin classes of the Business Entities found in the domain server. They typically have very little domain knowledge and are used to enforce required fields, field formats, restrict list values etc. The business domain server provides an xml structure, which states what fields are required and what list selections are appropriate for the given state of the object being viewed. UI Service Agents (4) have the responsibility of making available data to the system and can be seen as a courier used to handle the conversation with the Domain Server. User Interface Data Access Logic Components (5) are responsible for providing access to the data cache in this layer. This cache is used to minimize bandwidth usage and overall response time.

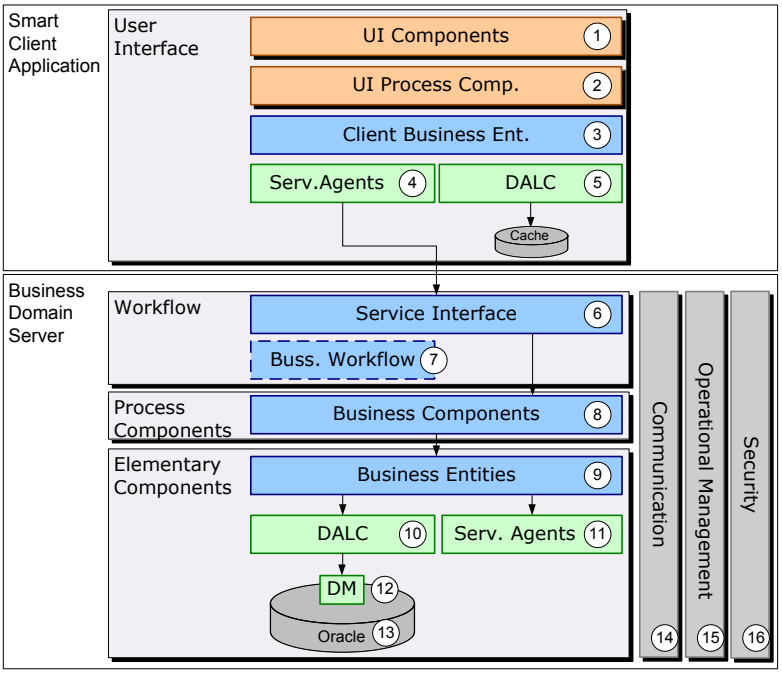

Figure 6.6: Layering schema of the case system

Table 6.8: Responsibility Trace Table of the case system's principal layers and components

Main Type of Responsibility →		**Consumer Interface**			**Task Specific**			**Domain Generic**			**Infrastructure Abstraction**		**Infrastructure**	
TSLR Responsibility → **Software Layer ↓**		Interface Construction	Event Capturing	Event Processing	Task Control	Task State Maintenance	TS Operation	DG Service Control	DG Data Transfer	DG Operation	Platform Abstraction	Application Abstraction	Platform Service	Application Service
User Interface	1	X	X	X										
	2				X	X	X							
	3								X	X				
	4										X			
	5										X			
Workflow	6	X	X	X										
	7											X		
Process C	8				X	X	X	X						
Elementary C	9								X	X				
	10										X			
	11										X			
	12								X		X			
	13												X	
Crosscutting	14												X	
	15												X	
	16										X			

The Business Domain Server is responsible for processing the requests from the client and other channels, while maintaining integrity and security. It is organized in three layers and three cross cutting concerns. The *Workflow layer* is responsible for coordinating workflow in a future release of the system. In the current release, it only had the responsibility to provide a facade to access the underlying domain functionality. Service Interface Components (6) provide the services that the application offers in a simple, secure manner and hide the underlying system implementation. The service interfaces are implemented with web services. The Business Workflow component (7) is included to enable the integration of an external workflow application in a future release of the system. The *Process*

Components layer is responsible for coordinating the processing of single business events that happened during the workflow. This layer contains two types of Business Components (8). Task Controllers are responsible for controlling the underlying Process Controllers and persisting (or undoing at a failure) all activities as one transaction. Process Controllers are reusable business activities and are responsible for coordinating the data transformations.

The *Elementary Components layer* contains Business Entities (9), responsible for maintaining the integrity of the information, and Data Access Logic Components (10), EC Service Agents (11) and Data Mappers (12), responsible for storage and retrieval of the information in the Oracle database (13) or in external systems.

The *Crosscutting Concerns* (MSDN 2009) Communication (14), Operational Management (15) and Security (16) were handled by services of the application platform infrastructure. Only access to the security library was abstracted by means of service interface classes.

6.5 Discussion

Since the research was intended to deliver an artifact relevant to the professional practice, our study can be characterized as design-science research (Peffers et al. 2008). Based on a practical problem, we defined research questions, studied leading literature about software layers, designed instruments in line with this literature, and evaluated the instruments.

To evaluate the typology and its related trace table, the instruments were reviewed by experts, applied in practical cases, and used in training courses for bachelor students. Five experts in the field of software architecture reviewed our proposals on completeness and accuracy. In their responses, they provided useful feedback, which was used for improvements. Some names were discussed and changed, descriptions were improved and examples added. A lot of discussion focused on an unambiguous name for the first type of responsibility in the TSLR and resulted in "Consumer Interface responsibility". In addition to the expert review, the completeness and accuracy of the TSLR was evaluated by means of a case study of the software architecture of a large software system. The outcome of the evaluation was positive; no responsibilities were found missing in the TSLR, and the arrangement and naming of the responsibilities appeared accurate. The mapping of the system's responsibilities on these of the typology required several iterations in which the architect's knowledge of the TSLR was deepened, as well as the researcher's knowledge of the particular software architecture. In this process, the trace table proved to be a valuable instrument. The visual overview supported

architectural reasoning and helped to recognize omissions and redundancies in the initial versions of the system's RTT.

There are some limitations to our research so far. One important limitation is that our research focused on responsibilities of the software of business information systems. Therefore, other types of systems, like embedded systems and games, might contain responsibilities not included in the TSLR. Another limitation has to do with the completeness of the typology. Despite our extensive literature study and validation activities, we cannot ensure that all types of responsibilities, common in business information systems, are represented in the TSLR. However, future additions and evolution are taken into account; the meta-model of the typology enables extensions in width and depth. Finally, the typology could be viewed and used as a layered model. However, *the typology is not intended to be a template for layered designs, with layers exactly matching the main types of responsibility of the typology.* Layered designs in practice should be designed to meet the specific requirements of the system. The number and names of the required layers may vary, the responsibilities per layer may vary, and a layer may contain sub-responsibilities from different main types of responsibility within the TSLR.

6.6 Conclusions

In this chapter, we proposed two novel instruments to support software architects in their task to design layered architectures of high quality: the Typology of Software Layer Responsibility (TSLR) and the complementary Responsibility Trace Table (RTT). These instruments, together with some illustrations of their practical use, provide answers to the research questions, which formed the basis of this study. We started with the observation that the terminology regarding layered designs is not clear and sometimes contradictory. We finished with a proposed typology and a trace table to aid the practical use of the typology.

The TSLR provides an overview of the distinct types of responsibility commonly found in the software of business information systems. The TSLR may be used when a layered design is drawn up and when an existing layered design is analyzed or reviewed. Furthermore, it is useful in training courses to discuss and exercise the different possibilities to divide responsibilities over the layers and their impact on the quality characteristics of the software system. The TSLR responsibilities are distilled from leading literature on layers in the field of software architecture. The TSLR separates and groups the responsibilities, gives them unambiguous names, specifies them and exemplifies them. At the level of infrastructural responsibilities a connection is established to the TOGAF Technical

Reference Model (The Open Group 2009), which classifies a huge number of infrastructural services.

The Responsibility Trace Table (RTT) shows the assignment of the TSLR responsibilities to the different software layers. The RTT is an instrument to complement a system's graphical representation of the layered design with a specification of the responsibilities of the layers. In addition, the RTT may be used to assess and enhance the quality of a layered design, since it shows omissions and redundancies in the assignment of the responsibilities.

To illustrate the application of the instruments three cases were presented: a design case, a review case, and a complex case of al large governmental software system. These cases were also used to evaluate the completeness, accuracy, and applicability of the instruments. Furthermore, experts in the field of software architecture conducted a review, and the instruments were used in training courses for bachelor students.

Further research may be aimed on the applicability and scope of the TSLR and RTT. At first, it will be interesting to study the effectiveness of the TSLR and RTT when practitioners and students apply these instruments. Next, to enlarge the field of application of the TSLR, literature and case studies are needed on the responsibilities of other types of software systems (other than business information systems). Finally, it will be interesting to study the applicability of the instruments in the context of other software architecture patterns.

Chapter 7

The EARScorecard – An Instrument to Assess the Effectiveness of the EA Realization Process

Enterprise Architecture (EA) is a well-accepted, but relatively young discipline. To add value to the organization, an Enterprise Architecture Management (EAM) function should be able to realize its goals in line with the corporate strategy. In this chapter, we propose the Enterprise Architecture Realization Scorecard (EARS) and an accompanying method to discover the strengths and weaknesses in the realization process of an EAM function. During an assessment, representative EA goals are selected, and for each goal, the results, delivered during the different stages of the realization process, are identified, examined and scored. The outcome of an assessment is a numerical EARScorecard, supplemented with a description of the strengths and weaknesses of the EA realization process, and recommendations. To evaluate and improve the assessment instrument, the EARScorecard was used in various organizations. An assessment case is discussed in depth to illustrate the use of the instrument.

7.1 Introduction

Enterprise Architecture Management (EAM) forms a means to enhance the alignment of business and IT and to support the managed evolution of the enterprise (Buckl et al. 2009). Enterprise Architecture (EA) can be defined, according to the ISO/IEC 42010 (ISO 2007), as "the fundamental organization of [the enterprise] embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution". A number of enterprise architecture frameworks have been proposed, including The Open Group Architecture Framework (The Open Group 2009), DoDaf (Department of Defense 2009), GERAM (FAC-IFIP Task Force 1999), the Zachman Framework (Zachman 1987), and many more, as described by Chen, Doumeingts and Vernadat (Chen et al. 2008).

Over the last decades, EAM is introduced in many organizations, but the introduction and elaboration often do not proceed without problems, and most practices are still in the early stages of maturity (Bucher et al. 2006; Van Steenbergen et al. 2010). Moreover, the performance of the EAM function typically is not measured (Winter et al. 2010). Therefore, research on the evaluation of EAM (e.g., Luftman 2000; Morganwalp and Sage 2004; Van der Raadt et al. 2007; Van Steenbergen et al. 2007) and improvement of the effectiveness (e.g., Foorthuis et al. 2010; Lankhorst 2009) is useful, since it may contribute to the further development of the professional practice.

Our study builds on this line of research and aims to contribute to it by the design of a product with practical relevance. The study was initiated as part of a larger study on the value of EA, sponsored by the Dutch Government and three profit organizations. We started from the notion, that to add value to the organization, an EAM function should be able to realize its goals. Next, we focused our work on the research question: How can we measure the EAM function's ability to realize its goals?

Two core concepts call for some elaboration: 'EAM function' and 'effectiveness of EA'. The EA (Management) function is extensively defined by van der Raadt and van Vliet [20]: "The organizational functions, roles and bodies involved with creating, maintaining, ratifying, enforcing, and observing Enterprise Architecture decision-making – established in the enterprise architecture and EA policy – interacting through formal (governance) and informal (collaboration) processes at enterprise, domain, project, and operational levels."

The effectiveness of EAM can be viewed, defined and measured in many different ways (Morganwalp and Sage 2004). The EARS approach focuses on the EA realization process, and states that this process is effective, when the EAM function is able to transform a given baseline situation into a target situation, as specified by one or more goals set out to the EAM function; see Figure 7.1. These EA goals, or in terms of TOGAF (The Open Group 2009) “requests for architecture

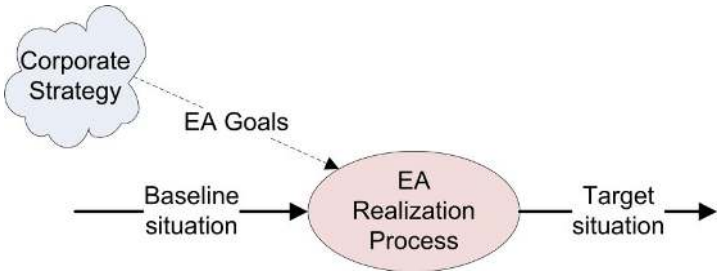

Figure 7.1: The EA realization process of an EAM in context

work", should be aligned with the corporate strategy. There is a large variety in type and scope of goals set to different EAM functions.

To answer the research question, we developed and evaluated an instrument to assess the effectiveness of the EA realization process: the Enterprise Architecture Realization Scorecard (EARS). Applying the instrument includes consecutively: a) selecting some representative EA goals; b) identifying and examining the results produced in the context of an EA goal; c) scoring the results on the basis of indicators; and d) describing the strengths and weaknesses of the EA realization process. An EARS assessment is primarily used for awareness and improvement regarding the effectiveness of the EAM function, but may also be used for governance with respect to the progress and quality regarding the realization of an EA goal.

The EARS instrument is not designed for specific types of EA goals, but is intended to be applicable for all types of EA goals. Some examples of EA goals are as follows: "The organization should be able to implement a change in legislation within three months", and "The application portfolio has to be rationalized to reduce costs".

The research approach applied to develop the EARS is that of design-science research (Hevner et al. 2004, Peffers and Tuunanen 2008), since our research was intended to deliver artifacts relevant to the professional practice. The applied approach conformed to the seven guidelines of Hevner et al. For instance, the design of the EARS was discussed and evaluated in two meetings with ten experts from the professional field, and in four meeting with participants from four different academic institutions. Furthermore, EARS assessments were conducted at large organizations to evaluate the applicability.

7.1.1 Related Work

A number of instruments with similar objectives is developed and proposed, like EA balanced scorecard (Schelp and Stutz 2007), EA maturity models (Luftman 2000, Ross 2003, Van Steenbergen et al. 2010), and EA analysis approaches (Buckl et al. 2009, Johnson et al. 2007). The main difference between the balanced scorecard approach and the EARS approach is that the balanced scorecard approach is concerned only with the outcome (added value) of EAM, while the EARS approach is also concerned with how the outcome is achieved.

The main difference with the maturity approach is that this approach measures the effectiveness of the EA realization process indirectly (assuming that when a certain maturity level is reached for each key area, the EAM function will operate effectively), while the EARS approach aims to measure the effectiveness of each step in the EA realization process directly, by assessing the results.

The main difference with the EA analysis approaches is that, expressed in terms of Buckl's classification schema (Buckl et al. 2009), most of them have a specific Analysis Concern, often a specific quality attribute, and have a related specific Body of Analysis, while in the EARS approach the Analysis Concern and the Body of Analysis will vary per EA goal. Furthermore, the EARS approach does not only focus on EA artifacts, but on all results of the EA realization process, including acceptance of architectural decisions, outcomes of architecture conformance checks, et cetera.

Another line of research, interesting to our study, attempts to relate benefits of EAM to applied techniques/mechanisms (e.g., Foorthuis et al. 2010, Lange et al. 2012, Tamm et al. 2011, Van der Raadt et al. 2010). The focus of this line of research is on general benefits of EAM, like reduced resource duplication, or improved agility. In contrast, the EARS approach focuses on goals set to a specific EAM function. Though, a general benefit can be included in the set of EA goals of an EAM function, and then its realization can be measured within an EARS assessment, especially when the EA goal is defined specific and measurable.

The next section of this chapter presents the EARS instrument, while the two following sections successively describe the method to apply the instrument and illustrate the application by means of a case study. The section thereafter discusses the research so far and its limitations. The last section concludes the article and provides an outlook to future work.

7.2 The EARS Instrument

The objective of the EARS approach is to measure the actual achievement of the EAM function regarding the realization of one or more EA goals. During an assessment, the focus is on the results of the EA realization process, because results show the actual effect of the EAM's work. In this section, we will discuss the concept of the instrument, explain the instrument, and provide a formal description of the instrument.

7.2.1 Concept of the EARS

The research question "How can we measure the EAM function's ability to realize its goals?" can be answered in different ways. One option is to measure the final result (changes in business operation) only and answer the question: To which extent is the operational performance matching with the target values of the EA goal?

The advantage of this approach is that it seems to be straightforward and relative simple. However, there are a number of disadvantages. Only goals that are

realized completely will be eligible for a measurement. Additionally, it is not made plausible that the final results may be attributed to EAM. Moreover, the resulting score does not give any grips for the causes, and consequently also not for improvements. Therefore the option 'measuring the final result only' was rejected and the alternative option was chosen: measure at a more detailed level. To find the best way to do this, the body of knowledge of (IT) governance was used, since measuring the organizational and IT performance is a well-established practice within this field. CobiT (IT Governance Institute 2007a) appeared to be especially useful for this study. It is an open standard for IT Governance, well accepted both in practice and in the academic world. The CobiT framework is based on the following principles: business-focused, process-oriented, controls-based and measurement-driven. These principles are extensively explained in the CobiT 4.1 Excerpt (IT Governance Institute 2007b). Transfer of these CobiT-principles to the field of EA resulted in a metamodel, shown in Figure 7.2, and a set of principles. Together, these form a concept, which enables measurement of the EAM function in achieving its goals, at a detailed level.

- *EA goals* are derived from the business goals and enterprise strategy.
  EA goals should best be specific, measurable, actionable, realistic, results-oriented and timely.
- *EA goals* are realized through a (repeatable) *EA realization process*.
- The *EA realization process* is composed of a logical sequence of *EA activities*.
- Per *EA activity* an *activity goal* and related *metrics* are specified.
  The metrics are primarily focused on the *result* of the *EA activity*.

7.2.2 EA Realization Process

In the EARS, the EA realization process is decomposed into five EA activities, each with a distinctive, assessable result. Examination of these five results provides sufficient insight into the realization process to discover its strengths and weaknesses, and to identify case specific point for improvements. Figure 7.3 shows the EA activities (depicted by rectangles) with their results, and Table 7.1 describes their characteristics.

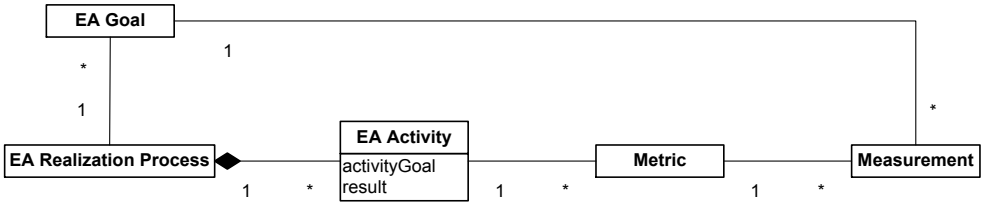

Figure 7.2: Metamodel of the EARS concept

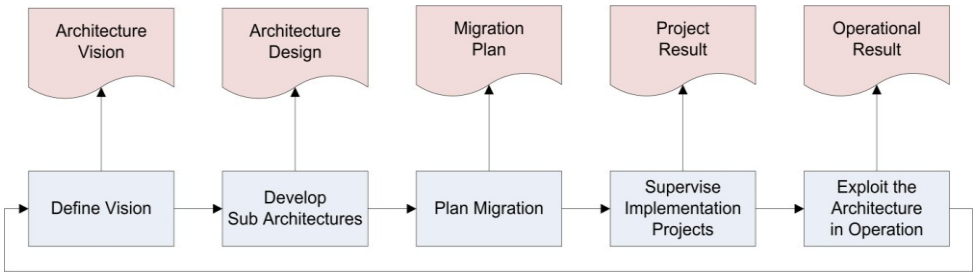

Figure 7.3: The five EA activities with their results

The five EA activities were derived from the Architecture Development Method (ADM) of TOGAF 9 (The Open Group 2009), because ADM offers an architecture development cycle that covers all life cycle aspect as required by GERAM (Saha 2004). To ensure completeness of the set of EA activities and results, other sources (e.g., Wagter et al. 2005, Winter and Fischer 2007) were also studied, and we used expert meetings to validate our proposals. Although EARS is based on TOGAF, its EA realization process differs from TOGAF's ADM. EARS distinguishes five EA activities, while ADM recognizes nine phases, so the mapping (shown in Table 7.1) is not one to one. The first two EA activities of EARS can be linked plainly to four ADM phases.

Table 7.1: Characteristics of the five activities

Id	**EA Activity**	**EA Activity Goal**	**Result**	**ADM Phase**
#1	Define Vision	Determine the EA goals within scope of the architecture iteration, develop a high level, integrated and approved solution direction towards matching these goals and create a concise realization plan.	Architecture Vision	A
#2	Develop Sub Architectures	Develop the required subsets of architectures to support the agreed architecture vision.	Architecture Design	B, C, D
#3	Plan Migration	Search for opportunities to implement the architecture and plan the migration.	Migration Plan	E, F
#4	Supervise Implementation Projects	Ensure conformance to the architecture during the development and implementation projects.	Project Result	F, G
#5	Exploit the Architecture in Operation	Assess the performance of the architecture in operation, ensure optimal use of the architecture, and ensure continuous fit for purpose.	Operational Result	G, H

For the last three EA activities, coupling is more complex. For instance, within ADM Phase G, Implementation Governance, the architecture is implemented within the solution under development, but the solution also is implemented in the operational environment within phase G. These two results are considered as very different within the EARS and consequently they are measured separately.

7.2.3 Aspects

During an EARS assessment, the five results are identified, examined, and valued. To enable a balanced way of measuring, for each result three aspects are scored separately: product, acceptance, and scope. Separate scoring of these aspect is practical, since an architect can design a top quality solution (product aspect), but if it is not accepted (acceptance aspect), nothing is gained. On the other hand, if the top quality solution is limited to one architectural domain, like technology, while other domains are affected as well (scope aspect), the goal may never be realized. The three aspects, with their focus, question and scale, are defined in Table 7.2.

Table 7.2: The aspects to be valued per result

Result Aspect	Focus/Question	Scale
Product	Focus: The accuracy of the outputs and the completeness, in terms of depth. Question: To which extent will the EA-goal be realized with it?	1-10
Acceptance	Focus: The acceptance and commitment of the stakeholders. Question: To which extent do they know, understand and agree with the product, and do they act committed?	1-10
Scope	Focus: The completeness, in terms of width, of the outputs. Question: Is the output width sufficient to realize the goal?	1-10

7.2.4 EARScorecard

For each EA activity result, the three aspects are scored separately, and these scores are recorded at the EARScorecard, which summarizes the assessment outcome. An example of a scorecard is shown in Table 7.3; the outcome of an assessment at a large financial organization; further explained in Section 7.4.

Most scores in an EARScorecard are at a scale of 1-10, where 1 stands for a minimal contribution of an aspect or result to the realization of the EA-goal (10 percent or less of what might be achieved), and 10 for an optimal contribution (100 percent). The intermediate values allow differentiation, where 5 (just insufficient) and 6 (just sufficient) mark boundary values. Since the scores represent substantiated opinions and not exactly measured data, the numbers are rounded off

to integers. The well-known scales for the scores and totals enhance the easy interpretation of the outcomes.

The aspect scores of product, acceptance and scope are specified by the assessor(s), whereas the totals in the scorecard are calculated, based on the aspect scores. The *aspect total* is calculated as the multiplication of the *aspect score* (product or acceptance) with the *scope score* for a result of a goal, divided by 10. The *result total* is calculated as the average of the two *aspect totals* for a result of a goal. Finally, the sum of the *aspect totals* constitutes the *goal total*, expressed on a scale from one to hundred. The general question during an EARI assessment is: To which extent is an EA goal realized and can this be related to the effort of the EA function? A satisfying answer to this question should lead to high scores for the result totals and consequently for the goal total. The value of the goal total can be used to mark progress regarding the EA goal, but it will not show the underlying reasons for the score, which can be very diverse. Generally, more interesting are the other totals and scores of an assessment, as they show the strengths and weaknesses of the EAM function in achieving its goals.

Table 7.3: EARS scorecard of the EA goal "Implement a corporate data warehouse"

Id	Result	Aspect	Aspect score	Scope score	Aspect total	Result total
#1	Architecture Vision	Product	8	8	6	5
		Acceptance	5		4	
#2	Architecture Design	Product	3	6	2	2
		Acceptance	2		1	
#3	Migration Plan	Product	5	2	1	1
		Acceptance	5		1	
#4	Project Result	Product	7	1	1	1
		Acceptance	6		1	
#5	Operational Result	Product	4	1	1	1
		Acceptance	3		1	
	Goal total				**19**	

7.2.5 Indicators

To support the assessors and to standardize the rating, indicators were developed for each combination of result and aspect. TOGAF's ADM (The Open Group 2009) was used as main source, since it provides elaborate descriptions of objectives, intent, approach, activities, artifacts, inputs and outputs for each phase (Saha 2004). As additional source, (Wagter et al. 2005) was used. The sets of indicators for result #1 - #5 are shown respectively in Table 7.4 – 7.8. They are also utilizable in cases where TOGAF ADM is not used. The technique of scaled coverage percentage (van Zeist et al. 1996) was used to classify the indicators and prioritize them with relative weights (W). The relative weights total to 1.0 per aspect.

Table 7.4: Set of indicators of result #1, Architecture Vision

Aspect	Id	Indicator	W
Product	1	The EA-goal is related to the business strategy and included in the vision.	0.2
	2	The EA-goal is SMART and (if needed) decomposed into high level stakeholder requirements.	0.2
	3	A high level solution direction is described and the solution direction to the goal is accurate.	0.2
	4	The solution direction to the goal is integrated with the solution directions of other goals.	0.3
	5	A comprehensive plan exists to realize the solution direction.	0.1
Acceptance	1	The architecture vision is well-known by the stakeholders.	0.2
	2	The stakeholders understand the vision, the solution direction to the goal and its implication.	0.2
	3	The stakeholders agree with the solution direction to the goal and its implications.	0.3
	4	The stakeholders feel committed to (this part of) the vision.	0.3
Scope	1	The architecture vision covers all aspects relevant to the goal: business, data, application, and/or technology.	1.0

Table 7.5: Set of indicators of result #2, Architecture Design

Aspect	Id	Indicator	W
Product	1	The baseline architecture is described.	0.2
	2	The parts affected by the goal, are identified.	0.1
	3	The target architecture is described and the solution to the goal is accurate.	0.2
	4	The solution to the goal is integrated with the solutions of other goals.	0.2
	5	The architectural artefacts are specific enough to substantiate architectural decisions.	0.2
	6	A gap analysis (impact analysis) is included.	0.1
Acceptance	1	The architecture design is well-known by the stakeholder.	0.2
	2	The stakeholders understand the solution to the goal and its implication.	0.2
	3	The stakeholders agree with the solution to the goal and its implications.	0.3
	4	The stakeholders feel committed to the architectural solution.	0.3
Scope	1	The architecture design covers all aspects relevant to the goal: business, data, application, and/or technology.	1.0

Table 7.6: Set of indicators of result #3, Migration Plan

Aspect	Id	Indicator	W
Product	1	An architecture roadmap to realize the goal is defined, and, if needed, the transition architecture is described.	0.3
	2	The work packages needed to realize the goal are assigned to projects in the project portfolio, and, if needed, specified in an implementation and migration plan.	0.7
Acceptance	1	The decision makers agree with the architecture roadmap and related plans.	0.3
	2	The decision makers include the required work packages into the project portfolio.	0.7
Scope	1	All the work (work packages, projects) needed to realize the goal is included in the migration plan. If not, consider the ratio between: a) work included, and b) all the work needed for the goal.	1.0

Table 7.7: Set of indicators of result #4, Project Result

Aspect	Id	Indicator	W
Product	1	The architecture definition and architecture requirements relevant for the goal are provided to the project(s), and are specific enough to direct decisions of the project architects.	0.3
	2	The architectures needed for the goal are realized in conformance to the architecture definition and the architecture requirement. Exceptions or changes are approved by the EA function.	0.7
Acceptance	1	The goal and architectural solution is well-known by the project architects.	0.2
	2	The project architects understand the architectural solution to the goal and its implication.	0.2
	3	The project architects agree with the architectural solution to the goal and its implications.	0.3
	4	The project architects feel and act committed to implement the architectural solution.	0.3
Scope	1	All the work (work packages, projects) needed to realize the goal is completed by the projects. If not, consider the ratio between: a) work completed, and b) all the work needed for the goal.	1.0

Table 7.8: Set of indicators of result #5, Operational Result

Aspect	Id	Indicator	W
Product	1	Improvements related to the goal are achieved in the operational environment, in conformance to the target performance indicators.	0.8
	2	The performance is still conform goal after x years of operation. Work needed to reach this result is organized and executed.	0.2
Acceptance	1	The stakeholders are satisfied with the realized operational environment.	1.0
Scope	1	All intended improvements related to the goal are achieved in the operational environment. If not, consider the ratio between: a) achieved improvements, and b) all the improvements related to the goal.	1.0

7.2.6 Arguments

Indicators aid the assessor, but have a high level of abstraction, since they have to be useful for very different types of EA goals. Consequently, an indicator score needs substantiation as well, which is included in the EARS approach by means of arguments. Arguments represent case specific evidence, assembled during interviews and examinations of architectural artifacts. Arguments are sorted per indicator and recorded in tables. Per argument a contribution (positive, negative, neutral) is documented, and preferably also a reference to its source (interviewee or document). A set of arguments is shown in Table 7.11.

7.2.7 Formal Description of the EARS

The EARS instrument is composed of the instantiations of *EA Realization Process*, *EA Activity*, *Aspect*, *Metric,* and *Indicator* in the final metamodel, shown in Figure 7.4. There is only one *EA Realization Process* and its processGoal is, to realize an *EA goal*, regardless of what the goal may be.

Instantiations of *EA Goal*, *Measurement,* and *Argument* are specific to an assessment. The Goal Question Metric approach [2] was taken into account, but no separate entity Question is included, because the questions at Aspect do satisfy in combination with the activity goals and the EA goal. The terms metric and measurement are often used in a quantitative approach, but in CobiT [12] they are

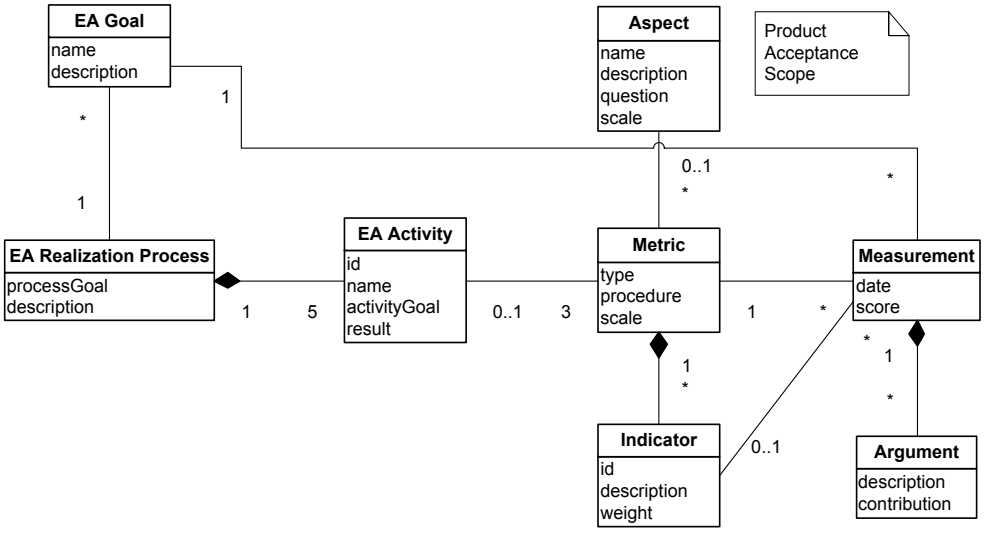

Figure 7.4: Final EARS metamodel

also used for qualitative usage, which is also the usage within the EARS approach.

Most metrics within the EARS describe how an aspect of a result of an EA activity can be measured. The metrics, needed to calculate the totals of the EARS scorecard, are described below.

First, the notations are introduced:

- Let $G = \{g_1, g_2, ..., g_n\}$ be the set of EA goals.
- Let $R = \{r_1, r_2, ..., r_5\}$ be the set of Results of the EA Activities of the EA Realization Process.
- Let $A = \{product, acceptance, scope\}$ be the set of Aspects.
- Let $PA = \{pa_1, pa_2\}$ be the subset of A containing *product* (pa_1) and *acceptance* ($pa_{2)}$ only.

Subsequently, the scores and totals can be defined as follows:

- The aspect score expresses the score for the product or acceptance aspect for a result of a goal:
  aspect_score is a function from G x R x PA to {1, …, 10}
- The scope score expresses the score for the scope aspect for a result of a goal:
  scope_score is a function from G x R to {1, …, 10}
- The aspect total can be calculated as the multiplication of the aspect score (product or acceptance) with the scope score for a result of a goal, divided by 10:
  aspect_total is a function from G x R x PA to [1, 10]
  $aspect\_total(g, r, pa) = (aspect\_score(g, r, pa)$ x $scope\_score(g, r))/10$
- The result total can be calculated as the average of the aspect totals for a result of a goal:
  result_total is a function from G x R to [1, 10]
  $result\_total(g, r) = (aspect\_total(g, r, pa_1) + aspect\_total(g, r, pa_2))/2$
- The goal total can be calculated as the sum of all the aspect totals of a goal:
  goal_total is a function from $G \rightarrow$ [1, 100]
  $goal\_total(g) = \sum_{i=1,j=1}^{5,2} aspect\_total(g, r_i, pa_j)$

The scales of the EARS are chosen as specified, because decimal scales are often used and quite understandable. Therefore, they enhance correct valuing and correct interpretation of the scores. Since the scores do represent substantiated opinions and not exactly measured data, the numbers are rounded off to integers.

7.3 Method

The process to execute an EARS assessment is summarized below. The main line of this process corresponds with the "overall process of enterprise architecture analysis" (Johnson et al. 2007).

1) Prepare the assessment with the responsible manager.
   a) Determine the objective of the assessment.
   b) Determine the position of the EAM function within the organization.
   c) Select one or two EA goals of the EAM function. Opt for representative goals, where the organization has been working on in recent years.
   d) Select the architect(s) and stakeholders, suitable to the selected goal(s). Include at least one relevant stakeholder per EA activity. A typical set interviewees consists of a business manager, information manager, enterprise architect, portfolio manager, solution architect, software engineer, business operations expert.
   e) Plan the assessment; the interviews and progress meetings.
2) Collect evidence.
   a) Assemble and study relevant documents (strategy, goals, architecture, roadmaps, project portfolios ...).
   b) Interview the stakeholders and architects.
   c) Describe the findings as arguments per indicator.
3) Interpret the evidence and set up a report.
   a) Process the arguments into indicator scores and scorecard scores.
   b) Interpret the scores and describe the strengths and weaknesses of the EA realization process.
   c) Set up an assessment report with the scorecard and graphics, strengths and weaknesses, and recommendations for improvement.
4) Present the outcomes of the assessment.
   a) Discuss the report and the findings with the responsible manager.
   b) Present the results to the architects and stakeholders.

7.3.1 Scoring

An EARS assessment is a retrospective examination of an EA goal's realization process, which may have spanned several years. To value the results and determine the effectiveness of the EA realization process, information is gathered by means of interviews and document study. The scores within the EARScorecard will often represent substantiated opinions; opinions of the assessor about the observed strengths and weaknesses of the EAM function, substantiated by arguments and

indicator scores. To score the results, the assessor should be able to determine and value the artifacts (depth and width) required to realize a specific goal.

Questionnaires and indicators are available to support assessors, but since the indicators have a high level of abstraction, other sources should be used as well. The EARS-indicators are derived from the TOGAF ADM input and output descriptions per phase (The Open Group 2009), so knowledge of ADM and the "Enterprise Content Metamodel" is desirable. Furthermore, many other sources are useful, like 'Essential layers, artifacts, and dependencies of EA' (Winter and Fischer 2007) and 'An engineering approach to EA design' (Aier et al. 2008).

7.4 Case Studies

To evaluate and improve EARS, the instrument was used in four organizations, located in the Netherlands; two full assessments and two brief assessments were conducted. The full assessments will be discussed below. One assessment was conducted at a governmental organization and another at a financial organization.

7.4.1 Case 1: A Large Governmental Organization

This governmental organization is practicing enterprise architecture for some years. The study focused on the EAM function responsible for a large organizational domain with more than 10,000 employees. The case study aimed to deliver the organization an assessment focused on awareness and improvement of the EA function.

Two goals were selected in dialog with the client, namely 'Provide clarity to customers more quickly', and 'Reduce the complexity of the processes'. These goals were selected because they were representative for the complete set of EA goals, and because the organization was well on its way achieving these goals. Thereafter, the responsible architect was consulted, documents relevant to the goals were collected and studied, and ten architects and stakeholders were interviewed. Finally, a report was prepared, which was discussed with, and approved by the responsible manager and some key stakeholders. The EARScorecard of the EA goal 'Provide clarity to customers more quickly' is shown in Table 7.9, and a graphical representation of the aspect totals and result totals is shown in Figure 7.5.

Table 7.9: EARScorecard of the EA goal "Provide clarity to customers more quickly"

Id	Result	Aspect	Aspect score	Scope score	Aspect total	Result total
#1	Architecture Vision	Product	9	10	9	10
		Acceptance	10		10	
#2	Architecture Design	Product	4	10	4	4
		Acceptance	4		4	
#3	Migration Plan	Product	10	10	10	10
		Acceptance	10		10	
#4	Project Result	Product	4	10	4	5
		Acceptance	6		6	
#5	Operational Result	Product	1	5	1	1
		Acceptance	1		1	
	Goal total				**59**	

The EARS scorecard shows large differences between the five results. The scores for the *Architectural Vision* are very high, because there is an approved, high-level description of what is necessary to realize the goal. Additionally, the impact of the changes is known. The high acceptance score is due to the fact that the architects work in close cooperation with the decision makers.

The score for the *Architectural Design* is relatively low. At the moment of the assessment, the architecture was focused on the baseline architecture, which sufficed to perform a proper impact analysis of the intended changes. An integrated target architecture, needed to realize all EA goals for the coming years, was mostly

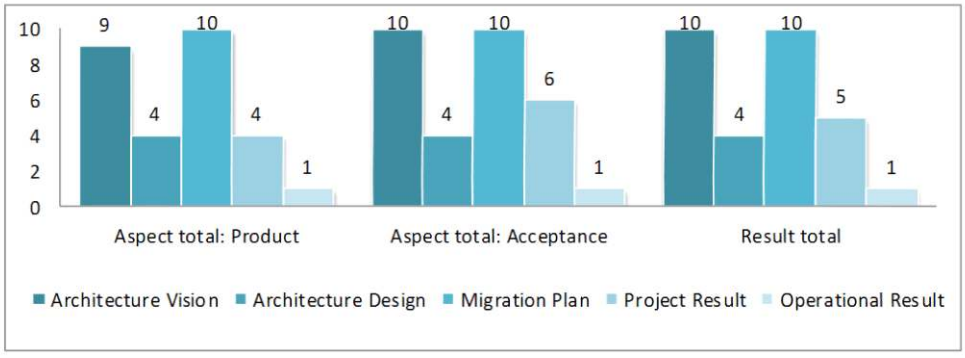

Figure 7.5: The result totals of the EA goal "Provide clarity to customers more quickly"

missing, while considerable changes were expected. Consequently, the projects related to the goal could not anticipate on the target architecture, which will result in higher than necessary transition cost in the near future.

Migration Plan scores high, because a realistic roadmap was developed and acceptance and commitment of the stakeholders was high and remained high. All four projects, needed to realize the selected goal, were included in the project portfolio, and were already under development or beyond.

The low score for *Project Result* is partly related to the missing target architecture, as discussed under Architecture Design. Consequently, the projects were not provided with architectural definitions and requirements. Positive was the collaboration with the project architects in the early stages of the project. Negative was the lack of checking of the conformance of the implementation to the architecture.

Finally, the low score for *Operational Result* is because the most important implementations were not yet operational. Positive returns were expected in the next calendar year.

7.4.2 Case 2: A Large Financial Organization

The company in this case is in transition from a decentralized organization, composed of more than ten companies and brands, to one centralized company, striving for one way of working and for operational excellence. For this assessment, the following EA goal was selected, "Implement a corporate data warehouse". A plan which included the goal was approved about three years before the assessment, and the organization had worked on its realization, since then. The EA goal was ambitious, since sub goals included not only corporate wide business intelligence (BI), including the replacement of many local BI-applications, but also the provision of integrated production data to portal and output services.

Strengths, Weaknesses and Recommendations

The assessment's evidence collection included a total of two days of document study and ten interviews, which mostly lasted 30-60 minutes. The EARS scores in Table 7.3, and Figure 7.6 and 7.7 provide an overview of the assessment's outcome, and show some strengths and weaknesses. In the assessment report, twelve strengths and sixteen weaknesses were described, based on indicator scores and arguments. Some of them will be explained below.

#1 Architecture Vision scores relatively high on the aspects product and scope, since the goal was part of a proper and quite complete architecture master plan, which provided an integrated solution to several goals, in line with the business strategy. However, a weakness of the plan was the absence of SMART sub-goals.

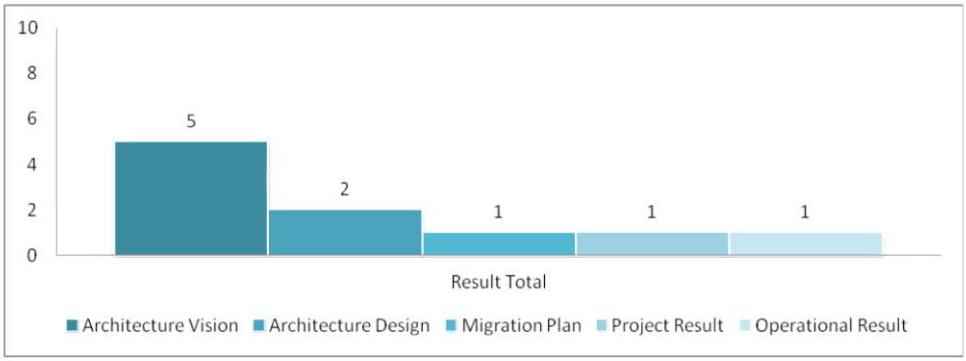

Figure 7.6: Result total per EA activity result

Acceptance scores moderate, because the vision was mainly developed on the ICT-side and was not well communicated with the business side.

#2 Architecture Design scores low, because nearly no architecture design was done before the implementation projects started. Only the baseline application architecture was described, but little more. This is probably one of the reasons why the first projects in the roadmap encountered huge problems. For instance, complexity turned out much higher than anticipated.

#3 Migration Plan scores low. The initial projects were approved by business management, but trust disappeared when these projects ran out of time and budget. Consequently, follow-up projects were not approved; reason for the remarkable decrease in scope score. At the time of the assessment, only a small part of the roadmap was realized: a few sub-goals have survived; the others have vanished with time.

#4 Project Result scores low, mainly because of the influence of the low scope

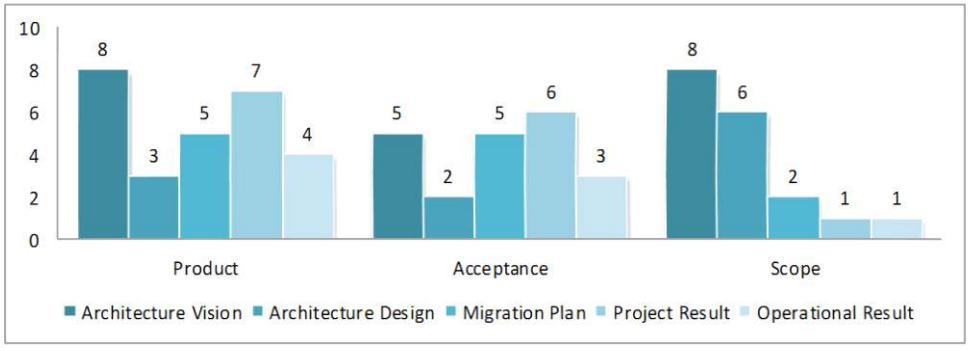

Figure 7.7: Product, Acceptance and Scope score per EA activity result

score. Product and acceptance score much higher, since the architects guided the projects actively (product aspect) and their contribution was appreciated (acceptance aspect). Because of the lack of Architecture Design, no architectural definitions and requirements were provided by the EAM function at the initiation stage of the projects, but principles and architectural artifacts were developed over the years.

#5 Operational Result scores very low. Once more because of the low scope score, but product and acceptance score low as well. A positive result is that one minor application was successfully implemented enterprise-wide, and it is well accepted by the business. However, the major BI-application was used by only a small part of the company, and the business users were dissatisfied.

Some recommendations provided in this case are the following:

- Identify explicit goals to the EAM function in collaboration with the stakeholders. Set realistic and SMART (sub) goals and work from these goals.
- Develop architectural artifacts to substantiate and verify the accuracy, impact and feasibility of the goals and solution directions. Do this for both the baseline and target situation, and use these as a base for roadmaps.
- Do not combine major goals and complex projects with a bottom-up strategy regarding the development of the EAM function and EA artifacts.

Indicator Scores

The product scores, acceptance scores and scope scores were each constituted by the weighted average of the related indicator scores. As an illustration, Table 7.10 shows how the scores of result #1 Architecture Vision are composed of the indicator scores. Per indicator, the indicator score (S), valued by the assessor on a scale of 1-10, is multiplied with the indicator's weight (W) to the indicator total (T).

Table 7.10: Aspect and indicator scores of result #1, Architecture Vision

Aspect	Id	Indicator	W	S	T
Product	1	The EA-goal is related to the business strategy and included in the vision.	0.2	10	2.0
	2	The EA-goal is SMART and, if needed, decomposed into high level stakeholder requirements.	0.2	6	1.2
	3	A high level solution direction is described and the solution direction to the goal is accurate.	0.2	7	1.4
	4	The solution direction to the goal is integrated with the solution directions of the other goals.	0.3	8	2.4
	5	A comprehensive plan exists to realize the solution direction.	0.1	7	0.7
		Product score			**7.7**
Accep-tance	1	The architecture vision is well-known by the stakeholder.	0.2	8	1.6
	2	The stakeholders understand the vision, the solution direction to the goal and its implication.	0.2	4	0.8
	3	The stakeholders agree with the solution direction to the goal and its implications.	0.3	5	1.5
	4	The stakeholders feel committed to (this part of) the vision.	0.3	4	1.2
		Acceptance score			**5.1**
Scope	1	The architecture vision covers all aspects relevant to the goal: business, data, application, and/or technology.	1.0	8	8.0
		Scope score			**8.0**

Arguments

The indicator values were substantiated by means of arguments collected during the assessment. For instance, with regard to result #1 Architecture Vision, twenty-five arguments were gathered, varying from two to seven arguments per indicator. Approximately 60% of these arguments originated from the study of architectural artifacts, while the remaining 40% did arise during the interviews. Arguments are described in case specific terms, so to ensure anonymity; Table 7.11 shows only a few, condensed examples.

Table 7.11: Arguments regarding two indicators of the product aspect of result #1

Aspect	Indicator	Contribution	Argument description
Product	1	+	The EA goal is based on the corporation's strategy and target operating model.
		+	Conformance is confirmed by several interviewees.
	2	-	The goal is not formulated explicitly, it is not SMART and no sub-goals were specified.
		+	Sub-goals can be derived from the architecture master plan.
		+	Stakeholder requirements are described in the master plan as business and ICT issues.
		-	No objectives were set for the EAM function, when the function was initiated.

7.5 Discussion

The EARS assessments, described above, proceeded without problems and provided interesting outcomes and recommendations to the organizations involved. The two full assessments show large differences in the EAM's goals and approaches, and the assessments delivered very different outcomes. However, some similarities were identified as well. Both EA functions scored low on Architecture Design, especially the target architecture. This was partly compensated, by a shared effort to draw up solution architectures within the projects. Another similarity is that both EA functions failed to check on conformance during the implementation. These findings match with research on the maturity level of 56 EAM cases (van Steenbergen et al. 2010), where the focus areas 'Development of architecture' and 'Monitoring' scored respectively low and very low on the maturity scale.

The case studies were also focused on the evaluation of the EARS approach itself. During the interviews and meetings of the case studies, additional information was gathered to gain insight in the applicability, effectiveness, and efficiency of the instrument. The EARS approach appeared to be effective, since the scorecard, indicator values and assembled arguments proved to be an adequate base to identify the strengths and weaknesses of the realization process and to provide recommendations. Moreover, the responsible managers and key stakeholders approved the outcome of the assessments, and interviewees who were asked whether the main aspects of the architecture function were covered during the interview, responded positively. As additional revenue, a responsible manager observed that the assessment stimulated the internal discussion regarding the focus, method and effectiveness of the architecture function.

During the case studies, the EARS instrument proved to be an applicable instrument. Some examples of findings during the case studies which substantiate this statement are the following:

- The EA goals were well identifiable and selecting representative goals did not cause problems.
- EA activities and the results were sufficiently distinctive and recognizable and could be found in practice.
- The result aspects "product", "acceptance" and "scope" were generally well identifiable. However for some activities, two result aspects are closely linked. Such as in # 3 Migration Plan, where "product" and "acceptance" are not well distinguishable and thus are given the same value.

7.5.1 Limitations

The outcomes of the case studies give us reasons to believe that the EARS can be applied conveniently and is effective as an assessment instrument for awareness and improvement purposes.

However, there are some limitations to our research so far. Although several assessments in different types of organizations were conducted in the Netherlands, our research findings are not inevitably valid for other companies, sectors or countries. Another limitation of our study is that we could not provide a valid conclusion regarding the efficiency of the assessment method, since no comparison with other were possible. A third limitation concerns the indicators and their relative weights. Since there was no sound scientific base available, we used professional literature and common sense. So the indicators and their relative weights will likely evolve with advancements in research and practice.

7.6 Conclusions and Future Work

In this chapter, we presented a novel instrument to assess and rate how well an EAM function is able to realize its goals: the Enterprise Architecture Realization Scorecard (EARS). The instrument decomposes the EA realization process into five EA activities, each with a distinct result. For each result, three aspects are examined and scored separately: product, acceptance, and scope. During the assessment of an organization specific EA goal, the results of the EA realization process are identified and examined by means of interviews and document study. The results are examined to assemble arguments, which are translated to numerical scores, by means of indicators. The outcome of an assessment is a report, with an EARScorecard, diagrams, strengths and weaknesses of the EA realization process, and recommendations.

We used two case studies to illustrate how the EARS instrument is used in practice. The application at a large governmental organization and a large financial organization delivered interesting outcomes; strengths and weaknesses were detected and substantiated, and recommendations were given. Since the selected goal and EAM function itself were quite different in both cases, the outcomes of the assessments and the recommendations differed significantly. The EARS approach appeared to be effective in practice. The scorecard, indicator values and assembled arguments proved to be an adequate base to identify the strengths and weaknesses of the realization process and to provide recommendations. Furthermore, the assessment stimulated the internal discussion regarding the focus, method, and effectiveness of the architecture function.

Our study contributes to the professional practice, by adding an assessment instrument that can be used to evaluate the effectiveness of an EAM function's realization process. To strengthen the link to the professional practice, the instrument is based on two well-accepted open standards: CobiT (IT Governance Institute 2007a), and TOGAF (The Open Group 2009).

The EARS instrument contributes to the research on architecture effectiveness by focusing on the EA realization process and its results. Distinctive characteristics of the EARS assessment approach are:

- Focus on EA goals that are specific to the organization;
- Focus on the EA realization process, its activities and results;
- Numerical scores in the EARScorecard to support reasoning about the strengths and weaknesses of the EA realization process;
- Aspects and indicators to aid the evaluations of results.

Future Work

Interesting topics for future work emerged during this study. Research is needed to determine whether the assessment results of one or two representative goals can be generalized to general statements about the EA function. Furthermore, comparative research on EARS and other EA assessments approaches could be interesting. It could contribute to the further development of the set of indicators. In addition, it might reveal and explain correlations between focus areas of maturity models and high scores in the EARScorecard.

Chapter 8

Conclusions

In this final chapter, we provide an overview of the most relevant findings per research question. Furthermore, we discuss the contributions, implications, and limitations of our research, and we present an agenda of possible future research. The sections of this chapter are all subdivided into subsection conform the three lines of research: Architecture Compliance Checking Support (ACCS), Layered Architecture Design Support (LADS), and Enterprise Architecture Realization Assessment (EARA). An overview of the three lines of research, the related research questions, and the artifacts per line of research is provided in Figure 8.1.

The ACCS line focused on research question RQ1 and resulted in three different types of artifacts. First, tests: the benchmark test and FreeMind test to test the accuracy of dependency detection; and the SRMA test to test the level of SRMA support. Second, an approach to enable SRMA support, grounded in the SRMACC metamodel, and third, ACC tool HUSACCT, which provides extensive SRMA support, based on the metamodel.

The LADS line of research focused on research question RQ2 and yielded two instruments to aid the design of layered software architectures: the Typology of

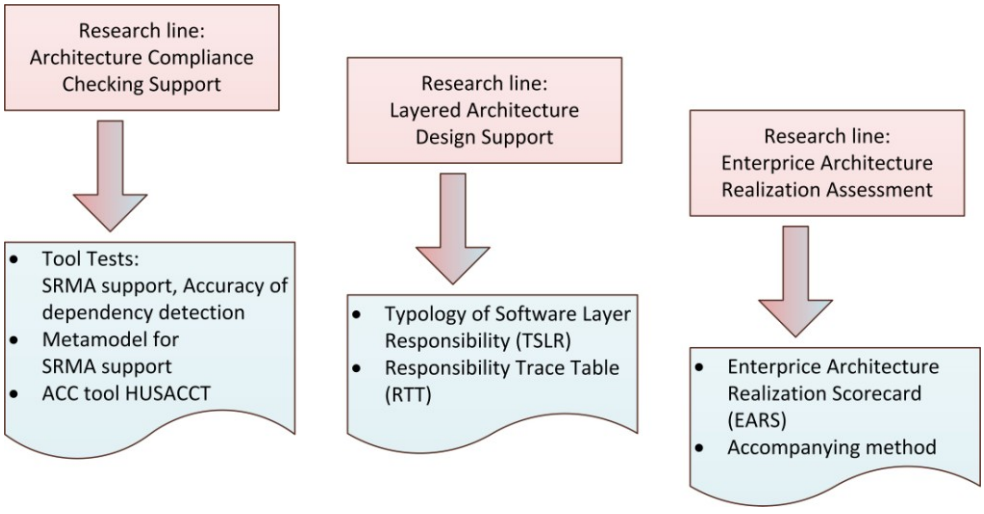

Figure 8.1: Lines of research and their artifacts

Software Layer Responsibility (TSLR), and the Responsibility Trace Table (RTT).

The EARA line of research focused on research question RQ3 and resulted in an instrument to assess and rate how well an EA management function is able to realize its goals: the Enterprise Architecture Realization Scorecard (EARS).

8.1 Answers to the Research Questions

As described in the introduction of this dissertation, the main question and driver of our research in this dissertation is:

How can IT architecture work be evaluated and improved?

To scope our research, we decomposed the main question into three major research questions. Consequently, the answers to the research questions combine to the answer to the main question. Below, the research questions are answered per research line.

8.1.1 Architecture Compliance Checking Support

Tool support enables efficient Architecture Compliance Checking (ACC). However, supporting tools are still inadequate (Clements and Shaw 2009) and the adoption of ACC-tools in practice is still limited (de Silva and Balasubramaniam 2012), which raised the following research question.

RQ1: How can architecture compliance be evaluated and improved?
To answer this question, we have studied literature on SA and ACC, identified requirements regarding ACC support, and we have studied existing static ACC tools. We identified opportunities for improvements that were not issued in other studies, and we focused our research on these opportunities, which resulted in the three sub-questions below.

RQ1.1: Do static ACC-tools provide functional support for semantically rich modular architectures?
We introduce the term *semantically rich modular architecture* (SRMA) for expressive modular architectures, composed of different types of modules, which are constrained by different types of rules; explicitly defined rules, but also rules inherent to the module types.

To answer RQ1.1, we identified requirements to the support of SRMAs and classified module types and rule types relevant for static ACC. Furthermore, we prepared a test, and we tested eight tools on their support of SRMAs.

We focused our test on the support of: a) common types of modules and their semantics; b) common types of rules; and c) inconsistency prevention within the

defined architecture. To determine the common module types, common rules types and inconsistency checks relevant to our research, we studied academic and professional literature, as well as software architecture documents from professional practice and ACC-tool documentation. We classified the following common module types: non-semantic clusters (logical and physical), layers, components, facades, and external systems. Furthermore, we identified and classified twelve types of common rules, five property rule types and seven relation rule types. Three rule types are specific for certain types of modules: the facade convention is specific for components with a facade, and the back call ban and skip call ban are specific for layers.

Our tests regarding the support of common module types show that five of the eight tools supported non-semantic clusters only. The three other tools distinguished also one or more semantically rich module types from our classification. SAVE supported the graphical definition of five types of modules, but did not support their semantics. Sonargraph Architect supported the semantics of a facade actively, while Structure101 supported the semantics of layers actively. However, no tool provided combined support of layers, components, and facades. Furthermore, no tool provided configuration options for the semantic support of a module type or module.

Our tests regarding the support of common rule types show that per tool only a few rule types were explicitly supported, mostly “is not allowed to use” and “is allowed to use”. More complex relation rules were by no tool explicitly supported. Consequently, complex relation rules at logical level required workarounds at tool-level, which often resulted in two or more unrelated rules; a threat to the maintainability and traceability of the set of rules. Furthermore, only two of the five property types were supported, and only partially, not explicitly.

Our tests regarding the support of inconsistency prevention show that only two tools, ConQAT and Lattix, scored high on the prevention of inconsistencies in the module and rule model. A relevant observation, since inconsistent models may result in an unreliable outcome of the compliance check.

Finally, as answer to RQ1.1, we concluded that only limited support was available for semantically rich modular architectures (SRMA), and that the scope of ACC tools should be widened from dependency checking to software architecture compliance checking, including SRMAs. We also concluded that solutions were needed to minimize the difference between logical rules, as perceived by the architect, and the technical implementation in the tool.

RQ1.2: How can SRMA support be provided in the context of static ACC?
As answer to this research question, we have designed a metamodel for extensive support of SRMAs in the context of static ACC. The metamodel, we labeled it

"SRMACC metamodel", identifies, describes and relates the core concepts needed to address the following objectives regarding SRMA support. The first is to provide basic SRMA support, which includes the provision of sets of common module and rule types and the functionality to check rules of these types. The second is to provide extensive SRMA support, which adds support of the semantics of the common module and rule types. The third is to enable configuration of the provided support. The metamodel provides the fundamental concepts for: a) the definition of the planned modular architecture, including common module and rule types; b) extensive semantic support of these types; c) module mapping; and d) conformance checking.

Moreover, we have developed HUSACCT (Hogeschool Utrecht Software Architecture Compliance Checking Tool), an ACC-tool that provides extensive and configurable SRMA support for all common module types in our classification and eleven (of the twelve) common rule types. HUSACCT aids the analysis of implemented architectures, definition of intended architectures, and execution of conformance checks. Browsers, diagrams, and reports are available to study the decomposition style, uses style, generalization style, and layered style of intended architectures and implemented architectures. HUSACCT is free-to-use and open source. It is based on the SRMACC metamodel, has been developed in Java, and analyzes Java and C# source code. The executable, user manual and introduction video may be retrieved via http://husacct.github.io/HUSACCT/.

To validate our approach, we have performed ACCs with our tool on professional systems and open source systems, and we have used the tool in bachelor and master courses on software architecture. The application cases have shown that SRMAs are used in practice, and that SRMA support can be provided based on the SRMACC metamodel.

RQ1.3: How accurate do static ACC-tools report dependencies and violations against dependency rules?

To answer this research question, we have investigated to which extent static ACC-tools report violation messages and dependency messages accurately. We classified dependency types, prepared a benchmark test, and tested ten tools based on this benchmark test. In addition, we have tested these tools based on the program code of open source system FreeMind, which we used to test the ability of the tools to report all depended-upon classes, all dependency-causing constructs, and all information needed to locate dependency-causing constructs in the source code.

Based on the test results we found the following answers to research question RQ1.3. The test results show large differences between the tools, but the main conclusion is that all tools could improve the accuracy of the reported dependencies and violations. The ten tools detected on the average 77 percent of

the dependency types in the benchmark test and 72 percent of the 109 manually identified dependencies in a class of FreeMind. However, we detected no false positives in the benchmark test, and no inconsistencies between dependency reports and violation reports.

All tools report violations to dependency rules at class level. However, at this level of abstraction, one message may represent several actual dependencies. Six of the ten tools also provide dependency details in reports or IDE plug-ins, but not always precisely enough to localize dependencies discretely within a method, or even better within a line of code.

Based on the test results, we identified ten hard-to-detect types of dependencies, and four challenges regarding dependency detection. To substantiate the relevance of our findings, we performed an analysis of the number of dependencies per dependency type in five open source systems. The analysis results revealed that a large fraction of the dependencies in these systems is potentially hard-to-detect, while considerable fractions are inheritance related and inner class related.

Our tests have shown that inheritance structures and inner classes hamper the accuracy of violation reporting in many cases. A dependency caused by usage of inherited methods or variables is often not reported, and if reported, than mostly as dependency on the accessed subclass only and not on the super class that implements the method or variable. In addition, a dependency caused indirectly by inheritance relations is not reported at all. Furthermore, usage of an inner class is often not reported at all, and if reported, it is frequently reported as a dependency on the outer class instead of the inner class, which diminishes the traceability in the source code.

8.1.2 Layered Architecture Design Support

The Layers pattern is one of the most common patterns used in software architecture (Harrison and Avgeriou 2008), but layered designs are often poorly defined and many violate the principles for which layers are designed (Clements et al. 2010). In this dissertation, we focused on the following research question.

RQ2: How can the quality of layered designs with respect to the assignment of responsibilities be evaluated and improved?

The starting point of our investigation was the observation that to answer the design question “Which types of responsibility are assigned to each layer?”, a uniform classification for the naming and characterization of types of responsibilities in software layers could be useful. This perception resulted in the following sub-questions, which were leading in our study: 1) What types of responsibilities are distinguished in layered architectures; 2) How can these types

of responsibility be named and defined unambiguously; and 3) How can a typology of responsibilities be applied in practice?

To answer these research questions, we studied leading literature about software layers, to get an overview of common types of responsibilities and the names given to them, and we designed two instruments. Based on this literature, we constructed the Typology of Software Layer Responsibility (TSLR) and the complementary Responsibility Trace Table (RTT). These instruments, together with some illustrations of their practical use, provide answers to the research questions.

The TSLR provides an overview of the distinct types of responsibility commonly found in the software of business information systems. The TSLR responsibilities are distilled from leading literature on layers in the domain of software architecture. The TSLR separates and groups the responsibilities, gives them unambiguous names, specifies them and exemplifies them. At the level of infrastructural responsibilities a connection is established to the TOGAF Technical Reference Model (The Open Group, 2009), which classifies a huge number of infrastructural services.

The Responsibility Trace Table (RTT) shows the assignment of the TSLR responsibilities to the different software layers. The RTT is an instrument to complement a system's graphical representation of the layered design with a specification of the responsibilities of the layers. In addition, the RTT may be used to assess and enhance the quality of a layered design, since it shows omissions and redundancies in the assignment of the responsibilities.

To illustrate the application of the instruments three cases were presented: a design case, a review case, and a complex case of al large governmental software system. These cases were also used to evaluate the completeness, accuracy, and applicability of the instruments. Furthermore, experts in the domain of software architecture conducted a review, and the instruments were used in training courses and projects for bachelor students. Based on the results of these application cases, we can conclude that the TSLR and RTT help to assess, discuss, and improve the quality of layered designs with respect to the assignment of responsibilities

8.1.3 Enterprise Architecture Realization Assessment

Over the last decades, Enterprise Architecture Management (EAM) is introduced in many organizations, but the introduction and elaboration often do not proceed without problems, and most practices are still in the early stages of maturity (Bucher et al. 2006, van Steenbergen et al. 2010). Moreover, the performance of the EA management function typically is not measured (Winter et al. 2010). In the course of a larger study on the value of Enterprise Architecture (EA), we focused on the research question below.

RQ3: How can the achievement of an EAM function be measured with respect to the realization of its goals?

As answer to this research question, we have presented a novel instrument to assess and rate how well an EA management function is able to realize its goals, the Enterprise Architecture Realization Scorecard (EARS). During the assessment of an EA goal of an EAM function, five types of results, delivered during the EA realization process, are analyzed and discussed in interviews with relevant stakeholders. Arguments are assembled and, by means of indicators, translated to scores. For each result, three aspects are scored: product, acceptance and scope. The scores are recorded at a scorecard and subsequently, totals at result level and goal level can be calculated. Finally, an assessment report is prepared, with a scorecard, strengths and weaknesses of the EA realization process (based on the scores in the scorecard, indicator scores and arguments), and recommendations.

Furthermore, we have presented two case studies to illustrate how the EARS instrument is used in practice. The application at a large governmental organization and a large financial organization delivered interesting outcomes: strengths and weaknesses were detected and substantiated and recommendations were given. Since the selected goal and EA management function itself were quite different from the first case, the outcome of the assessment and the recommendations differed significantly. The EARS approach appeared to be effective in these cases. The scorecard, indicator values and assembled arguments proved to be an adequate base to identify the strengths and weaknesses of the realization process and to provide recommendations.

8.2 Contributions and Implications

For each of the three lines of research in this dissertation, the contributions and implications for research, practice, and education are described below.

8.2.1 Architecture Compliance Checking Support

Contributions and Implications for Research

SRMA Support

We regard the recognition of *the support of semantically rich modular architectures (SRMA) as an issue in the context of ACC* as a relevant contribution. With the acceptance of our first paper on SRMA support on the International Conference on Software Maintenance (ICSM) 2013, one of the reviewers commented, "this is a topic that is very important", while another commented that the paper, "provides sufficient contribution for future research directions for approaches and tooling on architecture compliance checking". In line with the last remark, a study with significant references to our first paper on SRMA support (Caracciolo et al. 2015) has been published at the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2015; an interesting paper on a unified approach on ACC.

As next contribution in the SRMA-line, we have developed an approach to provide extensive support of SRMA, grounded in the SRMACC metamodel. The metamodel describes and relates not only the core concepts needed for SRMA support, but also those needed for a full cycle of ACC. The metamodel may be helpful to enhance existing tools or to develop new approaches.

The last contribution in the SRMA-line is HUSACCT (HU Software Architecture Compliance Checking Tool), a tool for extensive and flexible SRMA support. In addition, as open source tool with high qualities regarding dependency detection, SRMA support, and visualization, HUSACCT offers numerous opportunities for ensuing research and new research in the field of Architecture Reconstruction (AR) and ACC.

Accuracy of ACC

In time, the first contribution in the research line ACC support was the introduction of *accuracy of static analysis* as an issue in the context of ACC research. Before our research was accepted on the International Conference on Program Comprehension (ICPC) 2013, accuracy was no item in research papers on ACC methods and tools, as explained in chapter 4. Our work on ACC accuracy has shown that tools with a low accuracy are not capable to report all depended-upon classes and by far not all dependencies within a class's program code.

The first effects are visible. A paper (Olsson et al. 2014) has been published at the conference on Quality of Software Architectures (QoSA) in which a static ACC method is introduced, but now with attention to accuracy, with measures of true positives, false positives, precision and recall, and with a reference to our work.

Contributions and Implications for Practice

Our work was aimed at a practical goal: Contribute to the advancement of static ACC support. As a consequence, the adoption of ACC in practice might increase in the future. On short term, we perceive the results below as contributions.

First, we have developed test to measure SRMA support and accuracy of dependency detection, and we have published test results, which show point for improvement. We expect that this will have impact. Furthermore, the tests have been requested by several tool builders, who can use the tests for improvements.

Second, we have developed HUSACCT (with instruction video and user manual), which may be used, free of charge, by researchers, practitioners, and students alike. Furthermore, it is our hope that novel functionality in our tool will be taken over in other tools, or will inspire other tool builders to evolve their tools.

Third, we have performed software architecture compliance checking at software development departments of number of large organizations. ACC was new in these situations, and the contribution was, apart from the measurement results, an increase of architectural awareness and a better understanding of the relationship between architecture design and code.

Contributions and Implications for Education

Education in software architecture with subjects as modular architectures, module styles, architecture reconstruction, and architecture compliance checking might benefit from the usage of static analysis tools. HUSACCT is intended to be used for education in the fields of modular software architectures. Course material in the form of presentations, classical exercises, and individual or group assignments are developed for second and third year computer science students. Currently, HUSACCT and the additional course material is used in the education of software architecture at the HU University of Applied Sciences and Utrecht University, both in Utrecht, and HAN University of Applied Sciences in Arnhem, all in The Netherlands. Furthermore, some other universities have shown interest as well.

Finally, students of the specialization “Advanced Software Engineering” at the HU University of Applied Sciences have, from 2011-2013, contributed in various ways to our research and the development of HUSACCT. Conversely, these students are educated much more in-depth on the subject of software architecture, as described in our paper on the educational pattern *Multi-Level Assignment* (Köppe and Pruijt 2014).

8.2.2 Layered Architecture Design Support

Contributions and Implications for Research

We have proposed a novel instrument to support software architects in their task to design layered architectures of high quality: the Typology of Software Layer Responsibility (TSLR). The TSLR is a first typology of software responsibilities in modular architectures.

In addition, we have proposed an approach to apply the TSLR, including a supporting instrument, the Responsibility Trace Table (RTT). The approach is supportive during the design of new layered designs and the analysis of existing layered designs.

Furthermore, we have presented three cases, which we have used to illustrate and evaluate the practical use of the TSLR and RTT.

Contributions and Implications for Practice

Our research has provided a typology of responsibilities and an approach to apply the typology, which are supportive in the design and documentation of high quality layered designs. The TSLR and RTT may be used when a layered design is drawn up and when an existing layered design is analyzed or reviewed. Furthermore, the instruments are useful in training courses to discuss and exercise the different possibilities to divide responsibilities over the layers and their impact on the quality characteristics of the software system.

Currently, the practice of software architecture lacks a standard terminology with respect to responsibility assignment to layers in layered designs or, more in general, to modules in modular architectures. The TSLR may be a first step towards such a standard terminology.

Contributions and Implications for Education

The use of the TSLR and RTT may be helpful in the training of students, software engineers, and architects on the subject of layered software architectures. Drawing up or implementing a layered design requires knowledge of the different types of responsibilities.

At the HU University of Applied Sciences, we use the TSLR to let the students acquire this knowledge. Furthermore, we discuss the suitability of several layered designs to meet specified quality requirements, and we discuss proposals for layers in student projects. The visual character of the TSLR's classification schema, and the overview provided by an RTT, support the explanation and discussion of different design alternatives regarding the assignment of responsibilities to layers.

8.2.3 Enterprise Architecture Realization Assessment

Contributions and Implications for Research

Our work contributes to the research on architecture effectiveness by focusing on the EA realization process and its results, which resulted in a novel instrument, the Enterprise Architecture Realization Scorecard (EARS).

A number of instruments with similar objectives is developed and proposed, like EA balanced scorecard (Schelp and Stutz 2007), EA maturity models (e.g., Luftman 2000, van Steenbergen et al. 2010), and EA analysis approaches (e.g., Johnson et al. 2007). The main difference between the balanced scorecard approach and the EARS approach is that the balanced scorecard approach is concerned only with the outcome (added value) of EA management, while the EARS approach is also concerned with how the outcome is reached. The main difference with the maturity approach is that this approach aims to measure the effectiveness of the EA realization process indirectly, by assuming that when a certain maturity level is reached for each key area, the EA function will operate effectively. In contrast, the EARS approach aims to measure the effectiveness of each step in the EA realization process directly, by assessing the results. The main differences with the EA analysis approaches is that these are aimed at one concern, while in the EARS approach the concern of the assessment will vary per EA goal. Furthermore, the EARS approach is not only focused on EA artifacts, like the EA analysis approaches, but on all activities and results of the EA realization process, for example, including the acceptance of the architectural decisions, and the outcome of architecture conformance checks.

We have presented the EARS instrument, its metamodel and metrics, as well as the accompanying method. In summary, the characteristics of the EARS assessment approach are the following:

1. Focus on EA goals which are specific to the organization;
2. Focus on the EA realization process, its activities and results;
3. Numerical scores in the EARScorecard to support reasoning about the strengths and weaknesses of the EA realization process;
4. Aspects and indicators to aid the evaluations of results.

Finally, we have presented two assessment cases to illustrate the use of the instrument, one assessment at a governmental organization and another at a financial organization.

Contributions and Implications for Practice

Our study contributes to the professional practice, by adding an assessment instrument that can be used to evaluate the effectiveness of an EAM function's realization process. To strengthen the link to the professional practice, the

instrument is based on two well-accepted open standards: CobiT (IT Governance Institute 2007a) and TOGAF (The Open Group 2009).

In the case studies, the EARS approach appeared to be effective, since the scorecard, indicator values and assembled arguments proved an adequate base to identify the strengths and weaknesses of the realization process and to provide recommendations. Moreover, the responsible managers and key stakeholders approved the outcome of the assessments. As additional revenue, a responsible manager observed that the assessment stimulated the internal discussion regarding the focus, method and effectiveness of the architecture function.

8.3 Reflections, Limitations, and Future Work

In this section, we reflect for each of the three lines of research on the limitations of the work done and possibilities for future research.

8.3.1 Architecture Compliance Checking Support

In our opinion, architecting tools with functionality for architecture analysis, architecture reconstruction and architecture compliance checking are crucial to improve the effectiveness of software architecture, and to extend the role of software architecture in practice and education. With our presented work, we have aimed to contribute to the advancement of the support of ACC. We have identified requirements, tested existing tools on these requirements, and we have designed artifacts to answer the requirements entirely. However, much more work is needed to acquire a high adoption of ACC and ACC-tools in practice. Our work has focused on the support of semantically rich modular architecture and on accurate dependency detection, but we have encountered many other interesting initiatives in the field of ACC. Although scientists learn from each other, these initiatives are most often isolated from the others, just like our work. To contribute more to the advancement of SA, cooperation is needed. A joint research effort could result in an integrated solution, which integrates the features from the different studies and their related prototypes. Such a solution could be valuable to both, practice and research. Some examples of interesting studies, which focus on various aspects that should be included in an integrated solution, are the following:

- Continuous quality assurance (Deissenboeck et al. 2005);
- Declarative, tool-agnostic rule language (Caracciolo et al. 2015);
- Dependency constraint language (Terra and Valente 2009);
- Flexible ACC support (Deissenboeck et al. 2010);
- Just in time ACC support (Ali et al. 2012);
- Rule-based architecture conformance checking (Herold et al. 2013);
- UML-based architecture to code tracing (Adersberger and Philippsen 2011);

- Visualization (Knodel et al. 2006).

Before we proceed to future work, we will consider the most relevant limitations of our work (the limitations per study are discussed more extensively in the chapters 2-5). First, we have conducted several tests on ACC-tools and reported our findings. These tests were relevant, since they provided an impression of the state-of the art. The test revealed several weaknesses, but we believe it is important to realize that these findings may not be generalized to all other ACC-tools. Moreover, the findings may not be generalized to newer versions of the tested tools, since these tools may improve their performance on the tested characteristics.

Second, HUSACCT is an important artifact of our research, but its limitations should be considered as well, just as with other ACC-tools. We have learned from our studies that it is wise to test a tool's performance for a required task, and not to rely on it without doubt. This is also recommended in case of usage of HUSACCT. As might be expected, we have taken measures for quality control. With respect to the quality of SRMA support, we have automated the SRMA test in HUSACCT's development environment. Furthermore, the tool is used frequently for practical cases. To ensure the accuracy of dependency detection, we have automated the benchmark tests. Furthermore, we have extended the test set with many test cases, for instance, to cover all variations of the dependency types in the Freemind test as well. Even so, we cannot guarantee accuracy in all possible cases, and especially not for extensions in new versions of programming languages. With special regard to the latter: HUSACCT is not only delimited by internal design decisions, but also by the capabilities of the depended upon open source frameworks.

For future work that builds on our work to this point, we perceive numerous opportunities for interesting studies, of which the most relevant are described below. First, we intend to perform case study research with HUSACCT, aimed on in-depth analysis of the architectural rules and on violations to these rules in open source and closed source systems. Second, we intend to perform research on the set of module types, rule types, and patterns that require support according to software architects in practice. Third, we intend to perform research on the visualization of different types of architectural rules in intended architecture models, and visualization of violations against rules of these types. Fourth, we intend to perform research on the automatic recognition of module types and architectural patterns, on visualization of related results, and on automatic architecture refactoring advice. Finally, in the last years of this study we spent a lot of effort to raise the tool from an academic prototype to a tool fit for practical use. Only recently, we have started to promote the usage of HUSACCT in academic courses at other universities, and the usage of HUSACCT by professional organizations, as a means to introduce ACC in their software development process. We intend to continue this way.

8.3.2 Layered Architecture Design Support

The design of a modular architecture focuses on the organization of a software system into modules, where a module represents a responsibility assignment (Parnas 1972). Although the Layers pattern is commonly used in software architecture (Harrison and Avgeriou 2008) to modularize a system, very few scientific papers actually discuss the notion and concepts of layers (Savolainen and Myllarniemi 2009). Our research has focused on the vital step in the design of a layered design, the assignment of responsibilities. To support the design, but also to support the analysis of existing layered designs and the implementation of such a design, we have developed two instruments, the Typology of Software Layer Responsibility (TSLR) and its complementary Responsibility Trace Table (RTT). Furthermore, we have demonstrated the applicability of the TSLR and RTT.

Limitations do apply to our work. The first limitation is that our research focused on responsibilities of the software of business information systems. Therefore, other types of systems, like embedded systems and games, might contain responsibilities not included in the TSLR.

Another limitation has to do with the completeness of the typology. Despite our extensive literature study and validation activities, we cannot ensure that all types of responsibilities, common in business information systems, are represented in the TSLR. However, future additions and evolution are taken into account; the meta-model of the typology enables extensions in width and depth.

A third limitation is that the typology could be viewed and used as a layered model. However, the typology is not intended to be a template for layered designs, with layers exactly matching the main types of responsibility of the typology. Layered designs in practice should be designed to meet the specific requirements of the system. The number and names of the required layers may vary, the responsibilities per layer may vary, and a layer may contain sub-responsibilities from different main types of responsibility within the TSLR.

Future work is needed to address the limitations and enlarge the scope of the typology. At first, future research may be aimed on the applicability and scope of the TSLR and RTT. It will be interesting to study the effectiveness of the TSLR and RTT when practitioners and students apply these instruments. Next, to enlarge the field of application of the TSLR, literature and case studies are needed on the responsibilities of other types of software systems (other than business information systems). Finally, it will be interesting to study the applicability of the instruments in the context of other software architecture patterns.

8.3.3 Enterprise Architecture Realization Assessment

"The discipline of enterprise architecture has already been subject to impressive development, but there is still some way to go" (Simon et al. 2013). Our research in the domain of enterprise architecture has focused on the effectiveness of the EA realization process of EA Management (EAM) functions, and it has resulted in a novel instrument, the Enterprise Architecture Realization Scorecard (EARS). The instrument has been used in practice, and we have presented two assessment cases to illustrate the use of the instrument. The outcomes of the case studies give us reasons to believe that the EARS can be applied conveniently and is quite effective as an assessment instrument with awareness and improvement purposes.

However, there are some limitations to our research so far. Although three assessments in different types of organizations were conducted in the Netherlands, our research findings are not inevitably valid for other companies, sectors, or countries. Furthermore, our study could not provide a valid conclusion regarding the efficiency of the assessment method, since it did not include a comparison with other assessment approaches. The EARS approach appeared to be quite efficient to the research team, because after five to six interviews, the image was sufficiently sharp and the results could be rated. Subsequent interviews did add little new knowledge to the assessment, but were useful to confirm findings.

Interesting topics for future work emerged during this study. More case study research could provide more insight, in both the assessment instrument and the performance of EAM functions. Furthermore, it is interesting to compare EARS and other EA assessments approaches, for instance maturity model based approaches, in theory and practice. It could contribute to the further development of the set of indicators used in EA research. In addition, it might reveal and explain correlations between high scores in the EARScorecard and high scores on focus areas of maturity models, which may be used in both approaches to substantiate the approach, as well as the advice provided in practice.

References

Adersberger, J., and Philippsen, M. (2011). ReflexML: UML-based architecture-to-code traceability and consistency checking. In *5th European Conference on Software Architecture* (pp. 344–359).

Aier, S., Kurpjuweit, S., Schmitz, O., Schulz, J., Thomas, A., and Winter, R. (2008). An Engineering Approach to Enterprise Architecture Design and its Application at a Financial Service Provider. In P. Looos, M. Nüttgens, K. Turowski, & D. Werth (Eds.), *Proceedings of Modellierung betrieblicher Informationssysteme* (pp. 115 – 130). Gesellschaft für Informatik.

Ali, N., Rosik, J., and Buckley, J. (2012). Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study. *8th International Conference on Quality of Software Architectures*, 23–32.

Allen, P., and Frost, S. (1997). *Component Based Development for Enterprise Systems: Applying the Select Approach*. Cambridge University Press.

Arlt, S., Podelski, A., Bertolini, C., Schäf, M., Banerjee, I., and Memon, A. M. (2012). Lightweight static analysis for GUI testing. In *Software Reliability Engineering, ISSRE* (pp. 301–310). IEEE.

Avgeriou, P., and Zdun, U. (2005). Architectural Patterns Revisited — A Pattern Language. In *10th European Conf. Pattern Languages of Programs (EuroPLoP)* (pp. 431–470).

Barowski, L., and Cross, J. (2002). Extraction and use of class dependency information for Java. In *Reverse Engineering, 2002. Ninth Working Conference on* (pp. 309–315). IEEE.

Bass, L., Clements, P., and Kazman, R. (2012). *Software Architecture in Practice* (Third Edit). Addison-Wesley.

Binkley, D. (2007). Source Code Analysis: A Road Map. In *Future of Software Engineering (FOSE '07)* (pp. 104–119). IEEE.

Bischofberger, W. R., Kühl, J., and Löffler, S. (2004). Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking. In F. Oquendo, B. Warboys, & R. Morrison (Eds.), *European Workshop on Software Architecture* (Vol. 3047, pp. 1–9). Springer.

Bucher, T., Fischer, R., Kurpjuweit, S., and Winter, R. (2006). Enterprise Architecture Analysis and Application – An Exploratory Study. In *Proceedings of Trends in Enterprise Architecture Research*. Hong Kong.

Buckl, S., Matthes, F., and Schweda, C. M. (2009). Classifying Enterprise Architecture Analysis Approaches. In *Enterprise Interoperability* (pp. 66–79). Springer.

Buckley, J., Mooney, S., Rosik, J., and Ali, N. (2013). JITTAC: A Just-in-Time tool for architectural consistency. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 1291–1294). Ieee.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1*. John Wiley & Sons.

Callo Arias, T. B., Spek, P., and Avgeriou, P. (2011). A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, *16*(5), 544–586.

Canfora, G., Di Penta, M., and Cerulo, L. (2011). Achievements and challenges in software reverse engineering. *Communications of the ACM*, *54*(4), 142–151.

Caracciolo, A., Lungu, M. F., and Nierstrasz, O. (2015). A Unified Approach to Architecture Conformance Checking. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)* (p. To appear). ACM Press.

Chen, D., Doumeingts, G., and Vernadat, F. (2008). Architectures for enterprise integration and interoperability: Past, present and future. *Computers in Industry*, *59*(7), 647–659.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Merson, P., Ivers, J., … Nord, R. (2010). *Documenting Software Architectures: Views and Beyond*. Pearson Education.

Clements, P., and Nord, R. (2000). Documenting a Layered Software Architecture. In *Fourth International Software Architecture Workshop Limerick* (pp. 121–124).

Clements, P., and Shaw, M. (2009). "The golden age of software architecture" revisited. *IEEE Software*, *26*(4), 70–72.

Cockburn, A. (1997). Structuring Use Cases with Goals. *Journal of Object-Oriented Programming*, (Sep.-Oct. (part I) and Nov.-Dec. (part II)).

De Silva, L., and Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, *85*(1), 132–151.

Deissenboeck, F., Heinemann, L., Hummel, B., and Juergens, E. (2010). Flexible architecture conformance assessment with ConQAT. In *2010 ACMIEEE 32nd International Conference on Software Engineering* (Vol. 2, pp. 247–250). IEEE.

Deissenboeck, F., Pizka, M., and Seifert, T. (2005). Tool Support for Continuous Quality Assessment. *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, 127–136.

Department of Defense. (2009). The Department of Defense Architecture Framework (DoDAF), version 2.

Dijkstra, E. W. (1968). The structure of the "THE"-multiprogramming system. *Communications of the ACM*, *11*(5), 341–346.

Ducasse, S., and Pollet, D. (2009). Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, *35*(4), 573–591.

Dyer, R., Rajan, H., Nguyen, H. A., and Nguyen, T. N. (2013). *A large-scale empirical study of Java language feature usage*.

Emanuel, A. W. R., and Surjawan, D. J. (2012). Revised Modularity Index to Measure Modularity of OSS Projects with Case Study of Freemind. *International Journal of Computer Applications*, *59*(12), 28.

Erl, T. (2008). *SOA Design Patterns*. Prentice Hall.

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

FAC-IFIP Task Force. (1999). *GERAM: Generalized Enterprise Reference Architecture and Methodology*.

Feilkas, M., Ratiu, D., and Jurgens, E. (2009). The loss of architectural knowledge during system evolution: An industrial case study. In *2009 IEEE 17th International Conference on Program Comprehension* (pp. 188–197). IEEE.

Foorthuis, R., Steenbergen, M. Van, Mushkudiani, N., Bruls, W., Brinkkemper, S., and Bos, R. (2010). On course, but not there yet: Enterprise Architecture conformance and benefits in systems development. In *Proceedings of the International Conference on Information Systems*.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, M., and Stafford, R. (2003). *Patterns of enterprise application architecture*. Addison-Wesley, Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissedes, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

Gleirscher, M., and Golubitskiy, D. (2012). On the Benefit of Automated Static Analysis for Small and Medium-Sized Software Enterprises. *Software Quality. Process Automation In Software Development*.

Gleirscher, M., Golubitskiy, D., Irlbeck, M., and Wagner, S. (2013). Introduction of static quality analysis in small- and medium-sized software enterprises: experiences from technology transfer. *Software Quality Journal*.

Gorton, I. (2006). *Essential Software Architecture*. Springer Berlin Heidelberg.

Haitzer, T., and Zdun, U. (2012). DSL-based Support for Semi-Automated Architectural Component Model Abstraction Throughout the Software Lifecycle Categories and Subject Descriptors. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures* (pp. 61–70).

Harrison, N. B., and Avgeriou, P. (2008). Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation. In *6th Working IEEE/IFIP Conference on Software Architecture* (pp. 147–156). IEEE Comput. Soc.

Herold, S., Mair, M., Rausch, A., and Schindler, I. (2013). Checking Conformance with Reference Architectures: A Case Study. In *Enterprise Distributed Object Computing Conference (EDOC)* (pp. 71–80).

Hevner, A., March, S., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–105.

Huynh, S., Cai, Y., Song, Y., and Sullivan, K. (2008). Automatic modularity conformance checking. In *Proceedings of the 13th international conference on Software engineering - ICSE '08* (pp. 411–420). New York, New York, USA: ACM Press.

ISO. (2007). *Iso/iec 42010:2007 systems and software engineering {recommended practice for architectural description of software-intensive systems}*.

ISO/IEC. (2011). *25010 Systems and software engineering - System and software product Quality Requirements and Evaluation (SQuaRE) - System and software quality models*.

IT Governance Institute. (2007a). *CobiT 4.1*.

IT Governance Institute. (2007b). *CobiT 4.1 Excerpt, Executive Summary*.

Johnson, P., Johansson, E., Sommestad, T., and Ullberg, J. (2007). A Tool for Enterprise Architecture Analysis. *Proceedings of the International Enterprise Distributed Object Computing Conference*, 142–142.

Kazman, R., Bass, L., and Klein, M. (2006). The essential components of software architecture design and analysis. *Journal of Systems and Software*, *79*(8), 1207–1216.

Knodel, J., Muthig, D., Naab, M., and Zeckzer, D. (2006). Towards empirically validated software architecture visualization. In *Proceedings of the 2006 ACM symposium on Software visualization - SoftVis '06* (p. 187). New York, New York, USA: ACM Press.

Knodel, J., and Popescu, D. (2007). A Comparison of Static Architecture Compliance Checking Approaches. In *Working IEEE/IFIP Conference on Software Architecture* (pp. 12–21). IEEE.

Ko, A. J., Myers, B. A., Member, S., Coblenz, M. J., and Aung, H. H. (2006). An Exploratory Study of How Developers Seek , Relate , and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, *32*(12), 971–987.

Köppe, C., and Pruijt, L. (2014). Improving Students ' Learning in Software Engineering Education through Multi-Level Assignments. In *Proceedings of Fourth Computer Science Education Research Conference, CSERC'14*. Berlin, Germany.

Koschke, R. (2010). Incremental Reflexion Analysis. *14th European Conference on Software Maintenance and Reengineering*, 1–10.

Koschke, R., Frenzel, P., Breu, A. P. J., and Angstmann, K. (2009). Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, *17*(4), 331–366.

Koschke, R., and Simon, D. (2003). Hierarchical reflexion models. In *10th Working Conference on Reverse Engineering* (pp. 36–45).

Krafzig, D., Banke, K., and Slama, D. (2005). *Service-Oriented Architecture Best Practices*. Prentice-Hall.

Kruchten, P. B. (1995). The 4+1 View Model of architecture. *IEEE Software*, *12*(6), 42–50.

Kruchten, P., Lago, P., and Vliet, H. Van. (2006a). Building up and reasoning about architectural knowledge. *Quality of Software Architectures*, 43–58.

Kruchten, P., Obbink, H., and Stafford, J. A. (2006b). The Past, Present, and Future of Software Architecture. *IEEE Software*, (March / April 2006).

Lange, M., Mendling, J., and Recker, J. C. (2012). Measuring the realization of benefits from enterprise architecture management. *Journal of Enterprise Architecture*, *8*(2), 30–44.

Lankhorst, M. et al. (2009). *Enterprise Architecture at Work: Modeling, Communication, and Analysis*. Springer, Berlin.

Larman, C. (2005). *Applying UML And Patterns*. Prentice Hall PTR.

Löhe, J., and Legner, C. (2014). Overcoming implementation challenges in enterprise architecture management : a design theory. *Information Systems and E-Business Management*, *12*, 101–137.

Luftman, J. (2000). Assessing business-IT alignment maturity. *Communications of AIS*, *4*(Article 14).

Morganwalp, J., and Sage, A. (2004). Enterprise architecture measures of effectiveness. *International Journal of Technology, Policy and Management*, *4*(1), 81–94.

MSDN. (2009). *Microsoft Application Architecture Guide, 2nd ed.* Microsoft Corporation.

Murphy, G. C., Notkin, D., and Sullivan, K. (1995). Software reflexion models. *ACM SIGSOFT Software Engineering Notes*, *20*(4), 18–28.

Obitz, T., and Babu, M. K. (2009). *Enterprise Architecture Expands its Role in Strategic Business Transformation, Infosys Enterprise Architecture Survey 2008-2009*.

Olsson, T., Toll, D., and Ericsson, M. (2014). Evaluation of a Static Architectural Conformance Checking Method in a Line of Computer Games. In *QoSA '14 Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures* (pp. 113–118). ACM Press.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058.

Passos, L., Terra, R., Valente, M. T., Diniz, R., and Das Chagas Mendonca, N. (2010). Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software*, *27*(5), 82–89.

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2008). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77.

Perry, D. E., and Wolf, A. L. (1992). Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, *17*, 40 – 52.

Podgurski, A., and Clarke, L. A. (1990). A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, *16*(9), 965–979.

Pruijt, L., and Brinkkemper, S. (2014). A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking. In *WICSA 2014 Companion Volume* (pp. 1–8). ACM Press.

Pruijt, L., Köppe, C., and Brinkkemper, S. (2013a). Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparison of Tool Support. In *2013 IEEE International Conference on Software Maintenance* (pp. 220–229). IEEE Computer Society Press.

Pruijt, L., Köppe, C., and Brinkkemper, S. (2013b). On the Accuracy of Architecture Compliance Checking: Accuracy of Dependency Analysis and Violation Reporting. In H. Kagdi, D. Poshyvanyk, & M. Di Penta (Eds.), *21st International Conference on Program Comprehension* (pp. 172–181). San Francisco, CA, USA: IEEE Computer Society Press.

Pruijt, L., Köppe, C., Brinkkemper, S., and van der Werf, J. M. (2015). The Accuracy of Dependency Analysis in Static Architecture Compliance Checking. *Submitted*.

Pruijt, L., Köppe, C., van der Werf, J. M., and Brinkkemper, S. (2014). HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14* (pp. 851–854). ACM Press.

Pruijt, L., Slot, R., Plessius, H., Bos, R., and Brinkkemper, S. (2012). The Enterprise Architecture Realization Scorecard: A Result Oriented Assessment Instrument. In S. Aier, M. Ekstedt, F. Matthes, E. Proper, & J. L. Sanz (Eds.), *Trends in Enterprise Architecture Research* (Vol. LNBIP 131, pp. 300–318). Springer Berlin Heidelberg.

Pruijt, L., Slot, R., Plessius, H., and Brinkkemper, S. (2013c). The EARScorecard – An Instrument to Assess the Effectiveness of the EA Realization Process. *Journal of Enterprise Architecture*, *09-02*(May), 20–31.

Pruijt, L., Wiersema, W., and Brinkkemper, S. (2013d). A Typology Based Approach to Assign Responsibilities to Software Layers. In *Proceedings of the 20th Conference on Pattern Languages of Programs (PLoP '13)*. ACM Press.

Rahimi, R., and Khosravi, R. (2010). Architecture conformance checking of multi-language applications. *International Conference on Computer Systems and Applications*, 1–8.

Ross, J. (2003). Creating a strategic IT architecture competency: Learning in stages. *MIS Quarterly*, *2*(2).

Rozanski, N., and Woods, E. (2005). *Software Systems Architecture*. Addison-Wesley.

Rutar, N., Almazan, C. B., and Foster, J. S. (2004). A Comparison of Bug Finding Tools for Java. In *15th International Symposium on Software Reliability Engineering* (pp. 245–256). Ieee.

Saha, P. (2004). *Analyzing the open group architecture framework from the geram perspective*. *The Open Group, Tech. Rep*.

Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. In *Conference on Object oriented programming systems languages and applications* (pp. 167–176).

Saraiva, J., Soares, S., and Castor, F. (2010). Assessing the impact of AOSD on layered software architectures. In *European Conference on Software Architecture* (pp. 344–351).

Sarkar, S., Rama, G., and Shubha, R. (2006). A method for detecting and measuring architectural layering violations in source code. In *APSEC*.

Savolainen, J., and Myllarniemi, V. (2009). Layered architecture revisited—Comparison of research and practice. In *WICSA/ECSA* (pp. 317–320).

Schelp, J., and Stutz, M. (2007). A balanced scorecard approach to measure the value of enterprise architecture. *Journal of Enterprise Architecture*, *3*(4), 8–14.

Shaw, M., and Clements, P. (2006). The golden age of software architecture. *IEEE Software*, *23*(2), 31–39.

Shaw, M., and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

Simon, D., Fischbach, K., and Schoder, D. (2013). An Exploration of Enterprise Architecture Research. *Communications of AIS*, *32*(January 2013), 1–72.

Snoeck, M., Poelmans, S., and Dedene, G. (2000). A layered software specification architecture. In *Conceptual Modeling—ER 2000*.

Stafford, J. A., and Wolf, A. L. (2001). Architecture-level dependence analysis for software systems. *International Jounal of Software Engineering and Knowledge Engineering*, *11*(4), 431–451.

Sutton, A., Maletic, J. I., and Ohio, K. (2007). Mappings for Accurately Reverse Engineering UML Class Models from C ++. *Information and Software Technology*, *49*(3), 212–229.

Tamm, T., Seddon, P., Shanks, G., and Reynolds, P. (2011). How Does Enterprise Architecture Add Value to Organisations? *Communications of the AIS*, *28*(1).

Tempero, E. (2009). How Fields are Used in Java: An Empirical Study. In *Australian Software Engineering Conference (ASWEC 2009)* (pp. 91–100).

Tempero, E., Noble, J., and Melton, H. (2008). How do Java programs use inheritance? An empirical study of inheritance in Java software. In *European Conference on Object- Oriented Programming (ECOOP)* (pp. 667–691). Springer, Berlin.

Tempero, E., Yang, H. Y., and Noble, J. (2013). What programmers do with inheritance in java. In *27th European Conference on Object Oriented Programming* (pp. 577–601). Springer, Berlin.

Terra, R., and Valente, M. (2009). A dependency constraint language to manage object oriented software architectures. *Software: Practice and Experience*, *39*(12), 1073–1094.

The Open Group. (2009). The Open Group Architecture Framework: Version 9, Enterprise Edition.

Tichelaar, S., Ducasse, S., and Demyer, S. (2000). Famix and xmi. *Proceedings Workshop on Exchange Formats*, 296–299.

Van der Raadt, B., Bonnet, M., Schouten, S., and van Vliet, H. (2010). The relation between EA effectiveness and stakeholder satisfaction. *Journal of Systems and Software*, *83*(10), 1954–1969.

Van der Raadt, B., Slot, R., and Vliet, H. (2007). Experience Report: Assessing a Global Financial Services Company on its Enterprise Architecture Effectiveness Using NAOMI. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)* (p. 218b–218b). IEEE.

Van Eyck, J., Boucké, N., Helleboogh, A., and Holvoet, T. (2011). Using code analysis tools for architectural conformance checking. In *Proceeding of the 6th international workshop on SHAring and Reusing architectural Knowledge - SHARK '11* (pp. 53–54). New York, New York, USA: ACM Press.

Van Steenbergen, M., Schipper, J., Bos, R., and Brinkkemper, S. (2010). The Dynamic Architecture Maturity Matrix: Instrument Analysis and Refinement. In A. Dan, F. Gittler, & F. Toumani (Eds.), *Proceedings of Trends in*

Enterprise Architecture Research (Vol. 6275, pp. 48–61). Springer Berlin Heidelberg.

Van Steenbergen, M., van den Berg, M., and Brinkkemper, S. (2007). A Balanced Approach to Developing the Enterprise Architecture Practice. In J. Filipe, J. Cordeiro, & J. Cardoso (Eds.), *Proceedings of the International Conference on Enterprise Information Systems* (pp. 240–253). Springer.

Van Zeist, B., Hendriks, P., Paulussen, R., and Trienekens, J. (1996). *Kwaliteit van software producten*. Kluwer, Deventer, Netherlands.

Wagter, R., Berg, M. van den, Luijpers, J., and Steenbergen, M. van. (2005). *Dynamic Enterprise Architecture: How to Make It Work*. John Wiley & Sons, New York.

Winter, K., Buckl, S., Matthes, F., and Schweda, C. M. (2010). Investigating the state-of-the-art in enterprise architecture management method in literature and practice. In *Proceedings of the Mediterranean Conference on Information Systems*.

Winter, R., and Fischer, R. (2007). Essential layers, artifacts, and dependencies of enterprise architecture. *Journal of Enterprise Architecture*, (May), 1–12.

Wirfs-Brock, R., and Wilkerson, B. (1989). Object-oriented design: a responsibility-driven approach. In *Object-oriented programming systems, languages and applications (OOPSLA '89)* (pp. 71–75).

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer.

Woods, E., and Rozanski, N. (2010). Unifying software architecture with its implementation. In *4th European Conference on Software Architecture* (pp. 55–58). New York, New York, USA: ACM Press.

Yin, R. K. (2014). *Case Study Research : Design and Methods* (Fifth edit). SAGA Publications.

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

Zoller, C., and Schmolitzky, A. (2012). Measuring inappropriate generosity with access modifiers in java systems. In *Joint Conf. of Int. Workshop on Software Measurement and Conf. on Software Process and Product Measurement* (pp. 43–52). IEEE.

Publication List

Publications on which this dissertation is based

Pruijt, L., and Brinkkemper, S. (2014). A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking. In *WICSA 2014 Companion Volume/ First Workshop on Software Architecture Erosion and Architectural Consistency* (pp. 1–8). ACM Press.

Pruijt, L., Köppe, C., and Brinkkemper, S. (2013a). Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparison of Tool Support. In *2013 IEEE International Conference on Software Maintenance* (pp. 220–229). IEEE Computer Society Press.

Pruijt, L., Köppe, C., and Brinkkemper, S. (2013b). On the Accuracy of Architecture Compliance Checking: Accuracy of Dependency Analysis and Violation Reporting. In H. Kagdi, D. Poshyvanyk, & M. Di Penta (Eds.)*, 21st International Conference on Program Comprehension* (pp. 172–181). San Francisco, CA, USA: IEEE Computer Society Press.

Pruijt, L., Köppe, C., Brinkkemper, S., and van der Werf, J. M. (2015). The Accuracy of Dependency Analysis in Architecture Compliance Checking. Submitted for publication.

Pruijt, L., Köppe, C., van der Werf, J. M., and Brinkkemper, S. (2014). HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14* (pp. 851–854). ACM Press.

Pruijt, L., Slot, R., Plessius, H., Bos, R., and Brinkkemper, S. (2012). The Enterprise Architecture Realization Scorecard: A Result Oriented Assessment Instrument. In S. Aier, M. Ekstedt, F. Matthes, E. Proper, & J. L. Sanz (Eds.), *Trends in Enterprise Architecture Research* (Vol. LNBIP 131, pp. 300–318). Springer Berlin Heidelberg.

Pruijt, L., Slot, R., Plessius, H., and Brinkkemper, S. (2013c). The EARScorecard – An Instrument to Assess the Effectiveness of the EA Realization Process. *Journal of Enterprise Architecture*, 09-02(May), 20–31.

Pruijt, L., Wiersema, W., and Brinkkemper, S. (2013d). A Typology Based Approach to Assign Responsibilities to Software Layers. In *Proceedings of the 20th Conference on Pattern Languages of Programs* (PLoP ’13). ACM Press.

Other publications

Pruijt, L., and van der Werf, J.M.E.M. (2015). Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking. In *ECSAW ’15 Proceedings of the 2015 European Conference on Software Architecture Workshop /Second Workshop on Software Architecture Erosion and Architectural Consistency* (Article No. 56). ACM Press.

Köppe, C., and Pruijt, L. (2015). Tackling Real World Complexity in a Software Engineering Student Project - An Experience Report. Presented at *SPLASH-E'15*, Pittsburgh, USA.

Köppe, C., and Pruijt, L. (2015). Tool Demo - Teaching Software Architecture Concepts with HUSACCT. Presented at *SPLASH-E'15*, Pittsburgh, USA.

Wiersema, W., and Pruijt, L. (2015). Logical Layering Heuristic Pattern. In *Proceedings of the 22th Conference on Pattern Languages of Programs* (PLoP ’15). ACM Press.

Köppe, C., and Pruijt, L. (2014). Improving Students ’ Learning in Software Engineering Education through Multi-Level Assignments. In *Proceedings of Fourth Computer Science Education Research Conference, CSERC’14.* Berlin, Germany.

Plessius, H., Slot, R., and Pruijt, L. (2012). On the Categorization and Measurability of Enterprise Architecture Benefits with the Enterprise Architecture Value Framework. In S. Aier, M. Ekstedt, F. Matthes, E. Proper, & J. L. Sanz (Eds.), *Trends in Enterprise Architecture Research*, LNBIP 131 (pp. 79–92).

Pruijt, L., Slot, R., and Plessius, H. (2012). Enterprise Architecture Realization Index. In *Archivalue, Portfolio Management with Enterprise Architecture* (pp. 72–81). Enschede, The Netherlands: Novay.

Pruijt, L., and Wiersema, W. (2011b). Meer inzicht in een gelaagde architectuur, Deel 3: Ontwerpen van een Fysiek Lagenmodel. *Release - Vakblad Voor Software Architecten*, Juni, 15–19.

Pruijt, L., and Wiersema, W. (2011a). Meer inzicht in een gelaagde architectuur, Deel 2: Ontwerpen van een Logisch Lagenmodel. *Release - Vakblad Voor Software Architecten*, Maart, 18–21.

Pruijt, L. (2010). Meer inzicht in een gelaagde architectuur, Deel 1: Uitleg, terminologie en methoden. *Release - Vakblad Voor Software Architecten*, December, 22–26.

Pruijt, L., and Lommers, J. (2003). Productgerichte architectuur. *Software Release Magazine*, 1, 17–20.

Appendix 1: Application Case HUSACCT

To illustrate the applicability of our ACC approach and our tool HUSACCT, as described in Chapter 2-4, a case study is presented in this appendix. The architecture compliance check has been conducted with HUSACCT_4.3.

1.1 Introduction to the Case System

The assessed system is an E-commerce system of a governmental organization that is used by citizens and organizations, for example to register or view data, or to apply for a license. The system is developed in C# and its architecture is based on the .NET common application architecture (MSDN 2009). The system is composed of: 1) multiple web-based client applications for a variety of products and services; 2) one server-side ServiceComponent, the central component of the application that handles and coordinates service request from web client applications; and 3) multiple server-side plug-ins, which handle the specifics in processing of the different products and services.

The architecture compliance check focuses on the central ServiceComponent, which acts an application specific shell on top of Commerce Server, Microsoft's E-commerce system. The following keywords provide an impression of the responsibilities of ServiceComponent: product catalog management, profile management, basket and payment management, order management.

1.2 Intended Architecture

1.2.1 Intended Architecture as Documented

Prior to the actual compliance check, we have requested and received a description of the intended modular software architecture, including the modules, the rules and the mapping of modules to implemented software units. The intended architecture of the ServiceComponent is based on the .NET common application architecture. The architecture has proven to remain stable in the past three years, while the number of products and services, provided to customers of the organization via the E-commerce system, has grown from fifteen to sixty.

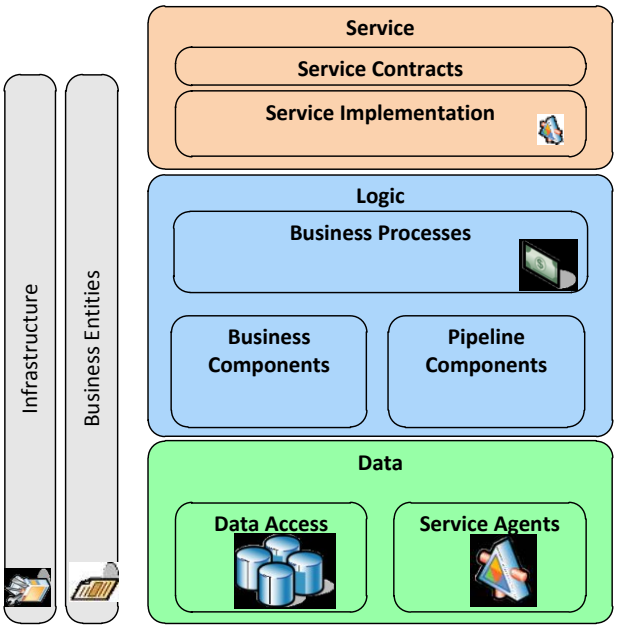

Figure A.1: Overview modular architecture ServiceComponent

The intended architecture of ServiceComponent can be labeled as a Semantically Rich Modular Architecture (SRMA), since it contains modules of five different types and rules of eight different types. An overview of the intended architecture of the ServiceComponent is shown in Figure A.1 and Figure A.2. The first figure provides a high-level overview. Three layers are distinguished, which have the following responsibilities: 1) the Service layer provides the service interface to the web applications; 2) the Logic layer contains the components responsible for the business logic of the application; and 3) the Data layer is responsible for access of the database and communication with infrastructural services. Furthermore, two commonly used modules are visible: Infrastructure, which contains utilities and other shared functionality, and Business Entities, which contains data transfer objects. The rules of a strict layered style apply here: layers are not allowed to make use of higher level layers, and layers are not allowed to skip a layer in their usage relations. Consequently, the Service layer and Logic Layer are not allowed to use infrastructural libraries that are abstracted by the Data Layer.

More rules may be derived from Figure A.2, which provides an overview of the modules and their intended usage relations in the form of an UML component diagram. Identification of the rules based on the component model in the

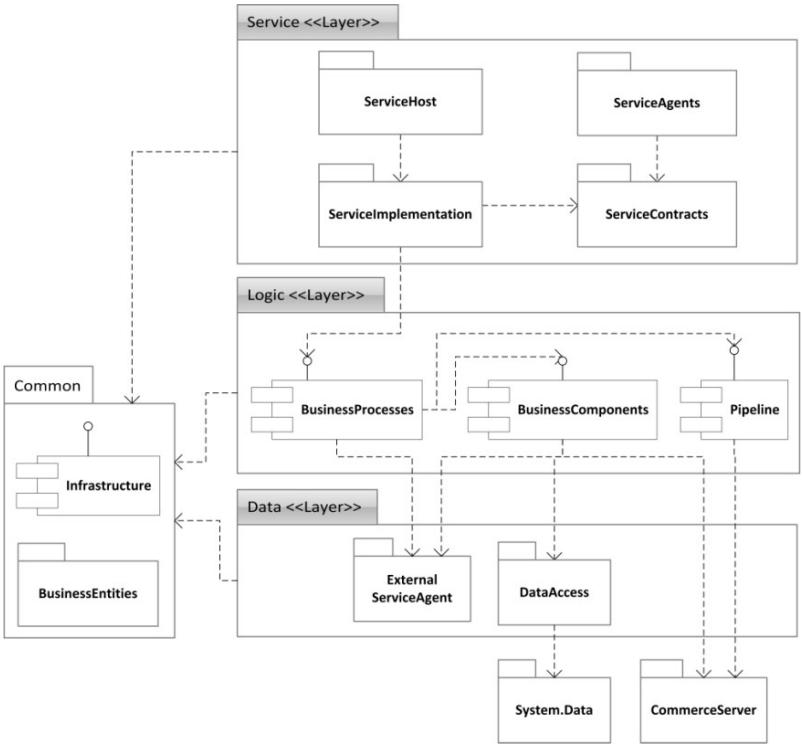

Figure A.2: Component model ServiceComponent

architecture document required interpretation, since an UML component model contains uses dependencies, while constraints need to be derived from the model as rules. The model presented here is an updated version of the originally received model and a set of specified rules. Based on the first conformance checks it seemed that some uses dependencies were missing in the original component model, which was confirmed by the architect. Conversely, several rules were added, mainly based on additional information obtained in an interview of the system's architect. Identification of the rules based on the UML component model in the architecture document required interpretation. A UML component model contains uses dependencies, while constraints need to be derived from the model. The most relevant modules and rules are discussed below. A full specification of the modules, the assigned software units and the checked rules is provided in the next section.

The Service layer is composed of four submodules, of which only ServiceImplementation is allowed to use the Logic layer, and more specifically,

only BusinessProcesses. Furthermore, each submodule of Service is allowed to use only one specified other module within Service.

The Logic layer is composed of three encapsulated modules, BusinessProcesses, BusinessComponents, and Pipelines, which may be used only via their interfaces. Furthermore, it is visible that only BusinessComponents and Pipelines are allowed to use Microsoft's CommerceServer.

The Data layer is composed of two modules, which may be used by a few modules only: DataAccess only by BusinessComponents, and Serviceagent only by BusinessProcesses and BusinessComponents. DataAccess is the only module allowed to use library System.Data.

Finally, nearly all modules in the three layers are allowed to use Common, but module Common.Infrastructure may only be used via its interface. For reasons of clarity, the graphical model is simplified at this point: ServiceContracts, ServiceHost, Pipeline, and DataAccess are exceptions; these modules are not allowed to are not allowed to use module Common.

1.2.2 Intended Architecture in HUSACCT

In HUSACCT, the intended architecture starts with the definition of the modules in the view Define intended architecture, visible in Figure A.3. This view shows the modules in the module hierarchy of the intended architecture of

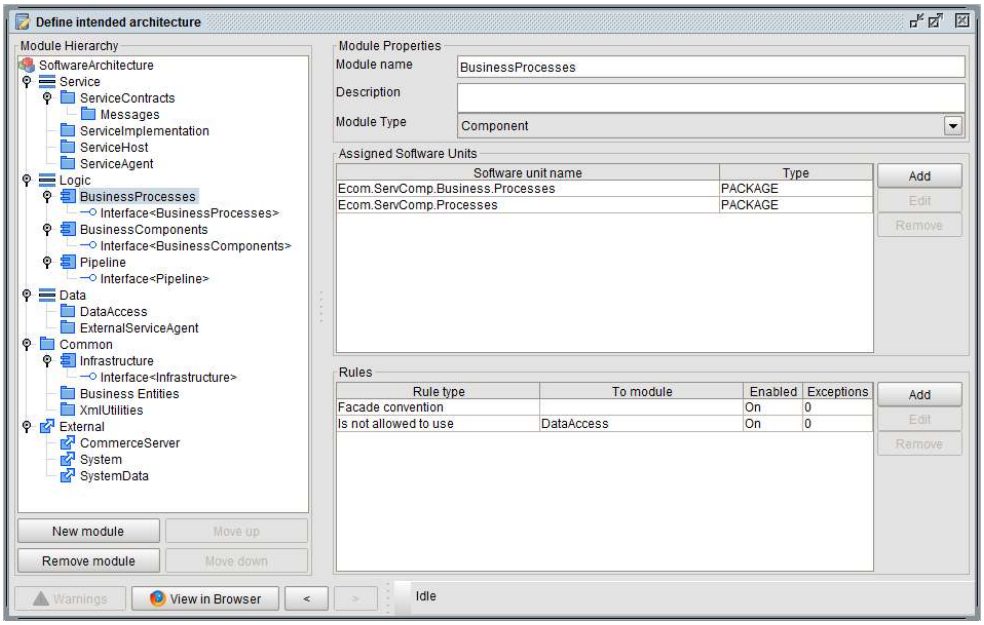

Figure A.3: Intended Architecture as defined in HUSACCT

ServiceComponent. When a module is added, a module type may be selected. As visible in Figure A.3, all five supported module types are present in the intended architecture: components, interfaces, layers, subsystems, and external systems.

Assignment of implemented software units to the intended modules is supported in this view too. The figure shows that module BusinessProcesses has two assigned software units in the implemented architecture. Software units (packages or namespaces, and classes) can be assigned easily, subsequent to source code analysis, by selection of units in an overview.

Rule definition is enabled from the same view as well. Figure A.3 shows that two rules are defined for the selected module BusinessProcesses. One rule is of type Facade convention, which forbids usage of the component other than via its interface(s). This rule is automatically generated, when a module of type Component is created. The other rule is of type "Is not allowed to use", and it restricts the usage of module Data.DataAccess.

1.2.3 All Modules with Assigned Software Units

A table with all modules, their type, and the assigned software units per module is provided below. The table is generated as part of the intended architecture report.

Table A.1: All modules with assigned software units

Id	Module	Assigned Software Units
1	Service (layer)	
2	ServiceContracts (subsystem)	Ecom.ServComp.Service.Contracts (package)
3	Messages (subsystem)	Ecom.ServComp.Service.Contracts.Messages (package)
4	ServiceImplementation (subsystem)	Ecom.ServComp.Service.Implementation (package)
6	ServiceAgent (subsystem)	Ecom.ServComp.Integration (package)
		Ecom.ServComp.Service.Agent (package)
7	Logic (layer)	
8	BusinessProcesses (component)	Ecom.ServComp.Business.Processes (package)
		Ecom.ServComp.Processes (package)
9	Interface<BusinessProcesses> (interface)	Ecom.ServComp.Processes.Interfaces (package)
10	BusinessComponents (component)	Ecom.ServComp.BusinessComponents (package)
11	Interface<BusinessComponents> (interface)	Ecom.ServComp.BusinessComponents.Interfaces (package)
12	Pipeline (component)	Ecom.ServComp.Pipeline (package)
13	Interface<Pipeline> (interface)	Ecom.ServComp.Pipeline.Interfaces (package)
14	Data (layer)	
15	DataAccess (subsystem)	Ecom.ServComp.Data (package)
16	ExternalServiceAgent (subsystem)	Ecom.ServComp.ServiceAgents (package)
		Ecom.Xml (package)
17	Common (subsystem)	
18	Infrastructure (component)	Ecom.ServComp.Infrastructure (package)
19	Interface<Infrastructure> (interface)	Ecom.ServComp.Infrastructure.Interfaces (package)
20	Business Entities (subsystem)	Ecom.ServComp.Business.Entities (package)
21	XmlUtilities (subsystem)	Ecom.ServComp.XmlUtilities (package)
22	External (external library)	xLibraries (package)
23	CommerceServer (external library)	xLibraries.Microsoft.CommerceServer (library)
24	System (external library)	xLibraries.System (library)
25	SystemData (external library)	xLibraries.System.Data (library)

1.2.4 All Architectural Rules with Exceptions

A table with all rules, including their exceptions is provided below. The table shows that 17 rules of eight different types of rules are included in the intended architecture. The table is generated as part of the intended architecture report.

Table A.2: All rules and exceptions

Id	Exception	From module	Rule type	To module	Expression
1		Common.Infrastructure	Facade convention		
2		Common	Is only allowed to use	External	
3		Data.DataAccess	Is the only module allowed to use	External.SystemData	
4		Data	Is not allowed to back call		
5		Logic.BusinessComponents	Facade convention		
6		Logic.BusinessComponents	Is not allowed to use	Logic.BusinessProcesses	
7		Logic.BusinessProcesses	Is not allowed to use	Data.DataAccess	
8		Logic.BusinessProcesses	Facade convention		
9		Logic.Pipeline	Is not allowed to use	Data	
10		Logic.Pipeline	Is the only module allowed to use	External.CommerceServer	
	Exception	Logic.BusinessComponents	Is allowed to use	External.CommerceServer	
11		Logic.Pipeline	Facade convention		
12		Logic	Is not allowed to back call		
13		Service.ServiceAgent	Is only allowed to use	Service.ServiceContracts	
	Exception	Service.ServiceAgent	Is allowed to use	Common	
	Exception	Service.ServiceAgent	Is allowed to use	External.System	
14		Service.ServiceContracts.Messages	Naming convention		*Response
	Exception	Service.ServiceContracts.Messages	Naming convention exception		*Request
15		Service.ServiceContracts	Is only allowed to use	External	
16		Service.ServiceImplementation	Is only allowed to use	Service.ServiceContracts	
	Exception	Service.ServiceImplementation	Is allowed to use	Common	
	Exception	Service.ServiceImplementation	Is allowed to use	Common.Business Entities	
	Exception	Service.ServiceImplementation	Is allowed to use	Logic.BusinessProcesses	
	Exception	Service.ServiceImplementation	Is allowed to use	External.System	
17		Service	Is not allowed to skip call		

1.3 Implemented Architecture

To view the implemented architecture, the code needs to be analyzed first. When the code analysis process is finished, an overview of the analysed architecture is provided as visible in Figure A.4. The figure shows the Decomposition View of the implemented architecture of ServiceComponent (ServComp in the implementation). The software units (packages or namespaces, classes, inner classes, and interfaces) within the application may be selected and opened. Statistical data is provided for the whole application, while unit specific statistical data is shown after selection of a package or class. In this case, the number of classes and lines of code of the system are very high compared to the number of dependencies. This may be explained by the fact that the packages ServiceAgents and Xml contain many generated classes, which hold many lines of code, but only a few lines of C#. The namespaces of interest to the ACC count 31 KLOC, including commented lines and blank lines. The number of classes is including inner classes, but excluding anonymous classes.

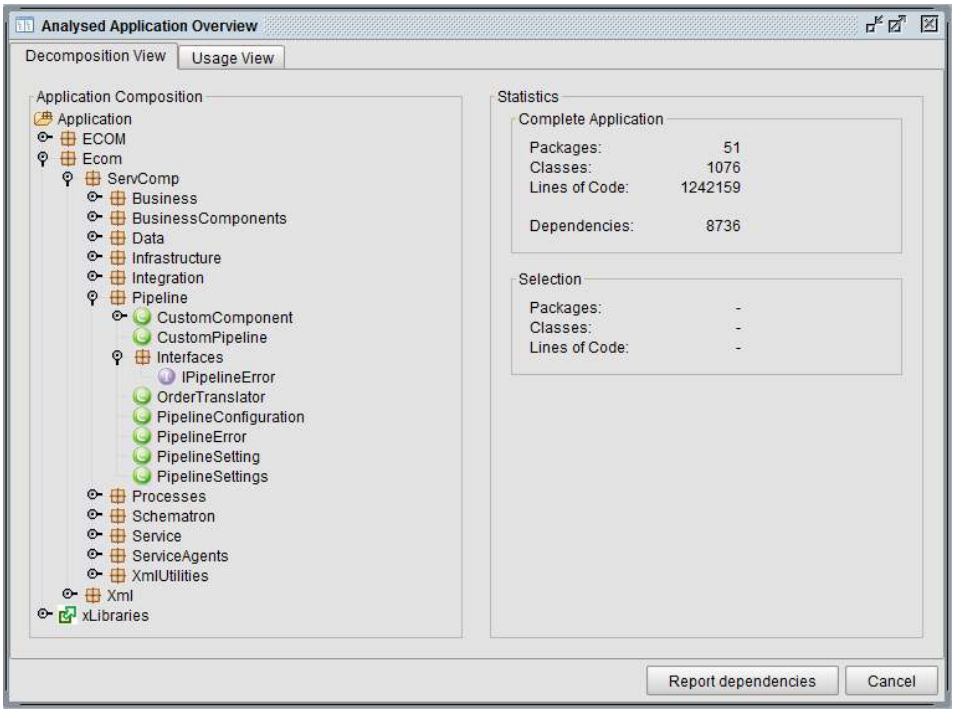

Figure A.4: Analysed application overview: Decomposition View

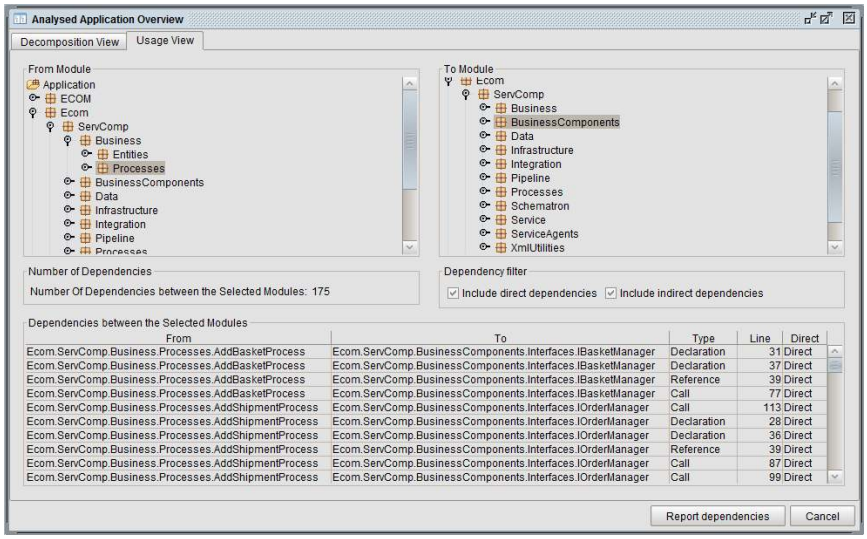

Figure A.5: Analysed application overview: Usage View

Figure A.5 shows the Usage View tab of the Analysed application overview. Within this view, usage relations between software units in the application can be browsed, by selecting a From module and a To module. As a result, the number of dependencies between the modules is calculated and shown, and the dependencies between the modules are presented in the table at the bottom of the view.

The Decomposition View and Usage View form powerful means for manual architecture reconstruction work. In addition, implemented architecture diagrams may be used to study the implemented architecture. Figure A.6 shows the top-level of the implemented architecture of the ServiceComponent.

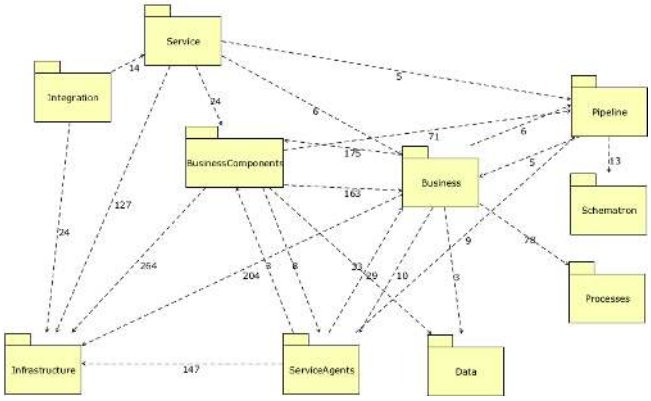

Figure A.6: Implemented architecture diagram, top-level view

1.4 Architecture Compliance Check

Activation of the compliance check starts up a process that iterates through the rules, and for each rule it checks if there is a class or dependency that violates the rule, including its exceptions. Each type of rule has its own checking algorithm. An overview of the results of the ACC of the ServiceComponent is presented in the table below. The table is generated as part of a violation report in spreadsheet format. The report also contains an overview of not violated rules, a sheet with all the violations, and a sheet with statistics on the frequency of dependency types and subtypes over all violating dependencies. The table below shows that ten of the seventeen rules are violated, with a total of 654 violations.

Table A.3: All violated rules with the numbers of reported violations

Id	Logical module from	Rule type	Logical module to	Violations
1	Common.Infrastructure	Facade convention	Common.Infrastructure	496
2	Data.DataAccess	Is the only module allowed to use	External.SystemData	7
3	Data	Is not allowed to back call	Data	12
4	Logic.BusinessComponents	Facade convention	Logic.BusinessComponents	5
5	Logic.BusinessProcesses	Is not allowed to use	Data.DataAccess	3
6	Logic.BusinessProcesses	Facade convention	Logic.BusinessProcesses	6
7	Logic.Pipeline	Is the only module allowed to use	External.CommerceServer	3
8	Logic.Pipeline	Facade convention	Logic.Pipeline	81
9	Service.ServiceAgent	Is only allowed to use	Service.ServiceContracts	2
10	Service.ServiceImplementation	Is only allowed to use	Service.ServiceContracts	39
			Total:	**654**

Figure A.7 shows the Validate conformance, Violations Per Rule view within HUSACCT, with comparable information as in the table above, from the report. In addition, this tab lists the violations for a selected rule. A double click on a

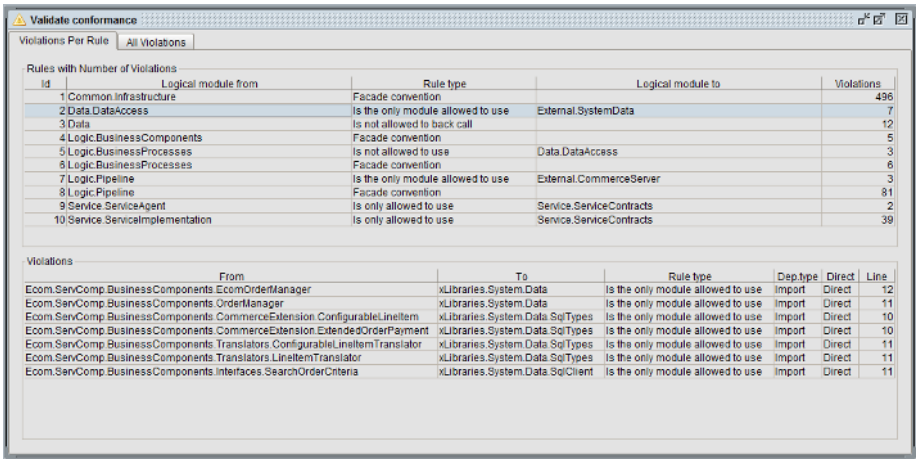

Figure A.7: Validate conformance view: Violations Per Rule

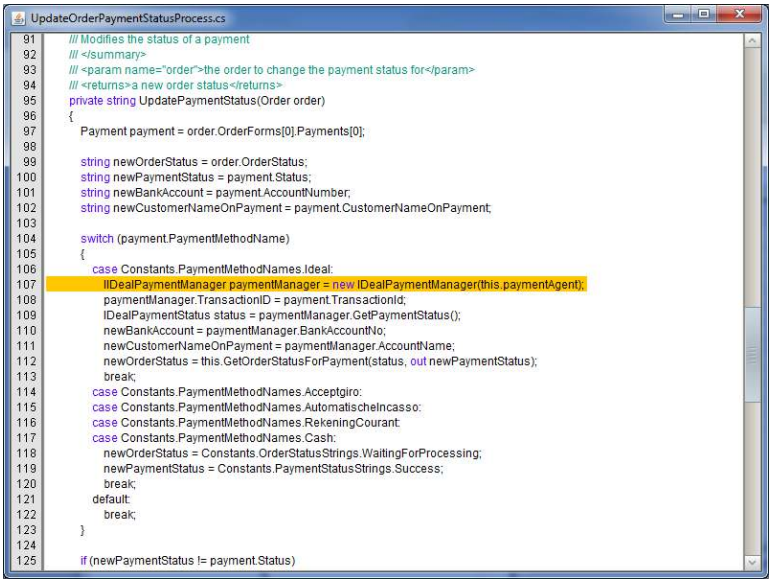

Figure A.8: Code viewer highlighting a line that contains a violation

violation activates the code viewer, which shows the source code of the related from class and highlights the line which hold the violating code construct. An example is visible in Figure A.8.

In addition, the reported violations may be shown in intended architecture diagrams, and in implemented architecture diagrams. The figures below show

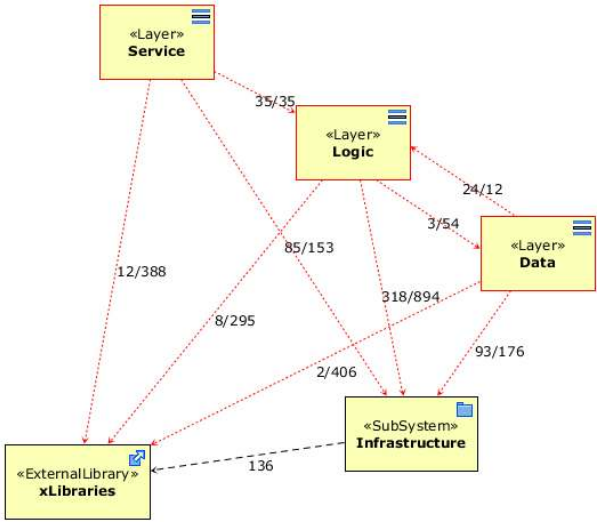

Figure A.9: Intended architecture diagram, top level

intended architecture diagrams. The diagram in Figure A.9 shows the intended top-level modules with their types, and with the violating and non-violating dependency relations in the implementation between the assigned software units. A black, dashed arrow in the diagram represents dependencies only. The related number indicates the number of dependencies. A red, dotted arrow represents violations and dependencies. The first number indicated the number of violating dependencies, while the second indicates the total number of dependencies.

Figure A.10 shows the modules at one decomposition level deeper in the hierarchy. The diagram shows the intended modules within the Service layer, the Logic layer, and the Data layer. For reasons of clarity, the intended modules Common, and External are hidden. Figure A.9 and A.10 show that violations against the architectural rules are not concentrated in one or two modules, but are widespread throughout the code of ServiceComponent.

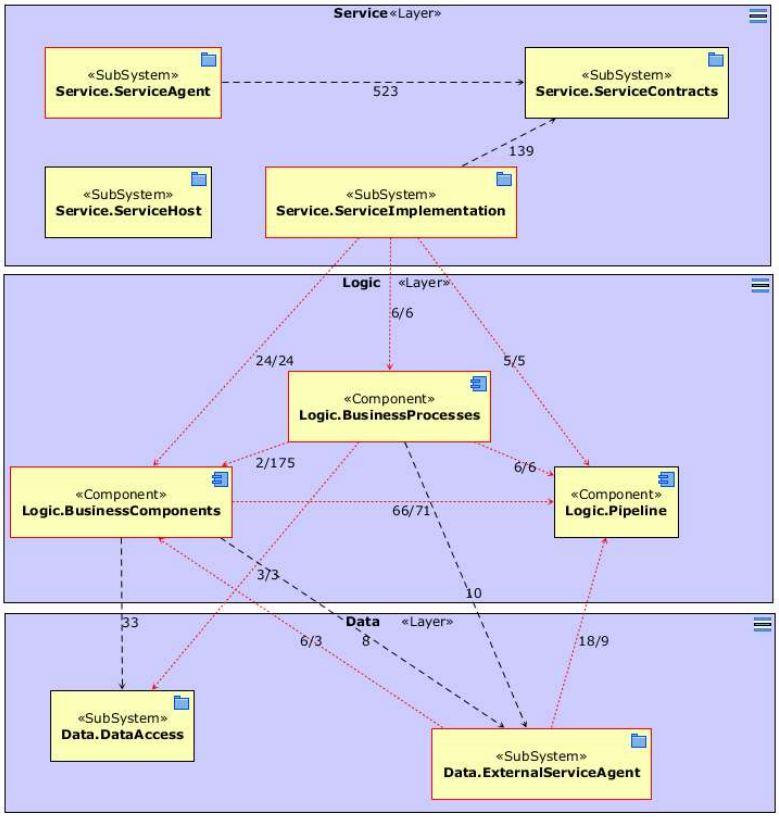

Figure A.10: Intended architecture diagram, decomposition level

1.4.1 Interpretation of the Findings

Positive findings
In general, the module ServiceComponent seemed to be well-structured:

- Layers are distinguished.
- Modules within the layers represent different types of functionality, which are in most cases quite well represented by the namespace names.
- The mapping of the intended architecture on the implemented architecture (the implementation units in the source) is in most cases straightforward, since the intended modules map to one or two complete namespaces only.
- In favor of the encapsulation of four modules, interfaces are provided.
- Data.DataAccess, and Data.ServiceAgent are implemented as adapters to reduce dependencies on infrastructural libraries.
- The intended architecture has been stable through the years, although the number of provided user services, which are processed by the system, has grown the last three years from 15 to 60 different services.

Analysis of the Reported Violations
Ten out of 17 architectural rules were reported in the implementation. The total number of violations is 654 on a total of 6115 dependencies (from classes within namespace Ecom.ServCom), thus 10.7% of the dependencies violate rules.
In our opinion, the violations may be grouped as follows:

- Facade Convention violations form the largest group (violated rule 1, 4, 6, and 8 in the overview of violated rules). This type of violations compromises the encapsulation of components. In case of ServiceComponent, the encapsulation of all four components is compromised, but of component Infrastructure on a large scale. Many of these violations seem to be harmless, but the large number of these violations may hamper the maintainability of the component. In case of PipeLine, seven classes in three different components violate this rule quite frequently, by direct usage of five different classes in Pipeline. In case of the other two components, only a few violations are reported.
- Violations of the layered model (violated rule 2, 3, and 7) undermine the core of the architecture and might have serious consequences. However, the number of violations is small. The twelve back calls from Data to Logic are caused by one class in Data.ExternalServiceAgents.BizTalk, which is using two different classes of module Logic.Pipeline and two different classes of module Logic.BusinessComponents.
  The other violated rules (2 and 7), in effect represent skip call rules, since they represent usage of external services, while the modules assigned as adapters to these external services are skipped. In these cases, only the

dependencies caused by using statements is reported, in fact dependencies on namespaces. In this case, the actual usages of external classes are not reported by HUSACCT, since the used classes are external and not included in using statements (as often in C#).

- Violations to the assigned responsibility of a module (violated rule 5, 9, and 10) indicate that a module has more implemented responsibilities than designed responsibilities, with the risk of duplications and reduced maintainability.
    - Especially module Service.ServiceImplementation requires attention, since it exceeds its designed responsibilities substantially.

Summary

In the last decades, architecture has emerged as a discipline in the domain of Information Technology (IT). A well-accepted definition of architecture is from ISO/IEC 42010: "The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." Currently, many levels and types of architecture in the domain of IT have been defined. We have scoped our work to two types of architecture: enterprise architecture and software architecture.

IT architecture work is demanding and challenging and includes, inter alia, identifying architectural significant requirements (functional and non-functional), designing and selecting solutions for these requirements, and ensuring that the solutions are implemented according to the architectural design. To reflect on the quality of architecture work, we have taken ISO/IEC 8402 as a starting point. It defines quality as "the totality of characteristics of an entity that bear on its ability to satisfy stated requirements". We consider architecture work to be of high quality, when it is effective; when it answers stated requirements.

Although IT Architecture has been introduced in many organizations, the elaboration does not always proceed without problems. In the domain of enterprise architecture, most practices are still in the early stages of maturity with, for example, low scores on the focus areas 'Development of architecture' and 'Monitoring' (of the implementation activities). In the domain of software architecture, problems of the same kind are observed. For instance, architecture designs are frequently poor and incomplete, while architecture compliance checking is performed in practice on a limited scale only.

With our work, we intend to contribute to the advancement of architecture in the domain of IT and the effectiveness of architecture work by means of the development and improvement of supporting instruments and tools. In line with this intention, the main research question of this thesis is:

How can the effectiveness of IT architecture work be evaluated and improved?

In the domain of software architecture, we have focused on modular architectures, and in particular on the architect's task to monitor the implementation activities and to take care that the solutions are implemented according to the architectural design. Here, Architecture Compliance Checking

(ACC) is a useful approach to bridge the gap between the high-level models of architectural design and the implemented program code. In our view, ACC is not only aimed at the improvement of the consistency of the implemented architecture to the intended architecture, but also on the improvement of the intended architecture, based on the insights gained at implementation level. Since the adoption of ACC-tools is still limited in practice and education, we have conducted a line of research with the focus on ACC support. We started this line of research from a functional point of view, by identifying requirements regarding ACC support, studying existing static ACC tools, and testing and comparing these tools on the identified requirements. Thereafter, we focused our research on the solution of identified problem areas.

An important requirement in our research concerns the support of semantically rich modular architectures (SRMA). We use the term SRMA for an expressive modular architecture description, composed of semantically different types of modules (e.g., layers, subsystems, components), which are constrained by different types of rules, such as basic dependency constraints, constraints related to layers, and naming constraints. In practice and literature, many architectures can be labeled as SRMA, since they contain modules with different semantics.

Based on our comparative tool research, we concluded that only limited support was available for SRMAs. We also concluded that solutions were needed to bridge the gap between modular architectures in software architecture documents on one side, and module and rule models in ACC-tools on the other side. In line with these conclusions, and based on an approach of design science research, we have designed a metamodel for SRMA support. Moreover, we have developed HUSACCT, an ACC-tool that provides extensive and configurable SRMA support. To validate our approach, we have performed ACCs with our tool on professional systems and open source systems, and we have used the tool in bachelor and master courses on software architecture.

Static ACC focuses on the existence of rule violating dependencies between modules. Because of the high number of dependencies at implementation level, accurate tool support is essential for effective and efficient ACC. However, the effectiveness is profoundly dependent on the tool's ability to detect all the dependencies between units in the implemented software and to report violating dependencies. Based on this notion, we have tested tools with the following research question in mind: How accurate do ACC-tools report dependencies and violations against dependency rules? To answer the question, ten tools were tested and compared by means of a custom-made test application. In addition, the code of an open source system was used to compare these tools on the number and precision of reported violation and dependency messages. The test results show large differences between the tools, but the main conclusion is that all tools could

improve the accuracy of the reported dependencies and violations. Based on the test results, ten hard-to-detect dependency types and four challenges in dependency detection are identified. To substantiate the relevance of the findings, the results of a frequency analysis of the hard-to-detect dependencies in five open source systems are presented.

In the design of modular architectures, the assignment of responsibility to modules is an essential step. Based on the allocated responsibilities, architectural rules may be defined to restrict dependencies. Layers are commonly used in modular architectures, but many layered architectures are poorly designed and documented, especially with respect to the assignment of responsibilities. To support software architects in their task to design layered architectures of high quality, we propose and illustrate two novel instruments: the Typology of Software Layer Responsibility (TSLR) and the complementary Responsibility Trace Table (RTT).

Last, but not least, we present our research in the domain of enterprise architecture (EA), aimed on the development of an assessment instrument to measure and improve the EA management function's ability to realize its goals. This research was performed in the context of the ArchiValue project, a collaboration between Novay, APG, the Dutch Tax and Customs Administration, BiZZdesign, University of Twente, and HU University of Applied Sciences Utrecht. The result is the Enterprise Architecture Realization Scorecard (EARS) and an accompanying method to discover the strengths and weaknesses in the realization process of an EA management function. During an EARS-assessment, representative EA goals are selected, and for each goal, the results, delivered during the different stages of the realization process, are analyzed, discussed and valued. The outcome of an assessment is a numerical EARScorecard, explicated with indicator-values, strengths, weaknesses, and recommendations. The EARS instrument, its metamodel and metrics are presented, as well as the accompanying method. Furthermore, two assessment cases are discussed to illustrate the use of the instrument.

Nederlandse Samenvatting

In de afgelopen decennia heeft architectuur zich ontwikkeld tot een discipline binnen het vakgebied van de Informatie Technologie (IT). Een algemeen aanvaarde definitie van architectuur is die van ISO/IEC 42010: "De fundamentele organisatie van een systeem, gerepresenteerd in de componenten van het systeem, hun relaties met elkaar en de omgeving, en de principes die leidend zijn voor het ontwerp en de evolutie". Inmiddels zijn vele niveaus en vormen van architectuur op het gebied van IT gedefinieerd. We hebben ons in ons onderzoek beperkt tot twee types architectuur: enterprise architectuur en software architectuur.

Het werk van IT-architecten is veeleisend en uitdagend. Het omvat onder meer: het identificeren van voor de architectuur relevante eisen (functionele en niet-functionele); het ontwerpen en het selecteren van oplossingen voor deze eisen; en het controleren dat de oplossingen worden uitgevoerd in overeenstemming met de ontworpen architectuur. Om de kwaliteit van een architectuur te kunnen beschouwen, hebben we de ISO/IEC standaard 8402 als uitgangspunt genomen. Deze standaard definieert kwaliteit als "het totaal aan eigenschappen van een entiteit benodigd om aan gestelde eisen te voldoen". In het verlengde van deze definitie stellen we dat een architectuur van hoge kwaliteit is, als die effectief is; de gestelde eisen afdoende beantwoort.

Hoewel IT-architectuur in de afgelopen decennia al in veel organisaties is geïntroduceerd, verloopt de toepassing in de praktijk niet altijd zonder problemen. Op het gebied van enterprise architectuur is aangetoond dat de meeste architectuurfuncties zich nog op de lagere niveaus van maturity (volwassenheid) bevinden; bijvoorbeeld met lage scores op de aandachtsgebieden 'ontwikkeling van de architectuur' en 'monitoring' (van de uitvoeringsactiviteiten). Op het gebied van software architectuur worden soortgelijke problemen waargenomen. Bijvoorbeeld, architectuur-ontwerpen blijken vaak mager en onvolledig, en controle op de naleving van de architectuur gebeurt in de praktijk slechts op beperkte schaal.

Met ons onderzoek willen we een bijdrage leveren aan de vooruitgang van de IT architectuur en door de effectiviteit van werkzaamheden te verhogen. En wel door de ontwikkeling en verbetering van ondersteunende instrumenten en tools. In lijn met deze intentie is de belangrijkste onderzoeksvraag van dit proefschrift:

Hoe kan de effectiviteit van IT-architectuur werk worden geëvalueerd en verbeterd?

Op het gebied van software architectuur hebben we ons gericht op modulaire architecturen, met bijzondere aandacht voor de taak van de architect om de ontwikkelactiviteiten te controleren. Architecture Compliance Checking (ACC) is een bruikbare benadering om te controleren of de implementatie van een informatiesysteem overeenkomt met de ontworpen architectuur. Met ACC kan de kloof worden overbrugd tussen de abstracte architectuurmodellen van een software systeem enerzijds en de programmacode anderzijds. Naar onze mening is ACC ook goed bruikbaar voor de verbetering van de beoogde architectuur op basis van inzichten die worden verkregen door de link naar het implementatieniveau heel concreet te leggen. Omdat ACC-instrumenten in de praktijk en in opleidingen nog slechts beperkt worden ingezet, hebben we een lijn van onderzoek opgezet met de focus op ACC ondersteuning. We zijn vanuit een functioneel perspectief begonnen, met het identificeren van eisen met betrekking tot ACC ondersteuning. Daarna hebben we bestaande statische ACC-tools bestudeerd, en we hebben die instrumenten getest op basis van de eerder geïdentificeerde eisen. Vervolgens hebben we ons onderzoek gericht op enkele geconstateerde problemen.

Een belangrijke eis in ons onderzoek betreft de ondersteuning van semantisch rijke modulaire architecturen (SRMAs). Wij gebruiken de term SRMA voor een expressieve modulaire architectuur beschrijving die is samengesteld uit semantisch verschillende modules (bijvoorbeeld software lagen, subsystemen, componenten), die beperkt worden door verschillende soorten regels. Enkele voorbeelden van verschillende typen regels zijn elementaire afhankelijkheidsbeperkingen (module A mag module B niet gebruiken), beperkingen met betrekking tot afhankelijkheden tussen lagen, of beperkingen met betrekking tot de naamgeving van klassen en packages. Veel van de in de praktijk en in de literatuur voorkomende modulaire software architecturen kunnen worden bestempeld als SRMA, omdat ze modules met een verschillende semantiek bevatten.

Op basis van ons vergelijkende ACC-tool onderzoek hebben we geconcludeerd dat er slechts beperkte ondersteuning beschikbaar was voor SRMA's. Daarnaast hebben we geconcludeerd dat oplossingen nodig waren om de kloof te overbruggen tussen enerzijds de modulaire architectuurmodellen zoals architecten die vastleggen in software architectuur documenten, en anderzijds de beoogde architectuurmodellen zoals die in ACC-tools worden vastgelegd. In lijn met deze bevindingen, en op basis van een benadering van design science research, hebben we een metamodel voor SRMA ondersteuning ontworpen. Bovendien hebben we HUSACCT ontwikkeld: een ACC-tool die uitgebreide en configureerbare SRMA ondersteuning biedt. Om onze aanpak te valideren, hebben we compliance checks met HUSACCT uitgevoerd op professionele systemen en open source systemen.

Daarnaast hebben we HUSACCT gebruikt in bachelor- en mastercursussen op het gebied software architectuur.

Een andere functionele eis die wij met betrekking tot ACC-tools als uitgangspunt namen, is dat de analyse van de programmacode nauwkeurige resultaten moet opleveren. Statische ACC richt zich op de controle van regels die afhankelijkheden tussen modules beperken. Vanwege het grote aantal afhankelijkheden op het niveau van programmacode is een nauwkeurig hulpmiddel essentieel voor een effectieve en efficiënte ACC. De effectiviteit van ACC wordt sterk beïnvloed door het vermogen van het ondersteunende ACC-tool om alle afhankelijkheden tussen eenheden in de geïmplementeerde software te detecteren en alle afhankelijkheden te rapporteren die regels overtreden. Op basis van deze gedachte hebben wij gereedschappen getest en wel met de volgende onderzoeksvraag in gedachten: Hoe nauwkeurig rapporteren ACC-tools afhankelijkheden en overtredingen van de afhankelijkheidsregels? We hebben tien ACC-tools getest en onderling vergeleken op basis van een op maat gemaakte test applicatie. Bovendien, hebben we de code van een open-source systeem gebruikt om de tools te vergelijken op aantal en precisie van de gerapporteerde overtredingen en afhankelijkheden. De testresultaten tonen aan dat alle tools de nauwkeurigheid van de gerapporteerde afhankelijkheden en overtredingen kunnen verbeteren. Maar ook dat er grote verschillen in nauwkeurigheid bestaan tussen de tools. Verder zijn op basis van de testresultaten tien moeilijk te detecteren afhankelijkheidstypes geïdentificeerd en vier uitdagingen met betrekking tot afhankelijkheidsdetectie. De relevantie van deze bevindingen hebben we vervolgens aangetoond door middel van een frequentieanalyse van de moeilijk te detecteren afhankelijkheidstypes in vijf open-source systemen.

Een tweede lijn van onderzoek op het gebied van software architectuur heeft zich gericht op de toewijzing van verantwoordelijkheid aan modules; een essentiële stap in het ontwerp van een modulaire software architectuur. Gebaseerd op de toegewezen verantwoordelijkheden, kunnen architectuurregels worden gedefinieerd om afhankelijkheden te beperken. Lagen worden vaak gebruikt in modulaire architecturen, maar veel gelaagde architecturen blijken slecht te worden ontworpen en gedocumenteerd; in het bijzonder met betrekking tot de toewijzing van verantwoordelijkheden. Om software architecten te ondersteunen in hun taak om een gelaagde architectuur van hoge kwaliteit te ontwerpen, presenteren en illustreren we twee nieuwe instrumenten: de Typology of Software Layer Responsibility (TSLR) en de complementaire Responsibility Trace Table (RTT).

Tenslotte presenteren wij ons onderzoek op het gebied van enterprise architectuur (EA) dat we hebben gericht op de ontwikkeling van een evaluatie-instrument voor het meten en verbeteren van de mogelijkheden van de EA management functie om haar doelstellingen te realiseren. Dit onderzoek werd uitgevoerd in het kader van het ArchiValue project, een samenwerking tussen Novay, APG, de Nederlandse Belastingdienst, BiZZdesign, Universiteit Twente, en Hogeschool Utrecht. Het resultaat is de Enterprise Architecture Realisation Scorecard (EARS) en een bijbehorende methode om de sterke en zwakke punten in het realisatieproces van een EA management functie te ontdekken. Tijdens een EARS assessment worden representatieve EA doelen geselecteerd, en voor elk doel worden de resultaten die tijdens de verschillende fases van het realisatieproces zijn opgeleverd, geanalyseerd, besproken en gewaardeerd. De uitkomst van de beoordeling is een numerieke EARScorecard, geëxpliciteerd met indicatorwaarden, sterke en zwakke punten en aanbevelingen. Het EARS instrument wordt gepresenteerd met metamodel en metrieken, evenals de bijbehorende methode. Verder worden twee assessment cases gepresenteerd, waarmee de toepasbaarheid van het instrument in de praktijk wordt geïllustreerd.

Curriculum Vitae

Leo Pruijt (1961) studied Biology at the VU University Amsterdam. After graduation in 1985, he taught Biology and Science at secondary schools on a temporary basis. He embarked on an IT retraining program in 1988, started working in IT in 1989, and continued studying IT, which resulted in an AMBI (Automatisering en Mechanisering van de Bestuurlijke Informatieverwerking) degree in 1993. From 1989 up to 2000, he worked as software developer, pre-sales engineer, information analyst, project leader, advisor methods and techniques, teacher and consultant for the following commercial organizations: Business Case, Westmount Technology, De Amersfoortse Verzekeringen, and ISES International. Since his first job, he has worked with model-driven tools, and he specialized in methods and techniques for the development of information systems, including their architecture.

In September 2000, Leo started as lecturer at the HU University of Applied Sciences Utrecht. Since that time, he has developed and taught courses in requirements engineering, object oriented analysis and design, testing, software architecture, and enterprise architecture. He contributed to the foundation of the Information Systems Architecture Research Group (ADIS) in 2007, and he has joined this group actively ever since. In 2013, Leo also became a member of the EE-Network research and training network (www.ee-network.eu). In line with his work in professional practice, his research focuses on methods, techniques, and tools to enhance the quality and efficiency of the work of IT professionals, and of students.