Architectural Knowledge Management: Supporting Architects and Auditors

Rik Farenhorst
Remco C. de Boer

2009

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

This research has been partially sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge.

MANAGING RISK Parts of this research have been performed in collaboration with DNV-CIBIT. DNV-CIBIT is part of the international knowledge organisation Det Norske Veritas (DNV), and is active in the Netherlands in consulting and education in the area of knowledge management, ICT and information management.

ISBN 978-90-8659-346-0
NUR 982

SIKS Dissertation series No. 2009-32

Printed by GildePrint, Enschede
Printed on 100gsm G-print

Cover illustration by Paul Stellingwerf
`http://www.paulstellingwerf.com`
Typeset in LaTeX by the authors

VRIJE UNIVERSITEIT

Architectural Knowledge Management: Supporting Architects and Auditors

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op maandag 5 oktober 2009 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Rik Farenhorst

geboren te Hoorn

promotor:	prof.dr. J.C. van Vliet
copromotor:	dr. P. Lago

VRIJE UNIVERSITEIT

Architectural Knowledge Management: Supporting Architects and Auditors

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op maandag 5 oktober 2009 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Remco Cornelis de Boer

geboren te Wester-Koggenland

promotor:	prof.dr. J.C. van Vliet
copromotor:	dr. P. Lago

Contents

Acknowledgments

This thesis could not have been written without the support of many people to whom I am grateful. First and foremost I thank my advisors Hans van Vliet and Patricia Lago for their active guidance over the past four years. We have had many good discussions and they were always available for questions or second opinions. Their supervision was for me a perfect mix between strict guidance and sufficient freedom that allowed me to become an independent researcher. In addition, I appreciate the fact that my relationship with Hans and Patricia has been much more than a formal one in many respects. I enjoyed the frequent lunches, informal chats, birthday cake sessions, and of course the 'multi-culti' dinners at Hans' place in Blokker.

I consider myself very lucky that I conducted my research at the Software Engineering Group of VU University Amsterdam. Not only because of the pleasant research environment offered by my both advisors, but also because of the rather large and diverse group of colleagues. I enjoyed the lunches, chats and fun I had with all staff members and fellow PhD students, and I really think life at the VU would have been much less interesting if it wasn't for Rahul Premraj, Natalia Silvis-Cividjian, Steven Klusener, Qing Gu, Adam Vanya, Pieter v.d. Spek, Maryam Razavian, Jorrit Herder, Joost Schalken, Niels Veerman, Dhaval Vyas, and Henriette van Vugt. I thank Peter Roelofsma and his colleagues of the eXamine team for their support during the preparation and execution of a large-scale survey. I thank Johan Hoorn for his methodical support during the design and analysis of a survey conducted as part of my research. Finally, I would like to thank Elly Lammers and Ilse Thomson a lot for all their support with respect to administrative issues.

Special thanks goes to my VU PhD colleagues of the GRIFFIN project, Viktor Clerc and Remco de Boer. Viktor's input during the 'GRIFFIN meetings' and in our room was often very useful and he is often able to look at research problems from multiple angles, which is quite refreshing. Remco has been the perfect 'partner in crime' for me over the past four years. Remco's strong problem-solving and methodical skills combined with his tendency of getting to the bottom of things were really inspiring. I think Remco and I made quite a good team, combining hard work with lots of fun, and I am proud of the joint research results we have obtained. Therefore, writing this thesis together was for me both a logical thing to do and a very pleasant experience.

Although it may appear as if the 'core' of the GRIFFIN project was in Amsterdam, I sincerely believe the GRIFFIN project would not have been complete without the participation of our partners at Rijksuniversiteit Groningen. I would like to thank Paris Avgeriou and Dieter Hammer for hosting periodic research meetings in Groningen, and for coming to the VU a bit more often. All GRIFFIN research meetings were full of interesting discussions and for me these research meetings offered a nice platform for presenting and 'testing out' research ideas and results. I thank Paris and Dieter for their constructive feedback. I also enjoyed the collaboration with the other researchers of the RuG, Anton Jansen, Jan v.d. Ven and Peng Liang. Their input was often valuable, their research ideas were rather complementary to ours, and I really fancied our many social events in the Netherlands and abroad.

One of the strengths of this research has been the collaboration with industry, so I am grateful that our industry partners allowed us to conduct various interesting case studies. I therefore thank

the representatives of the industry partners that participated in the GRIFFIN project over the past few years: Robert Deckers, Eltjo Poort, Michiel Perdeck, Gert Florijn, Bert Bos, Tim Trew, Pierre America, Matthijs Maat, and Martijn van Veelen. The vivid discussions in the yearly research meetings have been very useful. In particular, I would like to thank Dennis Kerssens for paving the way for case study research in Getronics PinkRoccade, and Thiel Chang, Jelle Gerbrandy and Theo Thijssen for allowing various student projects to be executed there. I am also grateful to the architects of the serviceline architecture department of Getronics PinkRoccade with whom I collaborated with great joy for several years, including Barend-Jan van Eijk, Noud van Schrojenstein Lantman, Maarten v.d. Berg, Marieke Raat, Mark Backer, Ab van den Hazel, Frank Fledderus, Bas Veul, Anne Teunissen, Rudi de Keizer, Adrie de Kooter, Joes Kloos, Erik van den Berg, Patrick Reijnen, Thé Nguyen, Nico van der Horst, Mila Soekhoe, Pierre Beynon, and Job Vondeling.

In addition to having so many research associates in the Netherlands, I am privileged that during my research I was able to meet and collaborate with so many well-respected researchers all over the world. I would like to thank Antony Tang, Muhammad Ali Babar, Torgeir Dingsøyr, Rafael Capilla, and Olaf Zimmermann for being part of my 'global' research network. Their input and ideas really contributed to the quality of my research and made conducting my research even more fun and versatile.

Conducting the research presented in this thesis was really a joint effort. In addition to collaboration with aforementioned colleagues and research partners, I have been assisted by many bachelor and master students who deserve to be mentioned as well. The development of some of the methods and tools discussed in this thesis would have taken much more time and effort if it wasn't for Reinier l'Abee, Jeroen Quakernaat, Ronald Izaks, Guido van Stormbroek, Mark van den Berg, Bart Schotten, Rolf Streefkerk, Sander Koers, Daan Zonneveld, Victor Rubio, and Michel van Heijster.

Finally, I am grateful for the support of friends and family the last four years who have stood by me at all times. They successfully distracted me with lots of non-research activities, which allowed me to regain energy when needed. But most grateful I am to my wife Taoying, who inspired me in many ways with her work spirit and endurance. However, her ability to put things into perspective was most priceless, since it helped me to reflect on my work and reminded me of all important things in life.

Amsterdam, June 2009
Rik Farenhorst

Acknowledgments

There are many people to whom I am thankful, and who have contributed in one way or another to the accomplishment of this thesis. Special thanks go to my advisors Hans van Vliet and Patricia Lago, from whom I received the academic guidance I was hoping to find when I decided to return to academia.

I was happy to join the GRIFFIN team that harbored so many good colleagues: Paris Avgeriou, Dieter K. Hammer, Anton Jansen, Peng Liang, and Jan S. van der Ven from the University of Groningen, and Hans van Vliet, Patricia Lago, Viktor Clerc, and Rik Farenhorst from the VU University Amsterdam; our meetings, discussions, and conference trips were not only intellectually, but also socially very satisfying. The fact that Rik and I eventually wrote a single thesis together is in my opinion indicative for the collegiality within our team. Rik, thanks for the countless hours of shared fun. It was a pleasure being 'under pressure' with you.

I appreciate the support and good company I received from all my colleagues at the Software Engineering group. A special word of thanks to Elly Lammers, for without her, life at the university would have been that much more difficult; thanks for always knowing where to go and how things (ought to) work.

Parts of the ideas in this thesis have been pioneered by some of our master's students. In particular, I would like to mention Juan Carlos Loza Torres and Gábor Szabó who worked on early prototypes of tools for quality criteria selection and reuse.

During my research, I have met numerous persons with whom I was privileged to collaborate and discuss interesting ideas. It is impossible to list all the new acquaintances and friends who I met at SHARK, WICSA, QoSA, and so many other venues. Some of these people have, through sharing their ideas and opinions, significantly contributed to my research. In this respect, I particularly want to thank Rich Hilliard, Torgeir Dingsøyr, and Muhammad Ali Babar. I also want to express my gratitude to Eefje Cuppen, who introduced me to the repertory grid technique, and Alex Telea, who provided me with a concise and clear overview of the basics of visualization theory.

Collaboration with industry was one of the key aspects of the GRIFFIN project. I want to thank our industrial partners Astron Foundation, DNV-CIBIT, Getronics PinkRoccade, Philips, and Logica, who provided us with valuable feedback on our progress as well as real problems to tackle. I especially want to thank Viktor Clerc, Mark van Elswijk, Gert Florijn, Mark Hissink Muller, Matthijs Maat, Frank Niessink, Lorenz de Deugd, and Robert Deckers for their contributions to my research.

For me personally, this adventure would not have been possible without the financial support from MVSD. Marcel Matthijs and Mineke Vrijenhoek, I enjoyed working with you once more and truly appreciate the part time job you offered me.

Finally, I want to express my gratitude to my wife Marjoleine. Mar, je was en bent mijn steun en toeverlaat op zoveel manieren en daarvoor ben ik je eeuwig dankbaar.

Amsterdam, June 2009
Remco de Boer

1 Introduction

We are searching for some kind of harmony between two intangibles: a form which we have not yet designed and a context which we cannot properly describe.
– Christopher Alexander

Such as possess the gifts of fortune are easily deprived of them: but when learning is once fixed in the mind, no age removes it, nor is its stability affected during the whole course of life.
– Vitruvius

1.1 Software Engineering

We cannot imagine a world without software anymore. In just a few decades, software has become ingrained in each aspect of everyday life. Supermarkets use electronic bar code scanning at the check out. The money that pays the groceries comes out of an ATM, or even remains in 'electronic' form all the time. When our cars break down, car mechanics first turn to the car's on-board computer for problem diagnosis before they even consider lifting the hood. Plain old telephone systems are gradually being replaced with modern Voice over IP systems that route phone calls over the Internet. Our stock markets, news outlets, planes and trains, our hospitals, mobile phones, TVs and DVRs, indeed, our whole environment runs on software.

Software engineering is the discipline of building large-scale software systems. Within this discipline, a continuous search is going on for better methods and techniques to build better software. This search has led to many approaches that have improved the way in which software is being built (cf. (van Vliet, 2008)). However, even though we have come to rely heavily on software, when compared to other engineering disciplines software engineering is still relatively young.

At the 2008 International Conference on Software Engineering, we celebrated the 40th anniversary[1] of the Garmisch/NATO conference on Software Engineering, which is regarded as the place at which the idea of 'software engineering' was born. Peter Naur, one of the Garmisch/NATO attendees, said about the topic of 'engineering' that "software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem." (Naur and Randell, 1968). This analogy should not be taken to the extreme, however, since there are many differences between software engineering and civil engineering[2].

Over the past 40 years, much of the software design landscape has changed. Modern software development increasingly often takes place in a geographically distributed context involving multiple development groups with different backgrounds and roles. Part or most of a software system is provided through COTS components, outsourcing, open source, multi-party collaboration and distributed development teams. Many of these changes were probably inconceivable for the software engineering pioneers. Nevertheless, many of the topics they discussed then and there still bear relevance to today's world.

1.2 Software Architecture

An exponent of the analogy between civil engineering and software engineering is the emergence of a subdiscipline which has become known as 'software architecture'. Software architecture as we know it today can be traced back to studies by Dijkstra and Parnas in the late 1960s, early 1970s. The general idea behind software architecture can be paraphrased as "Structure is important, and getting the structure right carries benefits." (Clements, online).

The notion of a software 'architecture' is one of the key technical advances in the field of software engineering over the last decades. The software architecture plays an increasingly important role to manage the complex interactions and dependencies in large-scale software development.

[1]Compare this with the 2000 year old treatise *De Architectura* ('On architecture') by the Roman writer, architect, and engineer Vitruvius Pollo (ca. 80 - ca. 20 BC).

[2]To paraphrase Bertrand Meyer at the 2008 ICSE conference: "You cannot fall off the drawing of a bridge", alluding to the lack of a true distinction between specification and implementation in software engineering.

The idea of software architecture *avant la lettre* is clearly present in several discussions at the Garmisch/NATO conference, especially those that center around the design of software. However, it was not until much later that attempts were made to properly define what constitutes software architecture.

In their seminal work on the foundations for the study of software architecture, Perry and Wolf (1992) define software architecture as:

Software Architecture = { Elements, Form, Rationale }

Perry and Wolf describe architectural elements as components and connectors, architectural form as constraints on these elements, and rationale as the motivation for the architectural choices made. In the years thereafter, many other definitions have been coined. Many of these definitions take from Perry and Wolf's the central notion of components and connectors (cf. (Software Engineering Institute, online)). For example, Shaw and Garlan (1996) state that "software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition and constraints on these patterns".

Over the years, much progress has been made in describing architectures in terms of components and connectors. For instance, modeling languages and notations have been proposed to improve stakeholder communication; patterns, vocabularies and product lines have been created to allow reuse; and the use of reference architectures for complex domains has become common practice. Most of these advancements, however, focus on the resulting architecture – the 'end-product' – and leave the decision making that has led to this architecture implicit. Still, it is through this decision making that the architect weighs stakeholder requirements and concerns, identifies constraints, and outlines the eventual solution. Indeed, according to Bass et al. (2003), one of the reasons software architecture is so important is because "architecture manifests the early design decisions".

In short, although considerable progress has been made in describing software architectures, we still lack techniques for capturing, representing and maintaining knowledge *about* software architectures. Consequently, additional – and not exactly quantifiable – costs must be spent each time knowledge has to be transferred within an organization. Moreover, people have to rethink the reasons why certain decisions have been made, or why errors occurred, and they have to repeatedly follow the same paths to achieve similar results. This may result in various problems, including high maintenance costs and high degrees of design erosion (cf. (Bosch, 2004)).

Despite making headway in several other knowledge intensive fields, until recently knowledge management received little attention in the software architecture field. However, the software architecture community has begun to recognize that knowledge management is vital for improving an organization's architectural capabilities. There has been an increased demand for suitable methods, techniques, and tools that support organizations in capturing and maintaining the details on which key architecture design decisions are based. Researchers and practitioners have proposed various approaches to capture and manage architectural design decisions and their rationale (e.g., (Dutoit and Paech, 2001; van der Ven et al., 2006b; Jansen, 2008)). One of the main objectives of these approaches is to help making explicit what is known by architects or implicitly embedded in an architecture. This may include knowledge about the domain analysis, architectural patterns used, design alternatives evaluated, and assumptions underpinning design decisions. Research pertaining to such architectural knowledge benefits from related research fields, especially from knowledge management research.

It is within this context of a shifting focus – from architecture as a high-level design to architecture as a collection of design decisions – that the GRIFFIN project took place. This thesis is one of its results.

1.3 The GRIFFIN project

The GRIFFIN project has been a research project under the auspices of the Netherlands Organisation for Scientific Research (NWO) Jacquard programme on software engineering research. The project ran from April 2005 until April 2009. Over this four year time period, two universities and several industrial organizations collaborated to study architectural knowledge management challenges.

Over the course of the project, about ten organizations in total became involved in the GRIFFIN project, some of them briefly and some others permanently. The organizations involved range from SMEs to multinationals, and from scientific institutes to IT service providers. The research consortium commenced with four industrial partners:

- **RFA**, a large software development organization, responsible for development and maintenance of systems for among others the public sector.
- **DNV**, a company that performs independent software product audits for third parties.
- **VCL**, a large, multi-site consumer electronics organization where embedded software is developed in a distributed setting.
- **PAV**, a scientific organization that has to deal with software development projects spanning a long time frame (up to a period of more than ten years).

The goal of the GRIFFIN project has been to develop notations, tools and associated methods to extract, represent and use architectural knowledge currently not documented or represented in the system. A tentative definition for architectural knowledge from the start of the project was “the integrated representation of the software architecture of a software-intensive system (or a family of systems), the architectural design decisions, and the external context/environment”. Part of the GRIFFIN research consisted of a further refinement of this definition.

The GRIFFIN consortium has been a strong proponent of problem-driven research. Each of the four industrial partners was assigned a dedicated researcher, who performed a series of case studies at that organization and addressed concrete research problems that troubled the organization. Sometimes, these case studies led to iterative refinements of the organization-specific research problems. Other times, parallel research problems were identified. In each case, we identified a coherent theme of architectural knowledge management challenges, related to the organization’s core business.

1.4 Outline of this Thesis

The research presented in this thesis has been conducted by two of the GRIFFIN researchers. Each of the researchers was tied to a particular industrial partner, hence each of the researchers studies architectural knowledge management from a different perspective. The work by Rik Farenhorst at RFA focused on how architectural knowledge management can support architects in their daily work. The work by Remco de Boer at DNV, on the other hand, aimed at employing architectural knowledge management to support auditors who are objective and independent assessors of the quality of a software product. Additionally, this thesis reports on joint work by both researchers. This joint work concerns the development of a general theoretical framework of architectural knowledge management.

This thesis is divided in three main parts that report on the joint work and individual studies of both authors, respectively.

- **Part I: Architectural Knowledge Management**. Provides an overview of architectural knowledge management issues and practices, and provides a general model of architectural knowledge upon which the other parts depend.

- **Part II: Supporting Architects**. Studies architectural knowledge management issues from the perspective of architects. A central theme to this part is how architectural knowledge can be effectively shared.

- **Part III: Supporting Auditors**. Studies architectural knowledge management issues from the perspective of auditors. A central theme to this part is how architectural knowledge relevant to a software product audit can be discovered.

Part I

Architectural Knowledge Management

Based on joint work of both authors

2
Managing Architectural Knowledge

2.1 Introduction

This thesis is about architectural knowlege management, especially about architectural knowledge management to support architects and auditors. The topic of this thesis parallels a current shift in the architecture field, in which architecture is less regarded as components-and-connectors, but more so as the result of a complex knowledge-intensive task ('architecting') that should be supported by knowledge management.

In this part, we examine the state of the art and state of the practice in architectural knowledge management. This leads to a theoretical framework that serves as the basis for the subsequent parts, in which we study how to support architects and auditors, respectively.

2.2 Research Questions

This thesis is about supporting architects and auditors with managing architectural knowledge. In order to arrive at concrete, usable, and successful approaches to architectural knowledge management, we first need to know what architectural knowledge *is* in the first place. Our first research question is thus:

RQ- I.1 *What is architectural knowledge?*

After we have gained an understanding of what architectural knowledge is, we can study *how* best to manage it. Hence, the second research question is:

RQ- I.2 *How can architectural knowledge be managed?*

In the end, however, we are not interested in architectural knowledge management as a mere theoretical exercise; we aim to explore typical problems and challenges to architectural knowledge management in industry. Therefore, our third research question is:

RQ- I.3 *What are typical challenges to architectural knowledge management in practice?*

Answers to these research questions allow us to better define what type of architectural knowledge management is needed to support architects and auditors. From the answers to the research questions in this part, we will derive new research questions which are addressed in Parts II and III of this thesis.

2.3 Research Methods and Methodology

To answer the research questions from §2.2, we employed two main research approaches: desk research and field research. Desk research, also called secondary research, consists of investigating and aggregating existing knowledge that has been published in (scientific) literature. It is therefore typically suitable for research into the state of the art of a particular field. Field research, on the other hand, is a type of primary research that takes part outside an academic 'laboratory' setting. It is therefore particularly useful to investigate the current state of the practice, and to identify current challenges in industrial practice.

We used desk research especially to answer RQ-I.1 and RQ-I.2. We explored the current state of the art in architectural knowledge management by conducting a systematic literature review. This review consisted of three stages. In the first stage, we performed an ad-hoc, preliminary review of architectural knowledge management strategies. In the second stage, based on a predefined protocol we identified, selected, and synthesized all studies that define or discuss 'architectural knowledge'. In the third stage, we employed social network analysis to identify the communities in which architectural knowledge plays a role.

We used field research especially to answer RQ-I.3, and partly to answer RQ-I.2. We followed a typical action research cycle together with the industrial partners of the GRIFFIN project. This action research cycle consisted of 1) the design of an initial theory of architectural knowledge, 2) experimentation with this theory in industrial organizations, 3) identification of mismatches between our theory and industry practice, 4) refinement of the initial theory into a 'core model' of architectural knowledge, and 5) characterization of the use of architectural knowledge in industrial organization and identification of architectural knowledge management challenges.

2.4 Outline of Part I

The remainder of this part contains two main research chapters, followed by a conclusions chapter.

- In Chapter 3 we provide answers to RQ-I.1 (partially) and RQ-I.2. We answer RQ-I.1 from the perspective of existing literature. We examine how architectural knowledge is defined in literature and how the different definitions are related. We answer RQ-I.2 partially from a literature investigation, and partially from our collaborative research with our industrial partners.
- In Chapter 4 we provide answers to RQ-I.1 (partially) and RQ-I.3. Since existing literature does not provide a single view on architectural knowledge, we extend the answer to RQ-I.1 by presenting a core model of architectural knowledge. This core model unifies existing, decision-centric models and provides a common frame of reference regarding what architectural knowledge entails. It will act as an operationalized definition of architectural knowledge throughout this thesis. Experimentation with this core model in four industrial organizations leads to an answer to RQ-I.3. In the remainder of this thesis, we will further examine two types of challenges: how to support architects and how to support auditors with architectural knowledge management.
- In Chapter 5 we revisit the three research questions and summarize the main lessons learned in Part I.

2.5 Publications

Most of the material presented in this part of the thesis has been published previously. The publications on which the chapters in Part I are based are listed in this section.

Parts of Chapter 3, have been published previously as:

- Ali Babar, M., R.C. de Boer, T. Dingsøyr, and R. Farenhorst. Architectural Knowledge Management Strategies: Approaches in Research and Industry. In *2nd Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent* (SHARK/ADI'07), page 2. IEEE Computer Society, 2007.
- de Boer, R.C. and R. Farenhorst. In Search of Architectural Knowledge. In *3rd international workshop on SHAring and Reusing architectural Knowledge* (SHARK'08), pages 71–78. ACM, 2008.

- Farenhorst, R. and R.C. de Boer. Knowledge Management in Software Architecture: State of the Art. In *Software Architecture Knowledge Management: Theory and Practice.* (Ali Babar et al., 2009). Springer, 2009.

Parts of Chapter 4, have been published previously as:

- de Boer, R.C., R. Farenhorst, J.S. van der Ven, V. Clerc, R. Deckers, P. Lago, and H. van Vliet. Structuring Software Architecture Project Memories. In *8th International Workshop on Learning Software Organizations* (LSO'06), pages 39–47, Rio de Janeiro, Brazil, 2006.

- Farenhorst, R., R.C. de Boer, R. Deckers, P. Lago, and H. van Vliet. What's in Constructing a Domain Model for Sharing Architectural Knowledge? In *18th International Conference on Software Engineering and Knowledge Engineering* (SEKE'06), pages 108–113, Knowledge Systems Institute, 2006.

- de Boer R.C., R. Farenhorst, P. Lago, H. van Vliet, V. Clerc, and A. Jansen. Architectural Knowledge: Getting to the Core. In *3rd International Conference on the Quality of Software-Architectures* (QoSA 2007), pages 197–214, Springer, 2007.

3 Architectural Knowledge Management - State of the Art

The main topic of this thesis is architectural knowledge management. But what is *'architectural knowledge management'? And what is 'architectural knowledge', for that matter? This chapter reports on the results of a study in which we examine how architectural knowledge is defined and how the different definitions in use are related. We describe the main views on architectural knowledge and from there discuss the state of the art in architectural knowledge management.*

3.1 Introduction

The software architecture community puts more and more emphasis on managing architectural knowledge. However, there appears to be no commonly accepted definition of what architectural knowledge entails. Nevertheless, architectural knowledge has played a role in discussions on design, reuse, and evolution for over a decade. Over the past few years, the term has significantly increased in popularity and attempts are being made to properly define what constitutes architectural knowledge.

In answer to the question 'what is architectural knowledge?', in §3.2 we describe four main views on architectural knowledge that emerged from a systematic literature review, and explore their commonalities and differences. Using two orthogonal architectural knowledge dimensions, we define four categories of architectural knowledge. The potential knowledge conversions between these categories, which we describe in §3.3, together form a descriptive framework with which different architectural knowledge management philosophies can be typified. This framework shows that the differences between the four views on architectural knowledge are very much related to the different philosophies they are based on.

Whereas traditionally tools, methods, and methodologies for architectural knowledge management were confined to a single philosophy, a trend can be observed towards a more overarching 'decisions-in-the-large' philosophy that unifies aspects of different single-philosophy approaches. In §3.4 we discuss how decision-oriented approaches usually intentionally opt for a codification strategy, while the unintentional practice is to support personalization too. Intentional personalization, or hybrid codification-personalization strategies, are hardly reported. The systematic review protocol that we used to derive the results presented in this chapter can be found in §3.5.

3.2 What is 'Architectural Knowledge'?

To be able to understand what architectural knowledge entails, we have conducted a systematic literature review in which we explored the *'roots'* architectural knowledge has in different software architecture communities. The review revealed four primary views on architectural knowledge. In §3.2.1 we elaborate upon these views, and show there is not a single encompassing definition. In §3.2.2 we identify the current definitions of architectural knowledge and analyze how these definitions relate to the four views on architectural knowledge. Since design decisions seem to be a linking pin, we further investigate the importance of the concept of decisions and how it links the different views. To better compare its different manifestations, in §3.2.3 we propose a framework that distinguishes between different types of architectural knowledge.

3.2.1 Different views on architectural knowledge

Architectural knowledge is related to such various topics as architecture evolution, service oriented architectures, product line engineering, enterprise architecture, and program understanding, to name but a few. In the literature, however, there are four main views on the use and importance of architectural knowledge. Those four views – pattern-centric, dynamism-centric, requirements-centric, and decision-centric – are introduced in this section.

Pattern-centric view

In the mid-nineties, patterns became popular as a way to capture and reuse design knowledge. People were disappointed by the lack of ability of (object-oriented) frameworks to capture the knowledge necessary to know when and how to apply those frameworks. Inspired by the work of Christopher Alexander on cataloging patterns used in civil architecture, software engineers started to document proven solutions to recurring problems.

Initially, patterns focused mainly on object oriented design and reuse; the canonical work in this area is the book by the ‘Gang of Four’ (Gamma et al., 1994). The aim was to let those design patterns capture expert and design knowledge, necessary to know when and how to reuse design and code. Soon, however, the patterns community extended its horizon beyond object-oriented design. Nowadays, patterns exist in many areas, including patterns for analysis (e.g., (Fowler, 1997)), architectural design (e.g., (Fowler et al., 2002; Buschmann et al., 1996)), and the development process (e.g., (Coplien and Schmidt, 1995; Coplien and Harrison, 2004)).

Patterns in software development serve two purposes. Patterns are reusable solutions that can be applied to recurring problems. They also form a vocabulary that provides a common frame of reference, which eases sharing architectural knowledge between developers. Although patterns are usually documented according to certain templates, there is not a standard template used for all patterns. The way in which patterns are codified make them very suitable for human consumption, but less so for automated tools.

Dynamism-centric view

A more formal approach to architectural knowledge can be found in discussions on dynamic software architectures. Systems that exhibit such ‘dynamism’ can dynamically adapt their architecture during runtime, and for example perform upgrades without the need for manual intervention or shutting down. Such systems must be able to self-reflect and ‘reason over the space of architectural knowledge’ (Georgas and Taylor, 2004), which invariably means that – unlike patterns – this architectural knowledge must be codified for consumption by non-human agents.

Since the software itself must understand the architectural knowledge, architecture-based adaptation has to rely on rather formal ways of codification. A survey by Bradbury et al. (2004) reveals that almost all formal specification approaches for dynamic software architectures are based on graph representations of the architectural structure. Purely graph-based approaches use explicit graph representations of components and connectors. In those approaches, architectural reconfiguration is expressed with graph rewriting rules. Other approaches, which use implicit graph representations, rely mainly on process algebra or logic to express dynamic reconfiguration of the architecture.

A particular family of formal languages for representing architectures is formed by so-called ‘architecture description languages’ (ADLs). Although not all ADLs are suitable for use in run-time dynamic systems, all ADLs are based on the same component-connector graph-like representation of architectures (cf. (Medvidovic and Taylor, 2000)).

Requirements-centric view

The architecture is ultimately rooted in requirements. Therefore, architectural knowledge plays a role in enabling traceability in the transition from requirements to architecture. But there is an inverse relation too, namely the fact that ‘stakeholders are quite often not able to specify innovative requirements in the required detail without having some knowledge about the intended solution’ (Pohl and Sikora, 2007). Hence, in order to specify sufficiently detailed requirements, one needs knowledge about the (possible) solutions, which means that requirements and architecture need to be co-developed. This is a subtle, but important difference: the transition-view is a bit older and denotes the believe that problem is followed by solution, whereas the more recent co-development view emphasizes that both need to be considered concurrently.

The relation between requirements and architecture has been a popular subject of discourse since the early 2000s. Although the related STRAW workshop series is no longer organized, many researchers still focus on bridging the gap between requirements and architecture. A survey by Galster et al. (2006) identified and classified the methodologies in this area, including Jackson’s problem frames (later extended to ‘architectural frames’ (Rapanotti et al., 2004)), goal-oriented requirements engineering, and the Twin Peaks model for weaving architecture and requirements (Nuseibeh, 2001a).

Decision-centric view

For many years, software architecture has mainly been regarded as the high-level structure of components and connectors. Many architectural description frameworks, such as the IEEE-1471 standard (IEEE 1471) therefore have a particular focus on documenting the end result of the architecting process.

Nowadays, the view on architecture seems to be shifting from the end result to the rationale behind that end result. This shift stems largely from the lack of solid documentation methodologies for the rationale of architectural designs. More and more researchers (e.g., (Kruchten et al., 2006a; Jansen and Bosch, 2005; van der Ven et al., 2006b; Tyree and Akerman, 2005)) agree that one should consider not only the resulting architecture itself, but also the design decisions and related knowledge that represent the reasoning behind this result. All such architectural knowledge needs to be managed to guide system evolution and prevent knowledge vaporization (Bosch, 2004).

The treatment of design decisions as first-class entities enables the consideration of a wide range of concerns and issues, including purely technical issues, but also business, political, and social ones. Architects need to balance all these concerns in their decision making. To justify the architectural design to other stakeholders, communication of the architectural design decisions plays a key role. In that sense, the decision-centric view is very much related to the (broader) field of design rationale.

3.2.2 Definitions of architectural knowledge

If there's anything clear from the four views on architectural knowledge, it must be that there is not a single encompassing definition of what this knowledge entails. In order to obtain a better understanding of the concept of architectural knowledge, we searched for studies defining or discussing 'architectural knowledge'. We found 116 such studies, 14 of which provide an actual definition. We examined the relations between those definitions through reciprocal translational analysis (Noblit and Hare, 1988). This analysis shows that most of the definitions focus on design decisions, although some refer to solution fragments or elements from the problem domain as well. A subsequent analysis of concepts related to those definitions shows that, contrary to the focus of the definitions, most authors seem to agree that architectural knowledge spans all these areas. We also encountered two surprising definitions: one, used by two authors, from the area of systems design and one over a decade old but remarkably similar to current ones.

One of the first striking observations from the 116 selected primary studies it the very clear upward trend in the number of publications that discuss architectural knowledge. There is undoubtedly an increasing interest in the topic. Fig. 3.1 shows the number of publications per year that discuss and/or define architectural knowledge. The figure clearly shows that in 1992 the concept was coined for the first time and that from that date on every year 'architectural knowledge' crops up in at least one publication. However, until the early 2000s architectural knowledge received fairly little attention. From 2001 onward, the number of publications per year has seen uninterrupted and rapid growth which continues to this date.

Synthesis of architectural knowledge definitions

Although we found 116 studies that discuss architectural knowledge, only 14 of them give a clear definition of what architectural knowledge entails. These definitions are listed in Table 3.1, ordered chronologically by publication date. This table shows that almost all definitions were proposed over the past three years, which corresponds to the increasing interest in architectural knowledge observed earlier.

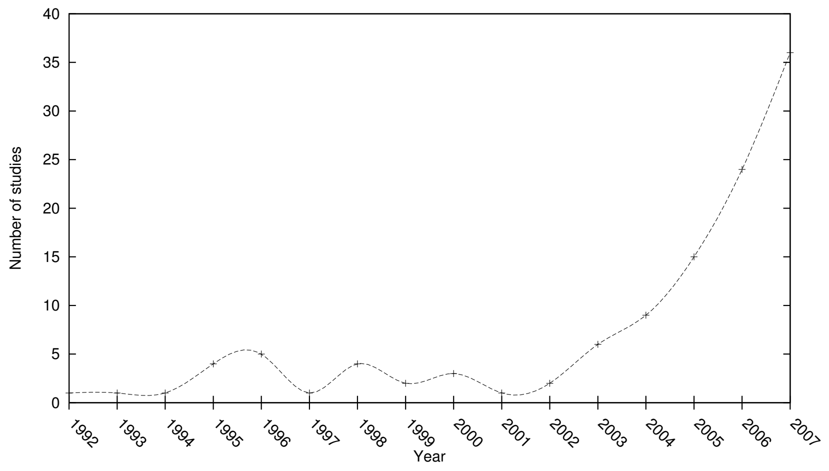


Figure 3.1: Studies that discuss architectural knowledge

Chronologically, a clear outlier in Table 3.1 is the work by Ran and Kuusela (1996). Already in 1996, they are the first to give a definition of architectural knowledge. Moreover, their definition is remarkably similar to the decision-centric view on architectural knowledge currently prevalent in the software architecture community. However, somehow this work seems unrelated to the stir that started in the early 2000s and brought more attention to architectural knowledge, especially in terms of design decisions. On the contrary, as far as we know Ran and Kuusela's work is hardly referenced in current discussions.

In the 14 definitions from Table 3.1, there are 15 concepts that play a major role. Not every definition relies on unique concepts, but similar concepts are used by different authors in different definitions. In Table 3.2, we have made this relation clear by translating each definition of architectural knowledge into the concepts it uses.

If we carefully examine the concepts used in definitions of architectural knowledge, we can identify five higher-level constructs to which those definitions relate: decision, problem domain, solution fragment, implementation, and systems design. The relation between the definitions and the five constructs is represented by the gray areas in Table 3.2. To a certain extent, the high-level constructs correspond to the different views on architectural knowledge (cf. §3.2.1), but there are also differences:

1. **Definitions that center around design decisions**
   By far the largest percentage of the definitions we found focus on design de-

Table 3.1: Definitions of ‘architectural knowledge’

Author(s)	**Architectural Knowledge Definition**
Ran and Kuusela (1996)	To avoid replication when representing variations and alternatives DDT structures architectural knowledge hierarchically into fine-grain elements we call design decisions.
Carayannis and Coleman (2005)	The architectural innovation is dependent on the system designers’ knowledge of the components in the system and their knowledge of the configuration of the components. Henderson and Clark (as cited in Afuah, 1998) show the knowledge as Component Knowledge (CK) and the latter Architectural knowledge (AK).
Chen (2005)	A distinction that is particularly significant in the product innovation context is the distinction between component-specific knowledge and “architectural” knowledge (Henderson and Clark, 1990). Component knowledge is knowledge that concerns a particular aspect of an organization’s product, process or operation. Architectural knowledge, on the other hand, relates to the various ways in which the components are integrated and linked together into a complete system.
Kruchten et al. (2005)	Architectural knowledge consists of architecture design as well as the design decisions, assumptions, context, and other factors that together determine why a particular solution is the way it is.
Kruchten et al. (2006a)	Architectural Knowledge = Design Decisions + Design, derived from ‘Architectural knowledge consists of architecture design as well as the design decisions, assumptions, context, and other factors that together determine why a particular solution is the way it is.’
Kruchten et al. (2006b)	Some researchers are looking into architectural knowledge – that is, architectural design decisions and their rationale.
Ali Babar et al. (2006)	We propose a framework for managing design rationale to improve the quality of architecture process and artifacts. This framework consists of techniques for capturing design rationale, and approach to distill and document architectural information from patterns, and a data model to characterize architectural constructs, their attributes and relationships. These collectively comprise Architectural Design Knowledge (ADK) to support the architecting process.
Lago and Avgeriou (2006); Avgeriou et al. (2007a)	Architectural Knowledge (AK) is defined as the integrated representation of the software architecture of a software-intensive system or family of systems along with architectural decisions and their rationale external influence and the development environment.
Lee and Kruchten (2007)	Software architectural knowledge is composed of the design and the set of decisions made to arrive at the design.
de Boer and van Vliet (2007)	Following a recent trend in software architecture research we refer to the collection of architectural design decisions and the resulting architectural design as ‘architectural knowledge’.
Farenhorst et al. (2007)	[..] not only the architecture design itself is important to capture, but also the knowledge pertaining to it. Often, this so-called architectural knowledge is defined as the set of design decisions, including the rationale for these decisions, together with the resulting architectural design.
Habli and Kelly (2007)	Architectural Knowledge = {drivers, decisions, analysis}
Ali Babar and Gorton (2007)	[The knowledge management component] provides services to store, retrieve, and update artifacts that make up architectural knowledge.
Bahsoon (2007)	We anticipate the architectural knowledge to constitute architectural artifacts such as deployable components and associated specification of what the components provide and require, quality requirements, scenarios corresponding to specific dependability requirements, and possibly dependable styles and patterns.

cisions (Avgeriou et al., 2007a; Ali Babar et al., 2006; de Boer and van Vliet, 2007; Farenhorst et al., 2007; Habli and Kelly, 2007; Kruchten et al., 2006a, 2005, 2006b; Lago and Avgeriou, 2006; Lee and Kruchten, 2007; Ran and Kusela, 1996). Some are limited to design decisions and accompanied rationale, some others are a bit broader and also explicitly refer to the resulting design. The decision construct to a large extent corresponds to the decision-centric view on architectural knowledge.

Table 3.2: Constituents of architectural knowledge definitions

		Kruchten et al. (2006) (IEEE)	Farenhorst et al. (2007)	Ran and Kuusela (1996)	Lee and Kruchten (2007)	De Boer and Van Vliet (2007)	SHARK workshop (2006 / 2007)	Kruchten et al. (2006) (QoSA)	Kruchten et al. (2005)	Habli and Kelly (2007)	Bahsoon (2007)	Babar et al. (2006)	Babar and Gorton (2007)	Carayannis and Coleman (2005)	Chen (2005)
Decision	Architectural Design		x		x	x	x	x	x						
	Design decisions	x	x	x	x	x	x	x	x	x					
	Rationale	x	x				x	(x)	x			x			
Problem Domain	Analysis									x					
	Non-funct. requirement										x				
	Assumption							(x)	x						
	Context						x	(x)	x						
	Driver									x					
	Scenario										x				
Solution Fragment	Patterns										x	x			
	Styles										x				
Implementation	Deployable component										x				
Systems Design	Component configuration													x	x
	Architectural constructs											x			
	Artifact												x		

2. **Definitions that contain elements from the problem domain**
Several definitions contain elements from the problem domain (Avgeriou et al., 2007a; Bahsoon, 2007; Habli and Kelly, 2007; Kruchten et al., 2006a, 2005; Lago and Avgeriou, 2006). Those elements are typically factors that influence the architect's work. None of the definitions solely contains problem domain concepts, but about half of the definitions does take the problem domain into account. The problem domain construct relates to the requirements-centric view on architectural knowledge, especially since the problem domain elements do not stand on their own but are – through the definitions – linked to either decisions or solutions.

3. **Definitions that specify solution fragments**
Closely related to the view of the pattern community on architectural knowledge, some definitions specify solution fragments as key aspects of architectural knowledge (Ali Babar and Gorton, 2007; Ali Babar et al., 2006; Bahsoon, 2007).

But, as with elements from the problem domain, those solution fragments are never considered to fully encompass architectural knowledge.

4. **Definitions that refer to implementation**
   One of the definitions (Bahsoon, 2007) refers to the implementation of architectural solutions in terms of 'deployable components'. This definition has been coined from a perspective of the role of architectural knowledge in dependable systems, and is consequently much related to the dynamism-centric view.

5. **Definitions from the realm of systems design** Two studies apply a surprising definition of architectural knowledge, referred to particularly in studies from systems design and product development, to the world of software-intensive systems (Carayannis and Coleman, 2005; Chen, 2005). This definition is completely orthogonal to other definitions used, and talks about components in a less technical sense than the usual component-connector meaning. One of the authors refers to components as "a particular aspect of an organization's product, process, or operation" (Chen, 2005), whereas the other defines components of a technical system as "products, processes, people, services, and technologies" (Carayannis and Coleman, 2005). Nevertheless, one of the authors uses this definition to assess knowledge creation in different software development settings. This topic would probably interest those who approach architectural knowledge from an orthogonal point of view as well. The systems design view on architectural knowledge does not, however, seem to fit any of the views we identified in §3.2.1.

Table 3.2 also shows two concepts (architectural construct and artifact) that we found to be too broad to fit one particular construct. From the definition alone, it remains unclear to which construct those concepts would or would not belong.

The apparent importance of the decision-centric view in discussing and defining architectural knowledge may be explained when we look at the links between this view and the other views; decisions appear to be the linking pin between the different views.

The relation between patterns and decisions is discussed by Harrison et al. (2007). Their conclusion is that the two are complementary concepts, and that '[u]sing a pattern in system design is, in fact, selecting one of the alternative solutions and thus making the decisions associated with the pattern in the target systems specific context'. Ran and Kuusela (1996) proposed a hierarchical ordering of design patterns in what they call a 'design decision tree'.

The relation between design decisions and requirements can be approached from two directions. Bosch (2004) conceptually divides an architectural design decision into a 'solution part' and a 'requirements part'. The requirements part represents the subset of the system's requirements to which the solution part provides a solution. Van

Lamsweerde (2003), on the other hand, argues that for alternative goal refinements and assignments, 'decisions have to be made which in the end will produce different architectures'.

Formal graph-based architecture representations are especially suitable for automated reasoning. In other words, those representations enable automated agents either take design decisions themselves (Bradbury et al., 2004), or to inform human architects about potential problems and pending decisions (Robbins et al., 1996).

Decisions may indeed be an umbrella concept that unify parts of those different views on architectural knowledge, because they closely relate to various manifestations of architectural knowledge concepts. Of course, there are differences between the views too: patterns are individual solution fragments, formal representations focus on the end result only, and requirements engineering is more occupied with problem analysis than solution exploration.

3.2.3 So, what is Architectural Knowledge?

If we want to better compare the different manifestations of architectural knowledge, it helps to distinguish between different types of architectural knowledge. Two distinctions are particularly useful: tacit vs. explicit knowledge, and application-generic vs. application-specific architectural knowledge.

Nonaka and Takeuchi (1995) distinguish between tacit and explicit knowledge. This distinction is applicable to knowledge in general, and its application to architectural knowledge allows us to distinguish between the (tacit) knowledge that an architect and other stakeholders build up from experience and expertise, and the (explicit) architectural knowledge that is produced and codified – for example in artifacts such as architecture descriptions.

The distinction between application-generic and application-specific architectural knowledge has been proposed by Lago and Avgeriou (2006). This distinction, which is not necessarily applicable to other knowledge than 'architectural' knowledge, allows us to distinguish between (application-generic) knowledge that is "a form of library knowledge" that "can be applied in several applications independently of the domain" and (application-specific) knowledge that involves "all the decisions that were taken during the architecting process of a particular system and the architectural solutions that implemented the decisions". In summary, application-generic knowledge is all knowledge that is independent of the application domain, application-specific knowledge is all knowledge related to a particular system.

Since the two distinctions are orthogonal, a combination of the two results in four main categories of architectural knowledge, which are depicted in Fig. 3.2. That figure

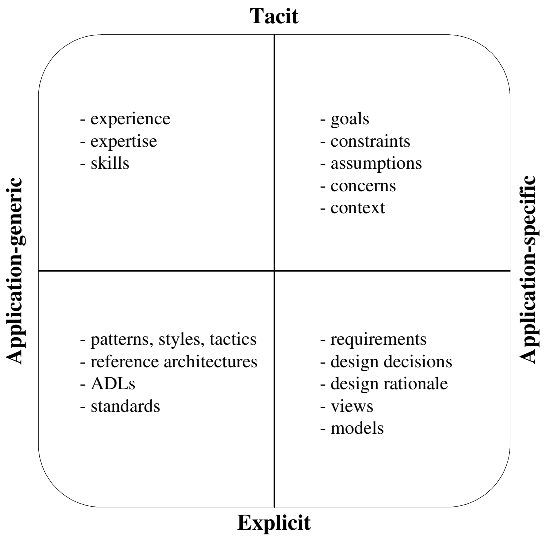


Figure 3.2: Architectural knowledge categories

also provides examples of the type of architectural knowledge that fits each of the four categories.

- Application-generic tacit architectural knowledge includes the design knowledge an architect gained from experience, such as architectural concepts, methodologies, and internalized solutions.
- Application-specific tacit architectural knowledge comprises contextual domain knowledge regarding forces on the eventual architectural solution; it includes business goals, stakeholder concerns, and the application context in general.
- Application-generic explicit knowledge is design knowledge that has been made explicit in discussions, books, standards, and other types of communication. It includes reusable solutions such as patterns, styles and tactics, but also architecture description languages, reference architectures, and process models.
- Application-specific explicit architectural knowledge is probably the most tangible type of architectural knowledge. It includes all externalized knowledge of a particular system, such as architectural views and models, architecturally significant requirements, and codified design decisions and their rationale.

3.3 Philosophies of Architectural Knowledge Management

Knowledge from each of the four architectural knowledge categories can be converted to knowledge in another (or even in the same) category. This conversion lies at the basis of different architectural knowledge management philosophies. For some, architectural knowledge management may be mainly intended to support the transition from application-generic to application-specific knowledge. For others, the interplay between tacit and explicit knowledge may be the essence of architectural knowledge management.

Nonaka and Takeuchi (1995) define four modes of conversion between tacit and explicit knowledge: socialization (tacit to tacit), externalisation (tacit to explicit), internalisation (explicit to tacit), and combination (explicit to explicit). Based on the distinction between application-generic and application-specific knowledge, we can define four additional modes of conversion:

- *Utilization* is the conversion from application-generic knowledge to application-specific knowledge. It is a common operation in the architecting process where background knowledge and experience are applied to the problem at hand.

- *Abstraction* is the conversion from application-specific to application-generic knowledge. In this conversion, architectural knowledge is brought to a higher level of abstraction so that it has value beyond the original application domain.

- *Refinement* is the conversion from application-specific knowledge to application-specific knowledge. Here, the architectural knowledge for a particular application is analyzed and further refined and related.

- *Maturement* is the conversion from application-generic to application-generic knowledge. It signifies the development of the individual architect as well as the architecture field as a whole, i.e., a kind of learning where new generic knowledge is derived and becomes available for application in subsequent design problems.

In total, there are sixteen architectural knowledge conversions. Each conversion is formed by pairing one of the four conversions between tacit and explicit knowledge with one of the four conversions between application-generic and application-specific knowledge. Together, the sixteen conversions form a descriptive framework with which different architectural knowledge management philosophies can be typified.

We saw earlier that the four views on architectural knowledge can all be related to decision making. At the same time, we saw that there are also obvious differences between those views. Those differences are very much related to the different architectural knowledge management philosophies they are based on.

The **pattern-centric** view, for example, is mainly geared towards the development of a shared vocabulary of reusable, abstract solutions. As such, the development of a shared tacit mental model is a major goal. This goal is achieved by sharing application-generic patterns that are mined from application-specific solutions. A second goal is the development of libraries of reusable solutions, which is made possible by maturement of documented patterns by cataloging them. The knowledge management philosophy in this view is primarily based on the following architectural knowledge conversions, which are also visualized in Fig. 3.3:

(a) Abstraction and combination: The most obvious example of this conversion is the process of pattern mining. Patterns are inherently abstractions. They are mined from existing architectural solutions and described in a clear and structured way to improve the reusability.

(b) Utilization and combination: One of the ways in which patterns can be reused in new designs is by looking them up in books, catalogs, or other repositories. Architects who wish to design a system comprised of several independent programs that work cooperatively on a common data structure could look up one of the many books available on architectural patterns (e.g., (Buschmann et al., 1996)) to find out that the 'blackboard' pattern is just what they need. Booch (online) works on the creation of a 'Handbook of Software Architecture', of which the primary goal is "to fill this void in software engineering by codifying the architecture of a large collection of interesting software-intensive systems, presenting them in a manner that exposes their essential patterns and that permits comparisons across domains and architectural styles.".

(c) Maturement and combination: Documented patterns may be organized in pattern catalogs. These catalogs present a collection of relatively independent solutions to common design problems. As more experience is gained using these patterns, developers and authors will increasingly integrate groups of related patterns to form so-called pattern languages (Schmidt et al., 1996). Although each pattern has its merits in isolation, the strength of a pattern language is that it integrates solutions to particular problems in important technical areas. An example is provided by Schmidt and Buschmann (2003) in the context of development of concurrent networked applications. Each design problem in this domain – including issues related to connection management, event handling, and service

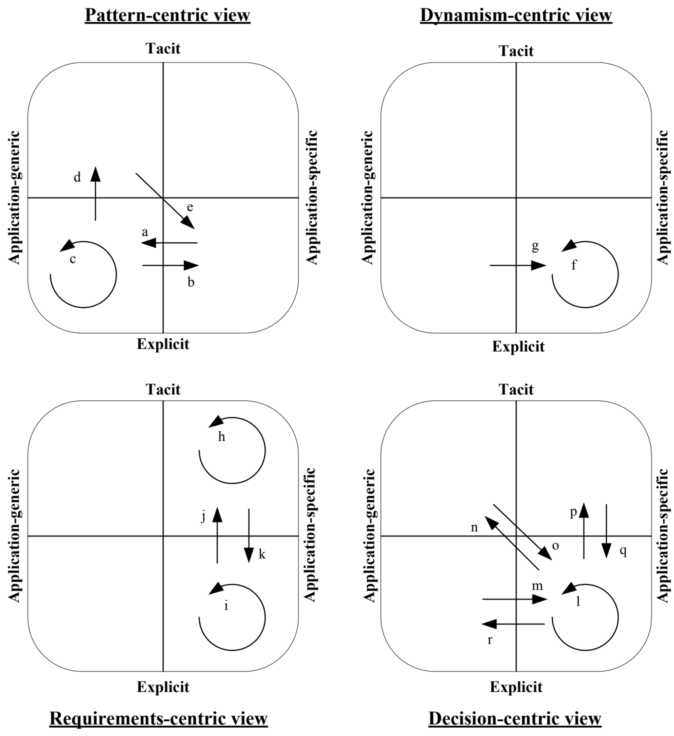


Figure 3.3: Architectural knowledge management philosophies

access – must be resolved coherently and consistently and this is where pattern languages are particularly helpful.

(d) Maturement and internalisation: Experienced architects know a large collection of patterns by heart. Such architects have no trouble discussing designs in terms of proxies, blackboard architectures, or three-tier CORBA-based client-server architectures. Their exposure to and experience with those patterns has led to internalised and matured pattern knowledge. The consequent shared, tacit vocabulary can be employed at will, without the need to look up individual patterns in order to understand what the other party means.

(e) Utilization and externalisation: When an architect has internalized several patterns, the use of those patterns as a means for communication or design is a combination of utilization (of the pattern) and externalisation (of the knowledge about the pattern and its existence).

While the pattern-centric view aims to support the development of tacitly shared application-generic architectural knowledge, the **dynamism-centric** view is focused much more on explicit application-specific architectural knowledge. Although some application-generic knowledge is utilized, the actual reasoning and reconfiguration uses application-specific knowledge. This architectural knowledge management philosophy is therefore primarily based on two conversions:

(f) Refinement and combination: Corresponds to the formal reasoning over codified architectural solutions in terms of models that are consumable by non-human agents. According to Bradbury et al. (2004), in a self-management system all dynamic architectural changes have four steps (initiation of change, selection of architectural transformation, implementation of reconfiguration, and assessment of architecture after reconfiguration) which should all occur within the automated process.

(g) Utilization and combination: corresponds to formally specifying components and connectors in generic languages such as ADLs or other graph representations. Medvidovic and Taylor (2000) propose a comparison and classification framework for ADLs, which enables the identification of key properties of ADLs and shows the strength and weaknesses of these languages. In their comparison, Medvidovic and Taylor point out that at one end of the spectrum some ADLs specifically aim to aid architects in understanding a software system by offering a simple graphical syntax, some semantics and basic analyses of architectural descriptions. At the other end of the spectrum, however, ADLs provide formal syntax and semantics, powerful analysis tools, model checkers, parsers, compilers, code synthesis tools and so on.

For the **requirements-centric** view, the primary goal is tracing knowledge about the problem to knowledge about the solution. In co-development, an additional goal is to connect knowledge about problem and solution so that a stakeholder's image of the problem and solution domain is refined. In this view, the primary architectural knowledge conversions are:

(h) Refinement and socialization: Especially in co-development, stakeholders and architects will interact to align system goals and architectural solutions. Such interaction between 'customer' and 'product developer' is a prime example of a situation in which tacit knowledge is shared (cf. (Nonaka and Takeuchi, 1995)).

Pohl and Sikora (2007) discuss the need for co-development of requirements and architecture. They argue that such co-design is only possible if there is sufficient knowledge about the (course) solution when defining (detailed) system requirements: “instead of defining requirements based on implicit assumptions about the solution, requirements and architectural artifacts need to be developed concurrently”. An example of such an approach is the Twin Peaks model (Nuseibeh, 2001a).

(i) Refinement and combination: Adding and maintaining traceability from requirements to architecture is a refinement step that combines explicit knowledge about elements from the problem and the solution space. Several techniques exist to guide the transition from requirements engineering to software architecture. In their 2006 survey, Galster et al. use architectural knowledge as knowledge about (previous) architectural solutions that can be reused when encountering similar requirements in a new project. They present patterns as a useful container for these reusable assets. Another approach for guiding the transition between requirements and architecture is that of feature-solution graphs by de Bruin and van Vliet (2003). They use architectural knowledge as knowledge about quality concerns (represented as features) and solution fragments at the architectural level (represented as solutions). These are modeled together as Feature-Solution graphs in order to guide the explicit transition (or alignment) between the problem and solution world.

(k) Refinement and externalisation: Externalisation of application-specific tacit knowledge – such as concerns, goals, and the like – is an important part of the requirements process. Externalisation of tacit knowledge (problem-related as well as solution-related) is a precondition for maintaining traceability.

(j) Refinement and internalisation: The interaction between architects and stakeholders is not purely a matter of socialization. In co-development, for example, specifications and design artifacts play a major role as well and may be an aid to let the parties ‘re-experience’ each other’s experiences (cf. (Nonaka and Takeuchi, 1995)). This internalization of explicit application-specific architectural knowledge may consequently lead to ‘new ideas and insights concerning both the envisioned system usage and the architectural solution’ (Pohl and Sikora, 2007).

Finally, the essential philosophy of the **decision-centric** view is externalizing the rationale behind architectural solutions (i.e., ‘the why of the architecture’) that can then be internalized and shape other people’s mental models, so as to prevent knowledge vaporization. The architectural knowledge conversions central to this philosophy are:

(l) Refinement and combination: corresponds to reasoning about codified architectural solutions, which need not be fully automated but may involve decision support. One such decision support approach is introduced by Robbins et al. (1996), who introduce the concept of 'critics'. Critics are active agents that support decision-making by continuously and pessimistically analyzing partial architectures. Each critic checks for the presence of certain conditions in the partial architecture. Critics deliver knowledge to the architect about the implications of, or alternatives to, a design decision. Often, critics simply advise the architect of potential errors or areas needing improvement in the architecture. One could therefore see critics as an automated form of the backlog that architects use in the architecting process to acquire and maintain overview of the problem and solution space (Hofmeister et al., 2007). Another approach that fits this conversion is the Archium tool proposed by Jansen et al. (2007), which is aimed at establishing and maintaining traceability between design decision models and the software architecture design.

(m) Utilization and combination: This conversion amounts to the reuse of codified, generic knowledge (decision templates, architectural guidelines, patterns, etc.) 'to take decisions for a single application and thus construct application-specific knowledge' (Lago and Avgeriou, 2006).

(n) Internalisation and abstraction: Based on experience and expertise, an architect may quickly jump to a 'good' solution for a particular problem. Such a good solution can be a combination of several finer grained design decisions. This combination of design decisions may become so common for the architect that the solution is no longer seen as consisting of individual decisions. It may be hard for the architect to reconstruct why a certain solution fits a particular problem; the architect 'just knows'.

(o) Utilization and externalisation: When a solution has been internalized, and the architect 'just knows' when to apply it, it becomes difficult to see which other solutions are possible. Part of the decision-centric philosophy (e.g., in (de Bruin and van Vliet, 2003)) is therefore to reconstruct and document the constituting design decisions.

(p) Refinement and internalisation: This 'consumption' of architectural knowledge takes place when people want to 'learn from it or carry out some quality assessment' (Lago and Avgeriou, 2006).

(q) Refinement and externalisation: The rationalization of taken architectural decisions, i.e., reconstruction and explanation of the 'why' behind them, is a crucial part of the decision-centric philosophy in which tacit knowledge is made explicit. In this respect, the software architecting can apply the best practices known from the 'older' and well-known field of design rationale. Regli et al. (2000) present a survey of design rationale systems, in which they distinguish between process-oriented and feature-oriented approaches. Process-oriented design rationale systems emphasize the design rationale as a 'history' of the design process, which is descriptive and graph-based. A well-known example is the Issue-Based Information System (IBIS) framework for argumentation. A feature-oriented approach starts from the design space of an artifact, where the rules and knowledge in the specific domain must be considered in design decision making. Often these type of design rationale systems offer support for automated reasoning. A more recent survey on architecture design rationale by Tang et al. (2005) provides information about how practitioners think about, reason with, document and use design rationale. It turns out that although practitioners recognize the importance of codifying design rationale, a lack of appropriate standards and tools to assist them in this process acts as barrier to documenting design rationale. Fortunately, the field of design rationale is working hard on developing more mature support. Recent developments have led to the creation of tooling such as Compendium and SEURAT (Burge and Brown, 2008), and models such as AREL (Tang et al., 2007).

(r) Abstraction and combination: Amounts to the construction of high-level structures (templates, ontologies, models) to capture and store architecture design decisions. Various approaches to codify architectural design decisions in such a way have been reported. Tyree and Akerman (2005) present a template to codify architectural design decisions together with their rationale and several other properties relevant to that decision, such as the associated constraints, a timestamp, and a short description. Kruchten (2004) proposes a more formal approach of codifying design decisions by using an ontology. Ran and Kuusela (1996) present work on design decision trees.

From our discussion on the different philosophies, we can conclude that the differences between the four views on architectural knowledge are very much related to the philosophies they are based on. However, while many single-philosophy approaches for architectural knowledge management exist (e.g., rationale management systems in the decision-centric view, ADLs in the dynamism-centric view, traceability support in the requirements-centric view, or pattern languages in the pattern-centric view), in recent years, a shift towards more overarching approaches can be observed. Many of these overarching approaches seem to have ties to the decision-centric view. This is related to our earlier observation that ‘decisions’ seem to be an umbrella concept for all views on architectural knowledge; there appears to be a movement to a fifth, unified, philosophy which we could call ‘decisions-in-the-large’.

The idea of *‘decision-in-the-large’* is related more to the architecting process than to pure rationale management (which we could call *‘decision-in-the-narrow’*), even though codifying rationale and preventing knowledge vaporization has been one of the prime drivers for the decision-centric philosophy. Obviously, some ‘decision-in-the-large’ approaches evolved from ‘decision-in-the-narrow’ approaches. But a focus on ‘decision-in-the-large’ seems driven more by architectural knowledge use cases than by anything else, often under the concept ‘knowledge sharing’. A classification of such use cases is presented by Lago and Avgeriou (2006), who distinguish between architecting, sharing, assessing, and learning.

3.4 Architectural Knowledge Management Strategies

Most of the proposed approaches to manage architectural knowledge from a decisions perspective (which includes both decisions-in-the-large and decision-in-the-narrow) can broadly be categorized into codification and personalization (Hansen et al., 1999). The codification strategy concentrates on identifying, eliciting and storing knowledge in repositories, which makes that knowledge widely available. This strategy promises to support high-quality, reliable, and speedy reuse of knowledge. The downside is that it usually means separating the knowledge from its creators. The personalization strategy emphasizes the interaction among knowledge workers. In this strategy the knowledge is kept with its creator, who is made known as possessor of the required knowledge. While academic approaches seem to focus on codification, industry practice seems to have a need for personalization as well.

3.4.1 Academic approaches

A number of research initiatives take a decisions-in-the-large architectural knowledge perspective to devise tools and methods to manage architectural knowledge.

DGA DDR Falessi et al. (2006) have proposed a framework that focuses on the reasons why design decisions have been taken. The framework contains a specific design decision rationale documentation technique called DGA DDR, which is driven by the decision goals and design alternatives available. The framework aims not only to document decisions previously taken, but also to support decision makers in taking these decisions. The framework consists of two main activities. In the first activity the project objectives and constraints are defined and it is investigated which decision relationships are appropriate for the project. In the second activity the knowledge is further refined and described in tables.

PAKME Researchers at National ICT Australia (NICTA) have proposed an architectural knowledge management framework, which incorporates concepts from knowledge management, experience factory, and pattern-mining (Ali Babar et al., 2005). This framework consists of various approaches to capture design decisions and contextual information, an approach to distill and document architecturally significant information from patterns, and a data model to characterize architectural constructs, their attributes and relationships. The main objective of the framework is to provide a theoretical underpinning and conceptual guidance to design and implement a repository-based tool support for managing architectural knowledge. A web-based knowledge management tool, called Process-based Architecture Knowledge Management Environment (PAKME), has been developed to support the proposed framework.

GRIFFIN Within the Griffin consortium, we and other researchers have been working on methods, tools, and techniques to manage architectural knowledge. One of the results of this project is a model of architectural knowledge (presented in Chapter 4) which, although later refined to a more generic model, was originally intended as a structure for software architecture project memories. A software architecture project memory stores architectural knowledge, such as the design decisions embodied in the architecture as well as the rationale underlying the design decisions. This allows management of know-why and know-how of software architectures, in addition to know-what already targeted by most existing notational and documentation approaches in software architecture, which typically focus on components and connectors.

ADDSS Two collaborating universities in Spain have proposed to extend traditional architectural view methods, such as the 4+1 view method, with a decision view that allows capturing design decisions in the architecture process (Dueñas and Capilla, 2005). Capilla et al. (2006) have proposed a web-based tool called ADDSS for recording architectural design decisions. In this work a meta-model and a web-based tool are proposed to record, maintain and manage the architectural decisions taken. The tool allows for modeling traces between decisions and supports traceability between design decisions and artifacts such as architecture diagrams.

3.4.2 Industrial practice

In parallel to the academic community, industry has on itself tried to employ – sometimes subconsciously – knowledge management practices to the software architecture process. Below we elaborate on a number of knowledge management initiatives we have encountered in industry that are related to software architecture and architectural knowledge.

DNV DNV is a medium-sized consultancy firm that performs independent software product quality audits for third parties. One of the key architectural knowledge elements that plays a role in such an audit is the set of applicable quality criteria. These criteria are a special kind of architectural design decisions that correspond to the desired architecture of the software product.

Many quality criteria are applicable to multiple projects. Some projects share certain desired quality characteristics for which more or less standard architectural approaches exist. Although over the years some sort of best practices regarding the documentation of quality criteria have grown within DNV, there is no mandatory format or template that is used to express quality criteria. Each project can adopt its own style to document the quality criteria that are used in the audit. Part of these quality criteria is harvested from documentation belonging to previous audit projects.

In the past, DNV has tried to employ a knowledge management approach to better support reuse of quality criteria. Quality criteria were to be made explicit and captured in a knowledge base that could be queried to find applicable quality criteria at the start of a new audit project. Unfortunately, construction of this knowledge base eventually was discontinued because it was hard to capture how quality criteria were related to each other. Part III, especially Chapter 14 and Chapter 15, further elaborates on the management of quality criteria in this organization.

DSTO Defence Science and Technology Organisation (DSTO) is a research and development organization, which provides scientific and technical advice on the acquisition of material to the Australian Defence Organisation. The Airborne Mission Systems (AMS) division of DSTO is responsible for evaluating software architectures for aircraft acquisition projects. AMS is required to understand and organize large amounts of architectural knowledge for a mission system's architecture to support the evaluation process. Currently, the architectural evaluation process mainly relies on the domain knowledge of local experts.

AMS's technical leadership became increasingly interested in building its capabilities in systematically evaluating system and software architectures and managing architectural knowledge for aircraft mission systems. Hence, AMS decided to improve its architectural evaluation practices by codifying and reusing an architecture evaluation process, architectural knowledge, and contextual knowledge.

RFA RFA is a large Dutch organization that develops and maintains software systems. These systems are typically critical for the public, large in size and complexity, and long lasting. We investigated and assessed the organization's current mechanisms for sharing architectural knowledge.

RFA has acknowledged the need to support sharing and reuse of architectural knowledge. To this end, the organization has deployed a knowledge repository in which commonly used architectural knowledge is stored and made reusable in the form of questions and answers. At the start of a project, architects use the tool to create a very first version of the architecture description. The tool will provide the architects with the stored questions. The architects' answers – the initial architectural design decisions for the project – are then structured and reflected in a text that provides the basis for the remainder of the architecting process. In Part II we report in more depth on the challenges RFA encountered with respect to sharing architectural knowledge.

VCL VCL is a multi-national software development organization. Development teams within this organization are located at multiple sites spread throughout the globe. Architectural knowledge is shared in the form of 'architectural rules'; architectural decisions that need to be complied with throughout the organization. A study within this organization focused on dissemination of and compliance with architectural rules in this multi-site environment (Clerc et al., 2007b). Dissemination of architectural rules takes place by means of small, text-based documents; so-called architectural notes, or *archnotes*.

3.4.3 A comparison with knowledge management theory

Knowledge management is a large interdisciplinary field, and has as such fostered a number of different approaches. Earl (2001) proposes different 'schools' of knowledge management, broadly characterized as the *technocratic*, the *economic*, and the *behavioral* school. Hansen et al. (1999) divide between two strategies: codification and personalization. Codification refers to organizations aiming their strategy on codifying knowledge and making it easily available for anyone through so-called knowledge repositories (Liebowitz and Beckman, 1998). Knowledge from individuals or groups can be codified (or acquired) through a number of means, such as interviews, questionnaires and architecture reviews. Personalization, on the other hand, focuses on helping people communicate knowledge, instead of storing it. This can be facilitated through 'yellow-page' indexes of experts or skills management systems, or a focus on company processes to share knowledge such as postmortem reviews. Alternative approaches are to focus on collaborative work such as pair programming, to design informal meeting-spaces or to use open-plan offices.

In software engineering, most research and most industry practice has been associated with codification (Dingsøyr and Conradi, 2002); personalization has been given less attention in knowledge management initiatives. This observation seems to hold true for software architecture research and architectural knowledge management as well.

The research projects listed in §3.4.1 all exhibit to a large extent a technocratic focus on codification of architectural knowledge, largely neglecting economic and behavioral aspects. Both ADDSS and DGA DDR aim to record and subsequently maintain architectural design decisions. Researchers from NICTA mention both codification and personalization as ways to manage knowledge; nevertheless, the focus of their framework is on mining and capturing (i.e., codifying) implicit knowledge in the form of architectural patterns. The Griffin project reports on their efforts to construct software architecture project memories that contain codified architectural knowledge in the form of architectural design decisions and related entities.

Industry also seems to favor an explicit choice for codification of architectural knowledge, although in practice some personalization aspects are also present. However, none of the organizations we observed seems to have intentionally made a choice for the latter.

The strategy chosen for quality criteria reuse in DNV is a classic example of a codification strategy. DSTO's goal to organize architectural knowledge denotes a focus on codification as well, and RFA has clearly chosen a (centralized) codification approach for architectural knowledge management. Personalization does not play a significant role in the chosen architectural knowledge management strategy, although interviews with employees of RFA brought to light that much architectural knowledge sharing takes place through formal and informal meetings. Important (implicit) knowledge within this organization therefore includes who knows what and which architectural knowledge can be found where.

In VCL, the use of archnotes denotes an explicit choice of the organization for a codification strategy for architectural knowledge distribution. This is probably not a surprising choice, since the geographic dispersion of the development teams to a large extent inhibits colloquial knowledge sharing. Nevertheless, to our surprise one of the most important 'archnotes' (in terms of usage frequency as well as utility in daily practice) turned out to be a 'yellow-pages'-like document that contains names, locations, and phone numbers of key personnel involved in the project. The users of the archnotes system seem to have exploited the lack of mandatory structure of the archnotes to introduce elements of a personalization strategy in the system. Although the organization explicitly chose to follow a codification strategy, there is apparently an implicit undertone that personalization is very important as well. However, VCL has never explicitly chosen to follow (or reject) a personalization strategy.

In summary, current (reported) architectural knowledge management approaches have a tendency to heavily rely on codification. This is true both in research and in industry. However, industry practice clearly shows that personalization is important as well. Different organizations seem to apply personalization aspects to management of architectural knowledge. Nevertheless, whenever personalization is encountered, this is more the result of an implicit choice rather than an explicit preference.

3.4.4 Intentional codification vs. unintentional personalization

The main argument for choosing a codification strategy has been that you can invest once in knowledge assets, and reuse them many times (Hansen et al., 1999). A common critique of the codification strategy is that it may create 'information junkyards'. McDermott (1999) claims that "if people working in a group don't already share knowledge, don't already have plenty of contact, don't already understand what insights and information will be useful to each other, information technology is not likely to create it". In addition, Swan et al. (1999) criticize the knowledge management field for being too preoccupied with tools and techniques. They claim that researchers tend to overstate the codifiability of knowledge and to overemphasize the utility of IT to give organizational performance improvement. They also warn that "codification of tacit knowledge into formal systems may generate its own pathology: the informal and locally situated practices that allow the firm to cope with uncertainty, may become rigidified by the system".

One can also question the strong focus on codification when it comes to managing knowledge related to software architectures. Frequent technological changes make up a force which makes knowledge reuse difficult in the software engineering field. For knowledge that is not to be reused many times, personalization is inexpensive compared to codification.

Then again, the high prevalence of codification in architectural knowledge management might be the result of a number of distinctive characteristics of the software architecture field. First, software architects are used to codify knowledge, through work with modeling techniques and through identifying architectural patterns. Second, software architects are very mature users of information technology, and should be able to use technical tools more efficiently than employees in other domains. Third, architectural knowledge is the earliest design knowledge, which can be expected to be used and revisited often throughout the whole software development life cycle. With a high level of knowledge reuse, the cost of codification may be outweighed by its benefits. These characteristics of the software architecture field seem to favor a codification strategy for managing architectural knowledge.

But in spite of a favor for using a codification strategy in software architecture, in the software engineering field there are very few reports of organizations that have opted for a codification strategy where the developed knowledge repositories have been used to a large extent over time (Dingsøyr and Røyrvik, 2003). The apparent issues with and discontinuation of various codification efforts in industry reported in §3.4.2 further illustrate the difficulty of sustained exploitation of knowledge repositories. So the question is: is codification really *the* answer?

Kankanhalli et al. (2005) have developed a model to explain the use of knowledge repositories, a central element in the codification strategy. Their study indicates that social incentives are important in order to increase knowledge repository usage. Moreover, some knowledge is difficult or impossible to codify, and there can be problems interpreting knowledge if sufficient context information is not codified as well. A personalization strategy can stimulate discussion within a company and may be more appropriate in certain situations.

Our investigation of current approaches in architectural knowledge management suggests that there is an ongoing trend in industry of increasing awareness of the need for a strategic choice for architectural knowledge management and, simultaneously, increasing efforts to express such knowledge. This results in a movement from unintentional personalization (UP) - used by most organizations - to intentional codification (IC). This situation is graphically depicted in Fig. 3.4.

However, the reported IC initiatives from industry all seem to encounter their own problems. Current research tries to alleviate such problems by inventing structured approaches to architectural knowledge management, using an IC strategy. In the meantime, both industry and research seem to ignore (or forget about) the existence of intentional personalization (IP).

Codification has the potential to contribute to better management of architectural knowledge, given that the research field takes into consideration how important social aspects are to get such systems into use. There are good reasons for many organizations to opt for a personalization approach in order to facilitate innovative solutions with minimal bureaucracy. However, choosing between codification and personalization need not be a black or white choice. A combination of the two strategies, what Desouza et al. (2006) call a hybrid approach, probably suits typical architecting activities best.

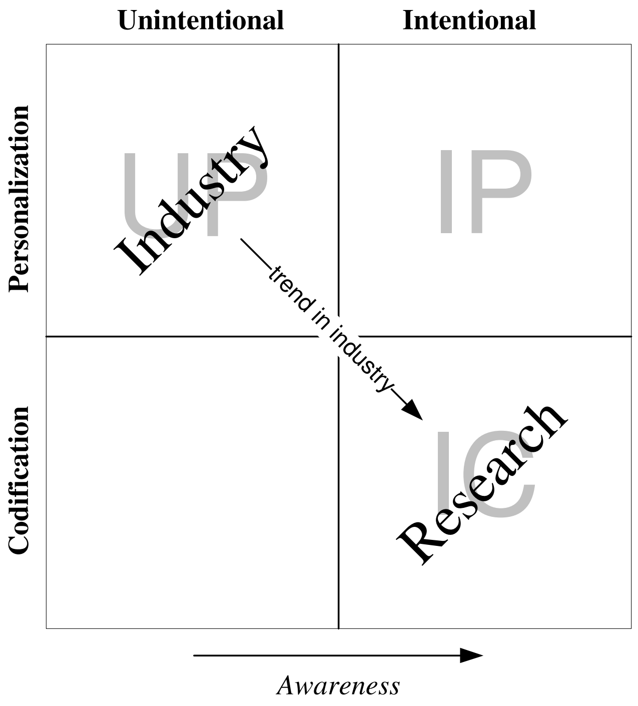


Figure 3.4: Architectural knowledge management strategies in research and industry

3.5 Justification

Our overview of the state of the art in architectural knowledge management is based on an extensive literature review that consisted of three stages. In the first stage (completed early 2007), we performed an ad-hoc, preliminary review of architectural knowledge management strategies. At that point in time, architectural knowledge was often still equated to 'architectural design decisions'; consequently we focused our analysis on approaches that had what we would now call either a 'decisions-in-the-narrow' or 'decisions-in-the-large' philosophy of architectural knowledge management.

In the second stage (completed early 2008), based on a predefined protocol we identified, selected, and synthesized all studies that define or discuss 'architectural knowledge'. In the third stage (completed late 2008), we employed social network analysis to identify the communities in which architectural knowledge plays a role.

The results of our review have been presented in a logical, rather than chronological, order in this chapter. In particular, the results of the 2007 studies are mostly to be found in §3.4. This made it possible to first discuss what architectural knowledge is, followed by a reflection on how to manage it.

In the remainder of this section, we further detail the systematic steps we followed in the second stage (§3.5.1 and §3.5.2) and the third stage (§3.5.3) of our review.

3.5.1 Systematic review protocol

Our discussion in this chapter is a typical example of secondary research (or 'desk research') in which the results of different primary studies are combined through some form of meta-analysis. For the primary studies, we rely only on studies that have been published in the literature.

In a literature review such as ours, there are two important considerations: how to select the primary studies, and how to perform the required meta-analysis. We have chosen to perform a systematic review for selecting studies and to use a meta-ethnographic approach as part of that systematic review for synthesizing the gathered data.

Studies selection

There are basically two ways of performing a literature review: (1) an ad hoc review, and (2) a systematic review. The main difference between the two is the formality and a priori planning of the systematic approach. In a systematic review, a protocol is defined that specifies the research questions to be answered, as well as the manner in which data will be gathered. This contrasts with the ad hoc manner of conventional literature reviews. Systematic reviews are a heavily used instrument in (evidence-based) medicine. Work by various authors (e.g., (Biolchini et al., 2005; Dybå et al., 2007; Kitchenham, 2004)) has led to guidelines for applying systematic reviews to the domain of software engineering as part of the evidence-based software engineering paradigm.

Biolchini et al. (2005) propose a protocol template for systematic reviews in software engineering. Their template combines review protocols from the medical field with earlier work on evidence based research and systematic reviews in software engineering. We have used Biolchini's template to develop a protocol for our systematic review of definitions of 'architectural knowledge'. The main elements of this protocol are given below.

- **Problem.** The problem we address is that there appears to be no commonly accepted definition of what architectural knowledge entails. This makes it a fuzzy concept, and makes the current understanding of what architectural knowledge entails unclear. This may evidently introduce misunderstandings when different authors use different definitions for the same concept.

- **Research questions.** Particular research question our systematic review should answer are:
  - 1. What are the different definitions of 'architectural knowledge', and how are they related?
  - 2. What concepts are deemed related to 'architectural knowledge'?
- **Sources list.** Since we are interested in definitions of architectural knowledge in the context of software-intensive systems, we include the major publishers of and indexes that contain software engineering related publications:
  - ACM Digital Library
  - IEEE Xplore
  - ISI Web of Science
  - SpringerLink
  - ScienceDirect
  - Wiley Inter Science Journal Finder
- **Search string.** We are interested in architectural knowledge in the context of software-intensive systems (and not in the context of, for example, civil architecture). Furthermore, we are aware that some authors prefer to use the phrase 'architecture knowledge' over 'architectural knowledge'. We assume that a definition of 'architectural knowledge' is not present in a study if the phrase itself is not present. We therefore want to search for publications that contain either the word 'software' or 'system' and the phrase 'architecture knowledge' or 'architectural knowledge'. Therefore, the search string we use is:

  > ('architectural knowledge' OR 'architecture knowledge') AND ('software' OR 'system')

- **Studies inclusion/exclusion criteria** Studies to be included in our review:
  - Must have 'architecture' as topic (and not, for instance, present an architecture for a particular system)
  - Must be about architecture of software-intensive systems (and not about, for instance, civil architecture)

- Must have ‘real’ content (hence not a TOC, cover page, advertisement, etc.)
- Must be in English
- Must discuss ‘architectural knowledge’

Data synthesis

A potential problem with common meta-analytic approaches in systematic reviews is their focus on integration of quantitative data. Dybå et al. (2007) address the limited applicability of this type of meta-analysis to software engineering research.

Our review in particular is not very well suited for quantitative techniques, since the primary studies we address can be expected to be mostly grounded in qualitative research. Instead of a quantitative approach we therefore need a qualitative approach to data synthesis. Based on the experiences from Dybå et al., we chose meta-ethnography as the approach to synthesize the selected studies.

Meta-ethnography was proposed by Noblit and Hare (1988) as a form of interpretive synthesis. Meta-ethnography relies on the translation of key concepts and metaphors of different studies into one another, for which there are three strategies. In reciprocal translational analysis, concepts from different studies are directly translated into each other. In refutational analysis, contradictions between studies are characterized. In lines-of-argument synthesis, a general interpretation is grounded in the findings of separate studies. For our review, we performed reciprocal translational analysis of the concepts that different authors use in defining and characterizing ‘architectural knowledge’.

3.5.2 Systematic review execution

The execution of our research commenced early January 2008. As a consequence, our review includes studies that were published and/or indexed before that date. We cannot be certain that we have covered all studies with a publication date in 2007, since especially the studies from Q4 may not have been indexed yet at the time we conducted our review.

Studies selection

Since one cannot expect a definition of ‘architectural knowledge’ always to be found in the abstract of a study, we run the risk of missing relevant studies when we limit our search to abstracts only. We therefore conducted a full-text search in the six identified sources, which resulted in a list of 751 studies that matched the search string.

First iteration: Scanning titles We evaluated compliance of the studies found with the inclusion criteria in four iterations. In the first iteration, we read only the titles of the identified studies and indicated (independently of each other) whether we considered the study out of the scope of our review, i.e., we could indicate at least one inclusion criterion for which it was clear from the study's title that it would not be met. In this phase, only if we both agreed on at least one exclusion criterion the study was excluded from our review. If, for example, we both determined from the title of the study that it was not in English we would exclude it. If, however, one of us found for instance that the study was not about architecture and the other concluded that the study was in fact about architecture but not about architecture of software-intensive systems, the study would remain in the set of studies to be analyzed in the second iteration.

Second iteration: Reading abstracts At the start of the second iteration we had 396 studies left, which means that we were able to exclude more than 45% of the studies by reading the title alone. In this iteration we followed the same procedure as in the first iteration, but instead of reading only the titles of the studies we now read the abstracts as well. After this iteration we were able to eliminate another 140 studies, which left us with 256 studies.

Third iteration: Scanning full-text In the third iteration we did a full-text scan of the studies. Based on this scan we again assessed the studies' compliance with our inclusion criteria. Because of the amount of work involved, half of the 256 studies was independently assessed by each researcher. In this third iteration, we particularly focused on the phrases 'architectural knowledge' and 'architecture knowledge', which were part of the search string and therefore must be present in the text. Apart from various studies that were out of scope because they discussed for example the field of civil architecture, we found that a number of studies did not contain the phrase 'architectural knowledge' as we had intended it. For example, some studies only referred to other studies that happened to have 'architectural knowledge' in their title, or the full-text query that we executed had registered a match on only the keywords (e.g., 'software architecture; knowledge engineering'). Both researchers gathered evidence from the full text of the studies to support their assessment of in- or exclusion. This evidence (mainly literal quotes from the full text) was thereafter discussed and only when both researchers agreed that the study did not meet the inclusion criteria it was excluded. Whenever necessary, the full text of the study itself was consulted during this discussion. In this way, we were able to exclude another 80 studies, leaving us with 158 studies.

Fourth iteration: Reading full-text In the fourth, and final, iteration the full text of the studies was not scanned, but thoroughly read. It was only in this phase that we could really assess whether a study did or did not discuss ‘architectural knowledge’. In some studies, while the study did use the phrase ‘architectural knowledge’ it remained completely unclear what the authors considered to be architectural knowledge. For example, a study could talk about ‘architectural knowledge present in the organization’ without the definition or scope of such architectural knowledge further being discussed in the text. Those studies were excluded from the review. After this iteration we were left with a total of 116 studies in which the authors talk about architectural knowledge in more or less concrete terms, i.e., the authors either provide a clear definition, or discuss several concepts related to ‘architectural knowledge’. As it turns out, the latter is much more common. Quite a lot of studies address the notion of architectural knowledge, but many of them refrain from explicitly stating what architectural knowledge entails. Out of the 116 publications we found, only 14 studies define what the authors consider architectual knowledge.

Data synthesis

The high number of studies (116) that matched our inclusion criteria made it infeasible to translate all studies found into each other. Since we were particularly interested in how architectural knowledge is defined, we decided to limit the meta-ethnographical synthesis to the 14 studies that provide a definition of architectural knowledge. To these studies, we applied reciprocal translational analysis in two ways.

We first focused on the concepts explicitly used in the different definitions. By translating those concepts into each other, we were able to identify four areas from which – according to the 14 definitions – architectural knowledge originates: problem domain, design decisions, solution fragments, implementation, and systems design.

3.5.3 Social network analysis

Since we are interested in the community structures that underly the topic of architectural knowledge, we enriched our dataset with bibliographical links. We assume the community structure can be found, or approximated, by taking into account the bibliographical references that various authors make to each others work. Strong communities will display many intra-community references, and relatively few references to work outside the community.

In the systematic literature review of studies that define or discuss ‘architectural knowledge’, we identified 116 such studies. These 116 studies form the basis for our discussion in this chapter. Consequently we will refer to these studies as the set of core studies (C).

There are two types of bibliographical references: references *from* studies in C to other studies, and references *to* studies in C from other studies. We will use B (for 'bibliography') to denote studies that are referred to *from* studies in C, and R (for 'referring') to denote studies that refer *to* studies in C. Hence, the mapping $R \to C \to B$ summarizes the enriched data set used throughout this chapter.[1]

To determine B, we simply took all references listed in the 116 studies from C. To determine R, however, we had to do a 'reverse search' on the studies in C, since the information needed is not present in C itself. For construction of R, we used the Google Scholar search engine which provides such a reverse search facility through its 'cited by' feature. While constructing R and B, we also obtained results not necessarily found in sources from the sources list identified in our systematic review. We do not see this as a limitation of our methodology. The goal of the dataset enrichment is different from the initial identification of primary studies in the systematic review; in the enrichment we are interested in community structures, and the different communities need not be limited to studies published through and indexed by the sources used for the review.

Together, B, C, and R comprise a 'social network' of scholarly publications and their interconnections. We analyzed this network using the Girvan-Newman algorithm (Girvan and Newman, 2002). The Girvan-Newman algorithm discovers communities in a graph, based on the 'betweenness' of edges, i.e., the number of shortest paths that run through an edge. The algorithm iteratively calculates the edge betweenness and removes the edge with the highest betweenness value, eliminating edges that act as connections between different communities. Eventually, the algorithm results in a fully disconnected graph of 'communities' that consist of a single node.

Obviously, a disconnected graph is not the strongest community structure possible. Newman and Girvan (2004) define the modularity (Q) as a measure of strength of the community structure discovered in a network. High values of Q ($0 \leq Q \leq 1$) indicate networks with a strong community structure. Their empirical evidence shows that local maxima of Q correspond to reasonable community divisions, hence it is good practice to stop the Girvan-Newman algorithm when a (local or global) maximum Q-value has been obtained. In our case, the first local maximum of Q occurred when the graph had been split up in 52 communities, while the global maximum occurred at 59 communities ($Q \approx 0,7832$), which is extremely high (according to Newman and Girvan, values above 0.7 are rare; typical values fall in the range $0.3 - 0.7$). Because of the data enrichment process we followed and the way the Girvan-Newman algorithm

[1]Note that B, C, and R are not completely disjoint; there are several occurrences of studies from C referring to other studies from C. Also, publications that are referred to from one study in C may themselves refer to other (earlier) studies from C.

works, each of these 59 communities consists of at least one study from C, plus zero or more publications from either B or R.

In order to assign meaning to the 59 communities that came out of the algorithm we examined the set of studies for each of these communities in turn, and gave them a label that corresponded best to the studies in that community. Often, the non-core studies in the community did help in further characterizing the community and helped in phrasing a suitable label. When this was more difficult (for example when the non-core studies varied too much in subject), we looked more specifically at the core studies in that community since these actually talk about architectural knowledge and can therefore lead to more fitting for the community name. In the end we ended up with 59 labels for the communities, although some overlapped to a certain extent.

In order to find out how exactly the communities had been discovered by the Girvan Newman algorithm we examined the hierarchical structure of the identified communities. The hierarchical relations capture the order in which the communities have been identified. Based on the order of community-split-ups we could assign names to larger-order (i.e., parent) communities as well.

According to our definition a community should consist of at least two core studies (that talk about architectural knowledge) written by different authors. Based on this rule we further analyzed the data. We limited the number of main communities by removing the single-core-study ones and the ones consisting of merely studies by the same author(s). In the end this refinement culminated into the four main communities discussed throughout this chapter: pattern-centric, dynamism-centric, requirements-centric, and decision-centric.

3.6 Conclusions

Conducting a systematic review is obviously more enduring and time-consuming than a more ad hoc overview of related work, but we feel it is well worth the extra effort. By systematically scrutinizing available literature, we obtained an understanding of the domains in which the concept is used, the different definitions of architectural knowledge, and how these are related. A potential and obvious drawback of this approach is the reliance on the search string used. Communities that discuss topics related to architectural knowledge but never use that phrase itself are per definition not included in our analysis. However, given our goal to investigate current definitions of architectural knowledge we do not see the exclusion of communities that do not use the literal phrase – let alone define it – as a shortcoming.

Based on the number of publications that discuss architectural knowledge, we see an increasing interest in the subject matter. However, only a small percentage of those studies (14 out of 116 publications) propose or refer to a definition of architectural knowledge. Most of those definitions focus on design decisions, although some refer to solution fragments or elements from the problem domain as well.

We found two surprising definitions of architectural knowledge. One surprise was a definition of architectural knowledge often used in the broad area of systems design, applied to the development of software intensive systems by two authors. This definition is orthogonal to the definitions of architectural knowledge in software development, but at the same time seems related given its application to characterize different software development projects by one of the authors. The other surprise was a definition of architectural knowledge coined in 1996, remarkably similar to recently proposed definitions that focus on design decisions. Between 1996 and 2005, no definitions or references thereto were published in the literature. It seems as if this early definition was forgotten and almost reinvented after nearly a decade.

We were somewhat disappointed by the low number of actual definitions (or references thereto) in the current literature. We attribute this low number of definitions to the relatively recent focus on architectural knowledge. The fact that 7 out of the 14 definitions come from studies that were published in 2007 leads us to believe that we will see more definitions being coined and discussed in the near future. We are unsure whether the perfect definition of architectural knowledge that everyone agrees upon will ever be found, but we do want to urge researchers to be precise and concrete in defining the concepts they consider part of architectural knowledge. In this way, ambiguity can be prevented and the community as a whole can work toward a better common understanding of the scope and span of ‘architectural knowledge’.

When it comes to management of architectural knowledge, both research and industry have an intentional focus on the codification strategy. However, seeing that most intentional codification efforts in industry suffer from various shortcomings, organizations seem to rely on unintentional personalization as the primary architectural knowledge management strategy. This suggests that there is a gap between what researchers are working on and the practice in the industry. We think both personalization and codification serve purposes for managing architectural knowledge, and both strategies as well as hybrid combinations should be investigated in more detail in the future. Furthermore, the industry should be aware of the implicit choices they make in architectural knowledge management, and strive for an explicit, intentional, choice of strategy.

We expect the interest in architectural knowledge to keep increasing over the coming years. Although it is unlikely that we will see a fully unified view on architectural knowledge anytime soon, the observed trends in architectural knowledge management indicate that future developments on managing architectural knowledge may further align the different views. We expect the trend towards ‘decision-in-the-large’ to continue, since both researchers and practitioners aim for a better understanding of what architects do, what their knowledge needs are, and how this architectural knowledge can best be managed in the architecting process. It seems likely that this trend continues in the direction of hybrid strategies so that the industrial need for a personalization strategy is also intentionally supported.

4 A Core Model of Architectural Knowledge

In the previous chapter we saw that there is not a single view on architectural knowledge. In this chapter, we present a core model of architectural knowledge that unifies existing, decision-centric models and provides a common frame of reference regarding what architectural knowledge entails. Practical applications of the core model include inter-organizational and intra-organizational exchange of architectural knowledge.

4.1 Introduction

Various authors address the notion of 'architectural knowledge' and provide models or definitions of what this notion entails. Key elements in all models are design decisions and their rationale (see Chapter 3). However, different authors use different words for what might be the same. For example, some models consider design decisions, others architectural decisions, but it is hard to determine whether these actually denote the same concept.

Having different notions of what architectural knowledge entails can hamper effective management of that knowledge. If, for instance, different organizations – or even departments within a single organization – use different concepts to communicate architectural knowledge, terminological misunderstandings may arise. It then becomes very hard, if not impossible, for these parties to exchange architectural knowledge. To enable architectural knowledge exchange, we need a model of architectural knowledge that acts as a common frame of reference.

In this chapter, we propose a reference model of architectural knowledge that has maximal expressivity in the architectural knowledge domain. Other models of what architectural knowledge entails can be expressed in the form of extensions to this model. These extensions may be domain-specific, organization-specific, or both.

4.2 Research Methodology

The structure of this chapter, schematically depicted in Fig. 4.1, is tightly coupled to the research approach we followed. In the remainder of this section we outline the research methodology we employed, the steps we followed, and in which sections the respective results are elaborated upon.

The approach we followed can best be described as an instantiation of action research. Action research is an iterative research approach in which the researcher actively participates in the studies he performs. The researcher wants 'to try out a theory with practitioners in real situations, gain feedback from this experience, modify the theory as a result of this feedback, and try again' (Avison et al., 1999).

'Action research' is not a single well-defined method; many different forms of action research exist. Canonical action research is "one of the more widely practiced and reported forms of action research in the information systems literature" (Davison et al., 2004). Canonical action research (CAR) is based on the five stage cyclic model presented in (Susman and Evered, 1978), which consists of the stages diagnosis (identifying a problem), action planning (considering alternative courses of action), action taking (selecting a course of action), evaluating (studying the consequences of an action), and specifying learning (identifying general findings). Although it is difficult to concretely distinguish the individual stages, the jest of the CAR cycle is clearly present in our approach.

Our research commenced with designing a 'theory' of architectural knowledge described in §4.3.1. This initial model of architectural knowledge is described in §4.3.2. We then experimented with the model, and tried to characterize the use of architectural knowledge by (and together with) four industrial partners: RFA (a large software development organization), VCL (a multi-site consumer electronics manufacturer), DNV (an SME involved in performing independent software product audits), and PAV (a scientific institute). Experience from the characterization attempts taught us that there were a number of mismatches between our theory and industrial practice. Reflection on the apparent mismatches led us to conclude that we should strive for a model of architectural knowledge that is both minimalistic and complete.

In order to accommodate for the desired properties, we refined the initial model. We hence arrived at a new 'version' of our theory of architectural knowledge, which we present in §4.4. Due to the properties of this new model, we refer to it as a 'core model' of architectural knowledge.

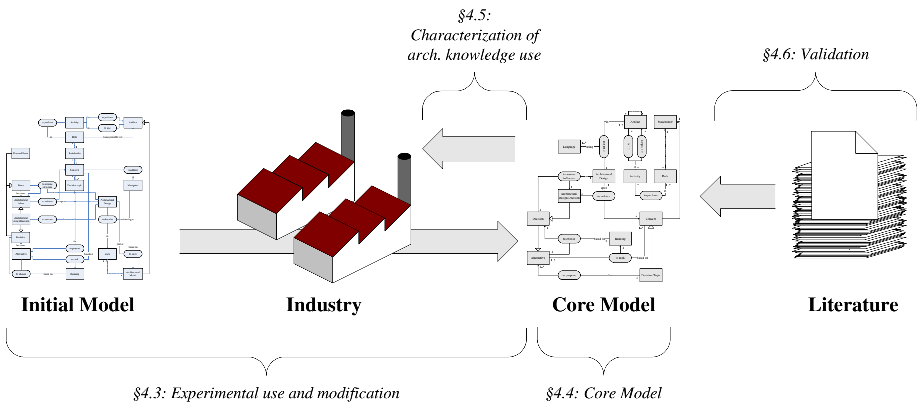

§4.5: Characterization of arch. knowledge use
§4.6: Validation
Initial Model
Industry
Core Model
Literature
§4.3: Experimental use and modification
§4.4: Core Model

Figure 4.1: Structure of this chapter

With the mismatches between theory and practice removed, we could successfully employ our core model to characterize the use of architectural knowledge by the four partners. This characterization, which is the subject of §4.5, led to a number of hypotheses regarding the probable cause of problems with architectural knowledge management in the collaborating organizations.

Since we want our model to be useful as a reference model to align different architectural vocabularies, we regard our core model as 'valid' when concepts from different architectural approaches can be expressed using terms from the model. Unfortunately, it is impossible to strictly prove that our model is valid in this sense. However, we can make the validity of the core model plausible by trying to falsify our model by comparing the model with (accepted) literature. This falsification attempt is discussed in §4.6.

4.3 An Initial Theory of Architectural Knowledge

To construct our initial theory of architectural knowledge, we used the FRS method, a domain modeling method originally developed within Philips Research (Philips, 2004) as an extension of the KISS method (Kristen, 1995). In §4.3.1, we present how we used this method. Since the method targets both domain objects and domain actions, the resulting model (presented in §4.3.2) captures both static and dynamic aspects of the architecting process, as well as the relations between them.

The basic stages for constructing a domain model are knowledge elicitation, knowledge structuring and knowledge specification. For elicitation, the FRS method relies on natural language; for structuring it uses packages, objects, and actions; and for specification it uses an object-interaction model.

Elicitation: Natural language Natural language text can be acquired in numerous ways, e.g., by analyzing (transcriptions of) interviews with domain experts, through scenario authoring, or through brainstorm sessions. Written documents can also be taken into account. Grammatical analysis of the acquired text provides a way of identifying candidate concepts. This analysis uses the sentences that express either some activity or a relation, such as specialization, composition, or aggregation. These sentences show how domain objects are changed, and which static relations exist between them. The direct object and indirect object(s) of a sentence are candidate objects in the domain. The verb(s) correspond to candidate actions when they indicate a change for the associated objects. Candidate actors are the subjects of the sentences.

Given for instance the sentence 'An architect discusses requirements with the customer, and they agree on a specification.', the candidate actors are 'architect' and 'they', the candidate objects are 'requirements', 'customer', and 'specification', and the candidate actions are 'to discuss' and 'to agree'.

Structuring: Packages, objects, and actions The FRS method applies knowledge structuring at two levels: structuring the domain itself in 'packages', and structuring the knowledge contained in each individual package. Packages provide a general grouping mechanism. By grouping domain concepts, potentially large and complex models can be developed and managed more effectively. The packages divide the modeling efforts into coherent subdomains with a limited scope.

One of the advantages of using packages is that it provides a means to 'divide-and-conquer'; it is often easier to focus on the contents of a single package instead of on the whole domain. Packages also help in disambiguation of terminology, since they provide a 'namespace' for their contents.

Structuring a domain in packages has another benefit. It can also aid in assessing the model quality. By viewing each domain concept as 'owned' by a single package, even if it also used in other packages, a dependency between packages emerges. This dependency can be further analyzed. For instance, cyclic dependencies might indicate a need to reorganize package contents.

Within each of the packages, the elicited knowledge is structured in more detail. This is done by identifying domain objects and domain actions from the set of candidates. Domain objects represent the static concepts in a domain. Domain actions change the state of one or more domain objects, and reflect the dynamic concepts in the domain. Domain objects and actions are related by a participation relation, i.e., objects participate in an action.

To guide the knowledge structuring process, the FRS method defines the following modeling rules with respect to the introduction and elimination of domain concepts.

- A domain object must have a unique identity, and may only be introduced when it undergoes actions in the domain that are specific to that object.
- A specialization relation may only be introduced when two or more objects share a participation relation. A good indication of a specialization relation is a juxtaposition of nouns, e.g., a 'design artifact' is an 'artifact'.
- Every object must have an associated instantiating action. Moreover, the objects should be unique, which means that synonyms from the elicitation stage should be merged. Homonyms within the same package, or namespace, should be renamed to provide unambiguous terminology.

Specification: Object-Interaction model The FRS method combines static and dynamic aspects of a domain at the class level. For this reason it prescribes the use of an object-interaction model, which captures the possible objects and actions in the domain.

An FRS object-interaction model[1] contains constructs familiar from UML, a modeling notation often used in software engineering. UML contains constructs to describe static aspects of a domain, e.g., through class diagrams. It can also describe dynamic aspects, e.g., through activity diagrams. UML, however, does not allow the combination of these two aspects in a single diagram at the class level.

Unlike standard UML, the FRS object-interaction model allows the combination of static and dynamic aspects. While domain objects are represented as UML classes, domain actions are depicted as UML classes of an additional stereotype 'domain action', represented by the standard UML activity symbol. Participation relations are drawn as UML associations between domain objects and actions. The association name corresponds to the role of the object in the action. Specialization relationships are represented as standard UML generalizations. In addition to these standard UML constructs, instantiating participations are represented by an arrow at the end of the instantiated object.

The object-interaction model corresponding to the sentence 'An architect discusses requirements with the customer, and they agree on a specification.' is depicted in Fig. 4.2.

[1]UML also has the notion of an object-interaction diagram, which has a different connotation, though.

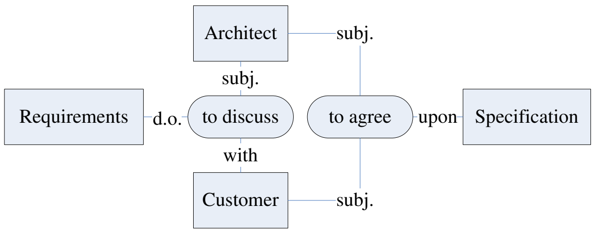


Figure 4.2: Example object-interaction model

When validation of the resulting object-interaction model indicates missing information, the model may be further refined. Validation and refinement are aided by the application of the rules that are listed in the previous section. This refinement is another iteration of knowledge structuring, which can be preceded by further knowledge elicitation when necessary.

4.3.1 Design

In this section, we outline how we modeled the architectural knowledge universe of discourse using the FRS method. A schematic overview of the process we followed is depicted in Fig. 4.3 and elaborated upon below. The abbreviations ‘E’, ‘St’, and ‘Sp’ used below denote elicitation, structuring, and specification of knowledge respectively.

Package identification [St]

We used the package construct to split the architectural knowledge domain into various subdomains. Each package covers part of the domain.

A schematic outline of the architecting process is depicted in Fig. 4.4. This figure shows the stakeholders as being central to the architecting process. This corresponds to the notion that a software architecture should be stakeholder-driven (cf. e.g., (Bass et al., 2003)). The stakeholders, in particular the architects, take decisions according to their roles. These decisions can be influenced by environmental events, i.e., events that are relevant but in principle do not belong to the software architecting and development realm. This includes technical, business, organizational, political, and economic events such as the establishment of budget constraints. The decisions are reflected in design artifacts and architectural descriptions, e.g., documented views, but also ‘lower-level’ artifacts such as source code. A stakeholder responsible for an artifact changed due

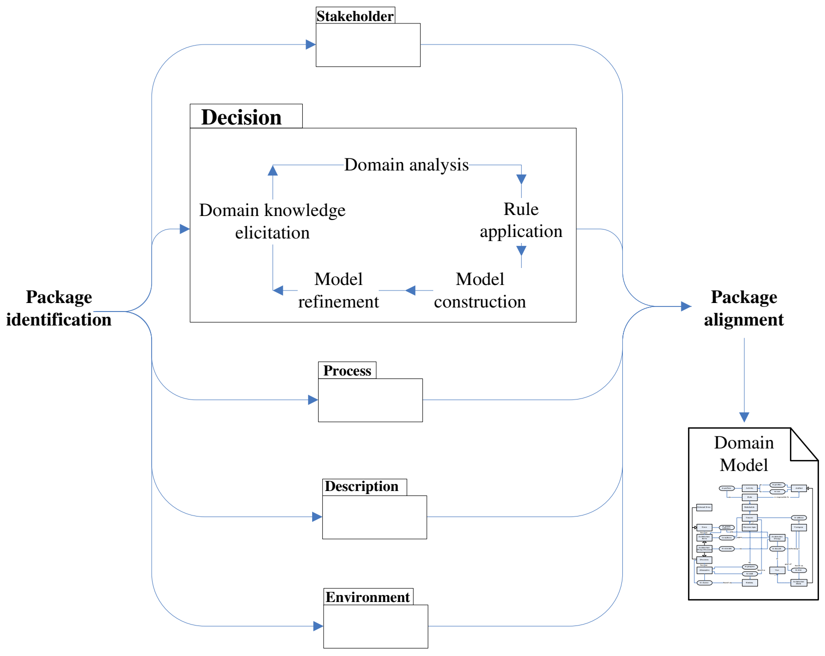


Figure 4.3: Outline of the design process followed

to a decision might find a need to take subsequent decisions. The responsibilities are defined by the stakeholders' role(s).

The schematic outline in Fig. 4.4 shows the interdependencies between architectural design decisions and design artifacts. In managing the architectural body of knowledge, the elements from this figure must be taken into account. Together, the architecting process, the stakeholders and their roles, their decisions, the architectural descriptions, and the environment sketch the outline of the architectural knowledge universe of discourse. We initially identified five packages, corresponding to these areas; 'Stakeholder', 'Decision', 'Process', 'Description', and 'Environment'. For each of these packages, the following activities have been carried out.

Domain knowledge elicitation [E]

Textual domain knowledge was acquired from various sources, including interviews with software architects, literature, standardization bodies, and our own experience.

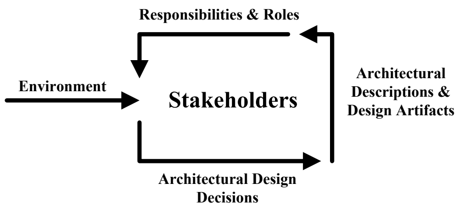

Responsibilities & Roles
Environment
Stakeholders
Architectural Descriptions & Design Artifacts
Architectural Design Decisions

Figure 4.4: Schematic outline of the architecting process

Besides experience with software architecture - partially captured in interview transcripts, partially our own - literature and standards were used as a source of domain knowledge. We used the IEEE-1471 standard for architectural descriptions of software intensive systems (IEEE 1471) as a basis for the 'Description' and 'Stakeholder' packages. The OMG's Software Process Engineering Metamodel (OMG, 2005a) standard was the foundation of the 'Process' package. Initial research on design decisions, such as (Tyree and Akerman, 2005), helped shaping the 'Decision' package. Due to a lack of literature on the environment of software architectures, the contents of the 'Environment' package remained open.

We were not particularly interested in the possible actors of an action, but rather in the action itself. To this end, the distilled sentences were 'anonymized' by replacing the subjects with 'someone'. So, instead of 'the architect takes a decision', we write 'someone takes a decision'.

During domain knowledge elicitation, redundant sentences are not yet filtered out. For example, the sentences "Someone takes a decision", "Someone chooses an alternative from a set of solutions", and "Someone picks an option" probably bear the same meaning. Chances are that these sentences will be merged during the rule application phase.

Domain analysis [E]

Grammatical analysis of the relevant sentences led to a set of domain objects as well as the most important actions on these objects. Based on the example sentences above, candidate objects are decision, alternative, solution, and option. Candidate actions for our domain model are 'to take' (a decision), 'to choose' (an alternative), and 'to pick' (an option).

Rule application [St]

Although listed as a separate phase, rule application took place for a large part in combination with domain analysis. In this phase, we judged among others the relevance of the sentences identified in the knowledge elicitation phase. This influences the sentences that are subject to the grammatical analysis in the previous phase.

For instance, we determined that the sentences "Someone chooses an alternative from a set of solutions" and "Someone picks an option" have exactly the same meaning; both the concepts 'to pick' and 'to choose' as well as 'alternative' and 'option' are interchangeable. Therefore, we eliminated one of the sentences from the set. The remaining candidate concepts were then further scrutinized. This led to the insight that 'alternative' and 'solution' refer to the same object. Since the alternative is chosen from a set of solutions, the alternative itself must be a solution. Furthermore, taking a decision can be seen as choosing from a number of alternatives. In other words, the chosen alternative *is* the decision. However, not every alternative is, or will be, a decision. This means that they both are unique domain objects.

Model construction [Sp]

By adhering to the FRS method's diagramming instructions, we constructed for each package an (initial) object-interaction model. See Fig. 4.5 for an example of the 'Decision' package.

Model refinement [St]

For most packages, the object-interaction model from the previous phase was not yet mature enough to be considered the end result. In the model refinement phase, we took each initial model and analyzed it for inconsistencies, and missing or superfluous information. This phase can be seen as another iteration of the previous phases.

As part of this phase, we also presented the model to the members of our research team that were not directly involved in the modeling process. We provided them not only with the graphical representation of the model, but also with the exact sentences reflected in the diagrams. These sentences can be reconstructed by generating a sentence for each domain action and its associated objects depicted in the model. For instance, the action 'to choose' in the model depicted in Fig. 4.5 corresponds to the sentence "Someone chooses from a number $(0 \ldots n)$ of alternatives based on a ranking, which results in a decision".

Based on this analysis, we solved a number of problems with the initial version of the model. Recurring problems were objects without instantiating actions, the need for a specialization relation, and domain objects or actions we had initially overlooked.

Package 'Decision'
Domain knowledge elicitation:

- Someone takes a decision
- Someone cancels a decision
- Someone executes a decision
- Someone chooses an alternative from a set of solutions
- Someone compares different alternatives
- Someone takes a decision on a certain topic
- Someone proposes an alternative
- Someone picks an option
- ...

Domain analysis:

- **Alternative**: A potential solution to a decision topic.
- **Criterion**: A condition in which alternatives can be distinguished.
- **Decision**: The choice of one from among a number of alternatives.
- **Decision topic**: A specific problem to be solved by the decision process.
- **Ranking**: The result of ranking, i.e., evaluating alternatives with respect to a criterion.

Model construction:

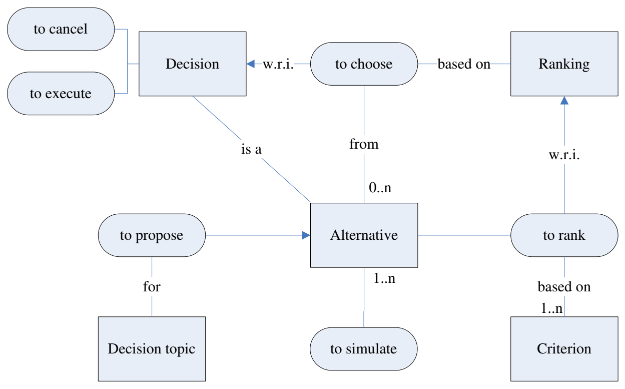


Figure 4.5: Constructing package 'Decision'

Objects without instantiating actions are a direct violation of the rule that every object must have an associated instantiating action. Instantiating actions were therefore introduced for these objects. The need for a specialization relation is often indicated by one object having the same association with two distinct objects. This need is satisfied by adding an object with a specialization relation to the latter two objects.

Package alignment [St/Sp]

During the refinement of the individual packages, more and more overlap between the packages started to emerge. Domain objects that were present in one package were also needed in other packages. In particular, (candidate) objects for the 'Stakeholder' package - which had not yet been completely refined - started to appear in all of the 'Decision', 'Description', and 'Process' packages. Furthermore, cyclic dependencies were starting to emerge. This was the trigger to try and align the individual packages into one coherent structure.

The alignment led to some further insights, and can perhaps be regarded as another refinement phase. For instance, the criterion that is used for ranking in the decision making process coincides with a stakeholder's concerns. Similarly, a stakeholder's concern is tightly linked to a decision topic that needs to be addressed.

Up until this moment, our efforts to model the 'Environment' package had only resulted in a list of possible external forces that might impact the architecture. Since the 'Environment' package had to be the representation of the 'outside world', it had in essence an unlimited scope. Therefore, it was possible to come up with new domain objects time and again. It turned out to be very hard to devise a structure that sensibly represents the whole environment. We decided that the best solution would be to capture the environment in a single domain object, dubbed 'External event'.

4.3.2 A model of architectural knowledge

The result of the package alignment phase is depicted in Fig. 4.6. In this figure, the rectangular elements represent entities; the elliptical ones denote actions on or with these entities. Arrows indicate the creation of new instances of an entity. Dashed lines represent attribute relationships. Sub-/superclass relationships are represented as generalizations (OMG, 2005b). Actions and entities that are linked together can be read as sentences, e.g "Someone chooses an alternative based on a ranking which results in a decision".

We tried to use this initial model of architectural knowledge to characterize the use of architectural knowledge in the four industrial organizations RFA, VCL, DNV, and PAV. This taught us that the model did not entirely fit all organizations. For example, the original model highly conformed to the IEEE-1471 standard for architectural description (IEEE 1471). This standard prescribes the use of so-called 'Viewpoints' to describe the architecture from the perspective of different stakeholders. The resulting 'Views' (partial descriptions of the architecture) are aggregated in a single architecture description. Although stakeholders and their concerns play a key role in any software architecting process, the tight coupling of the model to IEEE-1471's Views and View-

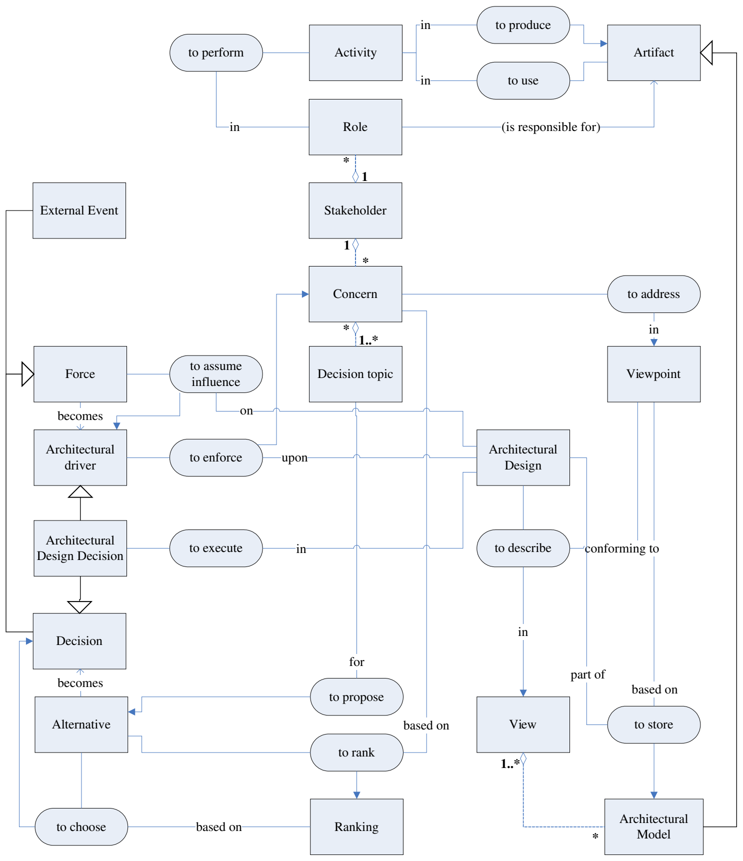


Figure 4.6: Initial theory of architectural knowledge

points turned out to be a mismatch with most organizations' practice. In hindsight this need not come as a big surprise, since organizations can (and do) use other approaches for documenting their architectures, which need not coincide with the IEEE-1471 way.

Although the initial model did not entirely fit all organizations, diagnosis of the use of architectural knowledge in those organizations at least showed that each of the organizations had its own perspective on architectural knowledge management, resulting in different issues at each of the organizations. The central issue within RFA was how to *share* architectural knowledge between stakeholders of a project. The main question within VCL was how *compliance* to architectural rules can be enforced in a multi-site environment. DNV was mainly concerned with how auditors can *discover* the architectural knowledge they need to perform an audit. The main challenge for PAV was how to improve *traceability* of its architectural knowledge. While the mismatches between theory and practice still prevented us from pinpointing the exact areas of improvement, at least we had an idea where to search for those areas in a next research iteration. However, this required that we removed the identified mismatches to further align theory with practice.

From a closer inspection of the mismatching concepts we learned that those concepts could either be expressed in terms of other concepts already present in the model, or as more generic concepts that *are* used by the organizations. This motivated us to restructure the model from Fig. 4.6. The restructured model has maximal expressivity in the architectural knowledge domain using a minimal set of core concepts. For this reason, we refer to the restructured model as a 'core model' of architectural knowledge. Elements in the core model cannot be expressed by using other elements in the core; elements that can be modeled in terms of core elements do not belong to the core:

- Both IEEE-1471 concepts of *Architectural Model*[2] and *View* are subsumed in the notion of ARTIFACT, i.e., an inanimate information bearer such as documents, source code, or books.
- Storing or describing the ARCHITECTURAL DESIGN in either of these artifacts can be abstracted to a single action 'to reflect'.
- The IEEE *Viewpoint* is a specific instance of a language used to reflect the architectural design in an artifact. The core model substitutes the more generic concept LANGUAGE for *Viewpoint*.
- An *External Event* that matters to any of the stakeholders can be expressed as a CONCERN of that STAKEHOLDER. Without external events there are no *Force*s or *Architectural Driver*s other than (ARCHITECTURAL DESIGN) DECISIONS, which removes the need for these concepts in the core model as well.
- The ENFORCEMENT of a decision upon the design is synonymous with the former notion of *execution* of an ARCHITECTURAL DESIGN DECISION.

The only change not related to removing concepts that can be expressed in terms of other core concepts, is that in the core model we view DECISION TOPIC no longer

[2]In this discussion, *italic* terms denote concepts only present in the initial model, while terms in SMALL CAPS are concepts (also) present in the core model. Likewise, in the remainder of this chapter *italic* terms denote concepts from non-core-models.

as a *part* of a CONCERN, but rather as a special *type* of CONCERN itself, namely a CONCERN for which a DECISION must be taken. Although this distinction was also present in the initial model, and hence does not conflict with the earlier interpretation, the new view makes the distinction clearer and more explicit. We shall discuss the interpretation of the core model in more detail in the next section.

4.4 A Core Model of Architectural Knowledge

In the core model of architectural knowledge, depicted in Fig. 4.7, the concepts of STAKEHOLDER and CONCERN coincide with the, widely accepted, definitions of these terms in IEEE-1471: a stakeholder is “an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system” (IEEE 1471). Both IEEE-1471 concepts of *Architectural Model* and *View* are subsumed in our notion of ARTIFACT. The ARCHITECTURAL DESIGN can be reflected using different LANGUAGEs, including models, figures, programming languages, and plain English.

Constructing an architectural design is essentially a decision making process. In our core model, decision making is viewed as PROPOSING and RANKING ALTERNATIVEs, and SELECTING the alternative that has the highest rank, i.e., the alternative that, after careful consideration based on multiple criteria (i.e., CONCERNs), is deemed to be the best option available with respect to the other alternatives proposed. It is especially this process of proposing, ranking, and selecting which is hard to articulate and distinguishes the good architects from the weaker. The chosen alternative becomes the DECISION. The alternatives that are proposed must address the DECISION TOPIC, and can be RANKED according to how well they satisfy this and other concerns. We view DECISION TOPIC as a special type of CONCERN, namely a CONCERN for which a DECISION must be taken. Example concerns for which no further decisions need to be taken – and which hence are no decision topics but do need to be taken into account when evaluating (ranking) proposed alternatives – are constraints, such as budget constraints, technological limitations, et cetera.

ARCHITECTURAL DESIGN DECISIONs are defined as those DECISIONS that are assumed to influence the ARCHITECTURAL DESIGN and can be enforced upon this ARCHITECTURAL DESIGN, possibly leading to new CONCERNs that result in a need for taking subsequent decisions. This ‘decision loop’ captures the relations between subsequent ARCHITECTURAL DESIGN DECISIONs. This loop also corresponds to the ‘divide and conquer’ technique of decision making, in which broadly scoped decisions are taken which may result in finer grained concerns related to the broader concern. Furthermore, it enables in theory traceability from concerns through decisions to artifacts, although this very much depends on whether those traces have been captured in

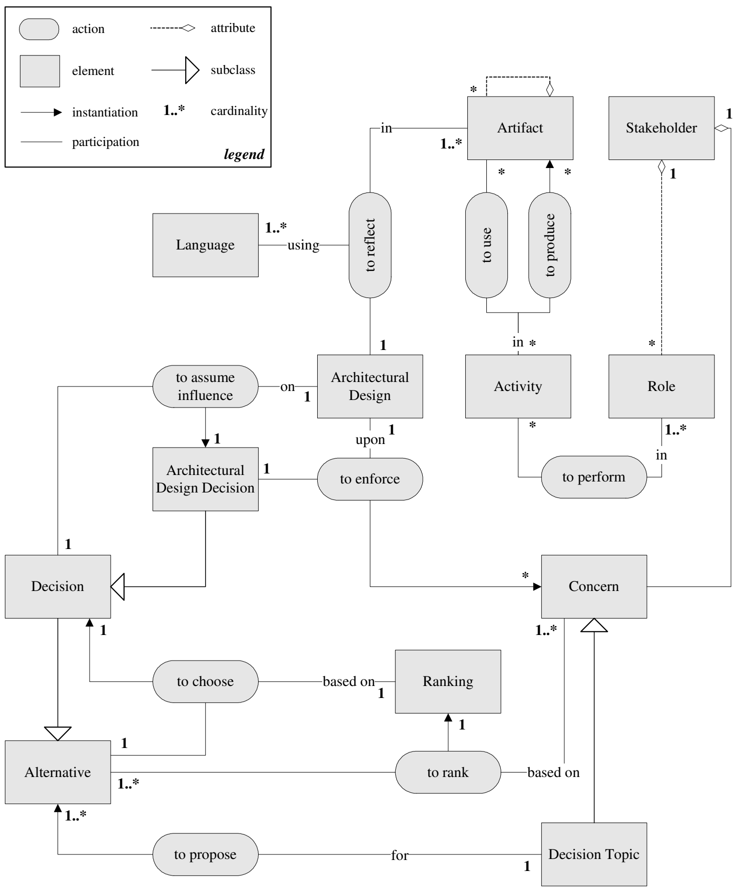


Figure 4.7: Core model of architectural knowledge

the reflection of the architectural design. Note that architectural design decisions need not necessarily be ‘invented’ by the architect himself; architectural patterns, styles, and tactics are examples of architectural design decisions (or, more precisely, alternatives) that are readily available from other sources. The ‘decision loop’ described above also captures the rationale of an architectural design; the answer to the question why an architectural design is the way it is. Rationale is in our opinion extremely interwoven with all elements in this loop, and is therefore not represented as a distinct element in our model.

The ARCHITECTURAL DESIGN (often called 'architecture' for short) is the result of all architectural design decisions. Note that reflection of (part of the) ARCHITECTURAL DESIGN is not limited to a single ARTIFACT: a single ARCHITECTURAL DESIGN DECISION might for instance be represented in the architecture description as well as impact the source code. ARTIFACTs themselves can again be composed of various (sub)ARTIFACTs, e.g., chapters in a document, or methods in a class. The concepts ROLE and ACTIVITY are borrowed from SPEM, which defines the software development process as "a collaboration between abstract active entities called process roles that perform operations called activities on concrete, tangible entities called work products" (OMG, 2005a). The 'work product' from SPEM resembles our notion of ARTIFACT. The latter is in our opinion a better known and widely accepted concept in Software Engineering.

4.5 Architectural Knowledge Use in Four Industrial Settings

We initially started our research with the goal to characterize the use of architectural knowledge in four industrial organizations. Although we were able to discover four different perspectives on architectural knowledge within those organizations (see §4.3.2), the mismatches of our initial model of architectural knowledge with the observed practice in those organizations hampered a further diagnosis of the problems those organizations encounter. Since those mismatches have been removed in our core model of architectural knowledge, it is interesting to see how the organization-specific 'models' of architectural knowledge of all four organizations can be expressed in terms of the core model. From a superficial look, each of the organizations appears to use architectural knowledge very differently. Alignment of the organization-specific models to the core model, however, allows for a more fundamental characterization of how the organizations perceive and use architectural knowledge.

The use of architectural knowledge in RFA and VCL is mainly located in the upper 'description' part of Fig. 4.7, i.e., the reflection of architectural design decisions in artifacts. The use of architectural knowledge within DNV and PAV is positioned more in the lower 'decision' part, i.e., the decision making process reflected in the decision loop. We hypothesize that the problems that the organizations experience in managing architectural knowledge are partially due to their focus on only a part of the theory of architectural knowledge as expressed by our core model.

An overview of the result of the four characterizations is provided in Table 4.1. In

Table 4.1: Diagnosis of industrial problems with architectural knowledge management

	RFA	*VCL*	*DNV*	*PAV*
Perspective	Sharing	Compliance	Discovery	Traceability
Main org. concepts	Design choices, Principles, Starting points, Prerequisites	Architectural rules	Quality criteria, Quality in use	Knowledge entities
Mainly used core concepts	Arch. design decisions	Arch. design decisions, *to enforce*	Arch. design decisions, Concern, Arch. design, *to reflect*	Concern, Decision topic Alternative, Arch. design decision
Problem	Ambiguous terminology	No sense of urgency regarding compliance with architectural rules	Implicit relation between architecture and "quality"	Lack of traceability between knowledge entities
Hypothesized cause	Decision making process not captured	Tacit decision making process	'Quality' and related arch. knowledge not confined to a single artifact	Implicit relations between artifacts
Possible solution	Explicit focus on relations between decisions through iterative 'decision loop'	Explicit focus on rationale of decisions through 'decision loop' elements	Uncover architectural design decisions, their cause, their effect on the software product	Annotate architecture documents with specific knowledge entities

this table we list for each of the organizations their prevalent perspective on the use of architectural knowledge, the main architectural knowledge concepts encountered (both organization-specific and at a core level), the hypothesized cause for the diagnosed problems, and a possible solution to this problem. In the following subsections, we further elaborate on these aspects for each organization in turn.

4.5.1 RFA: Development organization

RFA is a large development organization that develops and maintains software systems for among others the public sector. These systems are typically critical for the public, large in size and complexity, and long lasting. Because of the size of the organization and the projects, this organization focuses on how to effectively share architectural knowledge. To this end, RFA developed its own methodology and tooling to aid the software architects in creating and documenting architectural knowledge by means of architectural descriptions.

Architectural descriptions within RFA basically consist of a number of views based on predefined *viewpoints*, and a set of specific *architectural models* such as an object model, a functional data-model, etc. These models reflect a number of *architectural choices*, which are based on *decisions* that relate to *business objectives*. The choices take into account the *design principles*, *starting points*, and *prerequisites* to which the architect needs to adhere to when designing the software architecture, as well as the *stakeholder concerns*.

An example of a business objective documented in one particular architectural description is: *"The data of the subsystems needs to be easily accessible"*. A stated design principle based on this objective is *"The system needs to be accessible using web services as well as the Enterprise Service Bus"* and the final architectural choice made based on this principle is *"The system information exchange uses InfoMessaging and MS.NET Web Services"*.

We interviewed architects and managers from RFA, who in these interviews acknowledged that they are currently struggling with the different concepts, their relations and their more effective usage. This impairs effective sharing of architectural knowledge, since architects are unsure which concepts to use to describe their architectural design. As a result, readers are unsure where in the architecture description they can find the information they are looking for.

If we express the organization-specific terminology in terms of core concepts, an interesting pattern emerges. The 'different' notions of business objectives, design principles, and architectural choices all are in fact ARCHITECTURAL DESIGN DECISIONS, which are somehow related to each other. However, the organization's methodology does not define very concrete guidelines to distinguish between those decisions. RFA's struggle with terminology might partially be blamed on the use of different terms for the same architectural knowledge concept without a good definition of the discriminative features for these terms.

The explicit use of architectural knowledge within this organization can be primarily found in the 'description' area of the core model: reflecting architectural decisions in an architectural description. The decision making process – reflected in the decision loop in the model – is left implicit by the organization's methodology.

Our core model captures relations between different architectural design decisions in the decision loop, where a certain architectural design decisions leads to a new concerns that in turn leads to new decisions. A more explicit focus on this loop would help defeat the ambiguity in terminology within RFA. A design principle could then for instance be defined as a decision taken because of new concerns introduced by a business objective. In the example above, the concern introduced by the business objective would be the need for accessibility. Architectural choices are related to design principles analogously.

4.5.2 VCL: Consumer electronics

VCL is a large organization within the consumer electronics domain. This organization has arranged software development along subsystems. A release of the software for a consumer electronic product consists of integrating the relevant subsystems. Each subsystem is developed by a small, dedicated development team. The teams are located at multiple, geographically spread development sites.

This arrangement of the software and the software development activities demands guidelines to maintain the subsystem-based software architecture. To this end, a central architecture team issues architectural rules: a set of principles and statements on the software architecture that must be complied with throughout the organization.

Architectural rules originate from various *issues* that influence VCL's software development, such as defects identified in subsystem releases, change requests, or additional requirements. These issues need to be addressed in the software architecture. *Solutions* to these issues – which affect all subsystems – are captured in text-based documents. These documents are sent to all development teams as *architectural rules* that need to be adhered to.

The creation of architectural rules can be expressed in terms of the core model. The issues identified by the various stakeholders correspond to CONCERNs of STAKEHOLDERs. The architectural rules, i.e., the chosen solutions for these issues, are ARCHITECTURAL DESIGN DECISIONs that are ENFORCED upon the ARCHITECTURAL DESIGN through dissemination of the ARTIFACTs in which they are reflected.

Although VCL is similar to RFA in that the focus of architectural knowledge use is on the 'description' area of the core model, development in teams at distinct locations should put a particular emphasis on *enforcement* of architectural design decisions. However, once the architectural rules have been disseminated to the individual development teams, adherence to the rules on the subsystem architectures is the responsibility of the teams themselves. In practice, some of the rules are disregarded by the teams. Our core model suggests that one of the reasons for this might be the fact that only the architectural design decisions themselves are being communicated, while the decision making process itself remains tacit. This lack of insight into the reason of the architectural design decisions taken make that the development teams do not feel a sense of urgency regarding compliance with these decisions. We believe that more information about the rationale of the architectural rules, including the concerns that led up to the decisions, increases this sense of urgency with the developers.

4.5.3 DNV: Quality audits

DNV is a company that performs independent software product audits for third parties. A software product audit consists of comparison of *quality criteria* with the actual software product. Most quality criteria assess the effects of *architectural decisions* as reflected in the *software product artifacts*. A quality criterion might for instance be *"All access to data in a relational database should take place through dedicated data access objects. No direct communication of business objects with the database is allowed"*.

The customer that acquires a software product expects this product to have a certain 'quality in use' (ISO/IEC 9126-1). Given the concern 'quality in use', there are various quality characteristics for which quality criteria must be selected. For instance, the criterion that all data access must take place through data access objects favors the maintainability of the software product over its efficiency. Selection of quality criteria therefore depends on the relative importance of each of the quality characteristics indicated by the customer. The example criterion will only be selected if maintainability of the software product is indeed more important to the customer than efficiency.

The problem an auditor faces when performing a software product audit is that architectural design decisions and the resulting architectural design are usually not reflected in a single artifact. Even if there is a document called 'the architecture description', architectural decisions impact other product artifacts (e.g., documentation and source code) as well. There is no guarantee that the information in an architecture description is complete, or even up-to-date.

The reflection of the effects of architectural decisions in different software product artifacts can be readily identified in our core model. The LANGUAGE used to reflect ARCHITECTURAL DESIGN in ARTIFACTs can be a natural language (e.g., English in software product documentation), but also a programming language (source code) or graphics (e.g., diagrams and figures in a software architecture description). A more interesting and less apparent mapping is the mapping of 'quality' to the core model.

In terms of our core model, *quality in use* is a CONCERN of the customer, who is a STAKEHOLDER. The quality *characteristics* and *subcharacteristics* are DECISION TOPICs for which *quality criteria* are proposed and selected. The proposed criteria are the ALTERNATIVEs. Selection of quality criteria is based on their impact on the quality in use – the CONCERN– in terms of prioritized (sub)characteristics, as indicated above. In this way, trade-off analyses are being made regarding conflicting criteria. The chosen quality criteria describe the architecture in terms of how it ought to be. In other words, quality criteria are a special type of ARCHITECTURAL DESIGN DECISIONs: decisions that are expected to have influenced (rather than enforced upon) the ARCHITECTURAL DESIGN.

In DNV the relation between architecture and quality is not obvious at first sight. The core model helps us to describe quality in terms of architectural decisions and their effect on the software product, and shows us that quality criteria that apply to the architectural design are themselves architectural decisions. Using the core model, we can express a software product audit as a comparison of two types of architectural knowledge: architectural knowledge that is present in the software product, reflected in the ARTIFACTs that make up the product, and architectural knowledge that is expected by the customer, reflected in quality criteria. This makes it more apparent which architectural knowledge is most important for a software product audit: the architectural design decisions as well as their cause (e.g., stakeholder concerns and trade-offs) and effect (e.g., constraints on subsequent decisions) should be discovered from multiple artifacts for an effective assessment of a software product's quality.

4.5.4 PAV: Scientific research

PAV is a scientific organization that is involved in the development of large software-intensive systems, used for scientific research. One of their projects is the development of a highly distributed system that collects scientific data from around 15.000 sources, distributed over 77 different stations, each source generating around 2 Gbps of raw data. The challenge for this system is to communicate and process the resulting 30Tbps data stream in real-time for interested scientists.

In this project, architectural decisions need to be shared and used over a time span of more than 25 years. This is due to the long development time (more than 10 years), and a required operational lifetime of at least 15 years. The organization is judged by external reviewers on the quality of the architecture and the outcome of these reviews influences the funding, and consequently the continuation, of their projects. Therefore, it is of paramount importance to keep the system architecture at a high quality. In order to achieve this purpose, PAV needs to evaluate at all times the design maturity, the completeness, the correctness and the consistency of the architecture.

The evaluation of the architecture is to be performed at the level of *knowledge entities*, which are units of architectural knowledge shared and communicated among the project *stakeholders*. There are four different types of *knowledge entities*: *problems* that state how specific functional requirements or quality attributes must be satisfied; *concerns* that comprise any interest to the systems development, its operation or any other aspect that is critical or otherwise important to one or more stakeholders; *alternatives* that solve the described problem, potentially in different ways and with different consequences; *decisions* that denote the selection of one among multiple alternatives.

During the architecting process, the architect takes a number of architectural decisions that are gradually being refined into more low-level, technical decisions. The lowest level of an architectural decision is called a *specification*, and the architecting process finishes when all architectural decisions have been refined into specifications.

Knowledge entities can be expressed in *artifacts* that are documents in electronic or printed format. The organization also considers *artifacts* of smaller granularity, called *artifact fragments*, such as individual sections, paragraphs or pictures in a document, in order to be able to trace fine-grained *knowledge entities* within a single document. Finally, it is of high importance for the organization to keep trace of how the *requirements*, described in the *requirements specification* document are satisfied in the *software architecture* document. Some *requirements* have associated *risks*.

We can express PAV's organization-specific terminology in terms of our core model. A *problem* is being solved by the alternatives, which coincides with the core model concept of a DECISION TOPIC. The *knowledge entity* on the other hand is a generalization of four core model concepts, namely CONCERN, ALTERNATIVE, DECISION TOPIC and DECISION. A *specification* is a special, refined case of a DECISION. An *artifact fragment* is an ARTIFACT contained in other ARTIFACTs. *Requirements* and *risks* are special types of CONCERNs that need to be taken care of in the decision making process.

During our research in PAV we found that – although many of the core concepts are present – most relations between artifacts remain implicit, potentially leading to traceability issues. The mapping between the core model and the concepts specific to this organization brings to the architect's attention that four of the fundamental core model concepts, namely CONCERNs, DECISION TOPICs, ALTERNATIVEs and DECISIONs are in fact special cases of *knowledge entities*. With this in mind architects can annotate architecture documents with higher level of detail by taking into account the different types of knowledge entities. A more explicit focus on the core model's 'decision loop', and in particular the individual elements that make up this loop, is likely to result in better traceability.

4.6 Core Model Validation: Attempted Falsification through Literature

The industrial experiences described in §4.5 showed a practical application of our core model of architectural knowledge. In this section we determine the core model's theoretical significance by comparing its concepts to architectural knowledge concepts used in accepted software architecture literature.

As defined in §4.3.2 our core model is *minimal* in the sense that it is not possible to express some concepts in any other concepts, and *complete* in the sense that there are no concepts from other approaches that have no counterpart in the model. Unfortunately, it is impossible to 'prove' that our model exhibits the desired features of being complete and minimal. The best we can do is to search for counterexamples that prove our model does *not* exhibit those features, thereby demonstrating that our model is *not* valid. If we don't succeed in this falsification attempt, we accept that as supporting evidence for the validity of our model.

To properly apply the falsification approach on our core model, we have mapped on our model the complete set of concepts from three different terminological frameworks for architectural knowledge well-known from literature. Each of these frameworks has a slightly different perspective on architectural knowledge: IEEE-1471 (IEEE 1471) targets architectural descriptions, Kruchten's ontology (Kruchten, 2004) focuses on

Table 4.2: Falsification attempts on core model using software architecture literature

Core concept	**IEEE-1471**	**Kruchten's ontology**	**Tyree's decision template**
STAKEHOLDER	*Stakeholder*		
CONCERN	*Concern, Environment, Mission*	*Requirement, Defect, Risk, Plan*	*Assumption, Constraint, Requirement*
DECISION TOPIC		*scope*	*Issue, Group*
ALTERNATIVE		*Idea, Tentative*	*Position*
RANKING			*Argument*
ARCHITECTURAL DESIGN DECISION		*Decision*	*Assumption, Decision, Principle*
"DECISION LOOP"	*Rationale*	*relationships*	*Related decisions / requirements / principles*
ARCHITECTURAL DESIGN	*Architecture*		
LANGUAGE	*(Library) viewpoint*		
TO REFLECT	*(Library) viewpoint*		
ARTIFACT	*System, View, Model, Architecture description*	*Technical artifact*	*Artifact*

architectural design decisions per se, while Tyree and Akerman (2005) provide a template to capture architectural design decisions – thereby relating architectural decisions to architectural descriptions. Together, these perspectives cover all 'corners' of our core model.

Our falsification attempts are summarized in Table 4.2, which shows the mapping between core model concepts and the concepts of the three terminological frameworks for architectural knowledge. The table shows that we failed to find any concepts that do not fit our core model; we accept this result as support to the claim of validity of our core model. In the following subsections the relationships between core model concepts and concepts of each of the literature frameworks are elaborated upon.

4.6.1 Architectural descriptions: IEEE-1471

IEEE-1471 prescribes the use of so-called *view*s to describe an architecture. Views are reflections of part of the architectural design according to a particular perspective, or *viewpoint*. A viewpoint defines "the language, modeling techniques, or analytical methods to be used in constructing a view based upon the viewpoint" (IEEE 1471). In other words, a viewpoint defines the LANGUAGEs to use as well as how to *reflect* the architectural design in a view. The IEEE-1471 terms *model* and *view* are both subsumed in the core model concept ARTIFACT.

A *library viewpoint* is a *viewpoint* that is defined elsewhere, i.e., a specialized instance of a normal viewpoint. The core model captures the *rationale* of an architectural design decision in the trajectory from CONCERN and DECISION TOPIC through

RANKING of ALTERNATIVES to the eventual CHOICE of the DECISION.

In IEEE-1471 terms, a *system* has an *architecture*, reflected in the core model as an ARCHITECTURAL DESIGN that is reflected in a set of ARTIFACTs, which together correspond to the IEEE *system*. The *architectural description* is a particular ARTIFACT that conforms to the IEEE prescription of how the ARCHITECTURAL DESIGN should be reflected, i.e., using a *viewpoint* as LANGUAGE. The *environment* in which a system operates determines the setting and circumstances of developmental, operational, political, and other influences upon that system. These influences are represented by CONCERNs in the core, as described in §4.4.

Finally, a *mission* is defined as 'a use or operation for which a system is intended by one or more stakeholders to meet some set of objectives'. This is a special case of a CONCERN, i.e., 'an interest which pertains to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders'.

4.6.2 Documenting design decisions

Tyree and Akerman (2005) consider important decisions as the major forces that drive architecture. They present a template that can be used to document design decisions. According to this template, *assumptions* and *constraints* limit the alternatives that can be selected. Assumptions are DECISIONs that are assumed to have been taken and to influence the ARCHITECTURAL DESIGN, often resulting in new CONCERNs for STAKEHOLDERs. Constraints are posed by decisions already taken, reflected in new CONCERNs. These new CONCERNs are to be taken into account when ranking ALTERNATIVEs for a DECISION TOPIC.

During the architecting process, an architect comes up with *positions* for a certain *issue*. Ultimately, one of the *positions* is chosen based on some *argument*. This position becomes the *decision*. This sequence of steps is also visible in our core model, as the proposal of a set of ALTERNATIVES, out of which one ALTERNATIVE is chosen as DECISION.

A *group* can be used to organize a set of decisions based on their topic (e.g., integration, presentation, etc.). In our core model, DECISION TOPICs (i.e., concrete subjects for which a solution is proposed) correspond to the *group* concept of Tyree and Akerman (2005). Their template lists the positions (i.e., ALTERNATIVEs) 'Rearchitect existing batch logic in System A', 'Extend System B to handle a new product type', and 'Develop a replacement for System A' for the issue 'Current IT infrastructure doesn't support interactive approval functionality for most financial products', with a grouping labeled 'System structuring'. The example template clearly shows the proposed *positions* all target the group (i.e., DECISION TOPIC) 'System structuring'.

According to Tyree and Akerman, a *decision* states the architecture's direction. This corresponds to our notion of an ARCHITECTURAL DESIGN DECISION that is enforced upon the ARCHITECTURAL DESIGN. The template has room for the documentation of *related decisions*, *related requirements*, *related artifacts*, and *related principles*. The way in which DECISIONs (that include principles), ARTIFACTs, and CONCERNs (i.e., requirements) can be related has been extensively described in §4.6.3.

4.6.3 Ontology of architectural design decisions

Kruchten (2004) defines an ontology of architectural design decisions. He argues that 'design decisions deserve to be first class entities in the process of developing complex software-intensive systems' and proposes a model to do so.

Decision kinds The ontology defines three classes of architectural design decisions: existence decisions ('ontocrises'), property decisions ('diacrises') and executive decisions ('pericrises'). In terms of our core model, these design decision classes are extensions of the core concept ARCHITECTURAL DESIGN DECISION.

Decision attributes Kruchten defines a number of attributes of architectural design decisions, all of which can be mapped to concepts in our core model. The *epitome* is the ARCHITECTURAL DESIGN DECISION itself, summarized in a short textual statement (e.g., 'Use Java'). *Rationale* is the justification of a decision, captured in the core model by the trajectory from CONCERN and DECISION TOPIC through RANKING of ALTERNATIVEs to the eventual CHOICE of the DECISION. The *category* of a decision can be mapped on the DECISION TOPIC concept. In our core model, CONCERNs consist of one or more DECISION TOPICs for which a DECISION must be taken. These DECISION TOPICs are concrete subjects for which a solution is proposed. As indicated by Kruchten's remark, categories are "useful for exploring sets of design decisions that are associated to a specific concern or quality attribute". The *author* of a decision corresponds to the STAKEHOLDER that took the decision. A *timestamp* is an intrinsic property of any action, including the actions represented by ellipses in Fig. 4.7. The *scope* of a decision is an intrinsic property as well, which affects the enforcement of the decision. Examples of scope listed by Kruchten include time scope ("Until the first customer release...") and organization scope ("The Japanese team..."). The *cost* and *risks* associated with a decision are both forms of STAKEHOLDER CONCERNs.

Decision evolution The evolution of design decisions is captured in the *state* attribute. In the early decision making phase, for instance, *Idea*s and *Tentative* decisions (i.e., decisions with a state 'idea' or 'tentative') correspond to the core concept of an ALTERNATIVE. Both are not yet decisions, and might never become one. Tentative decisions can be used in running 'what if' scenarios, i.e., a ranking of different ideas.

An architectural design decision that is *decided* is represented as an ARCHITECTURAL DESIGN DECISION in our core model. If a decision is *approved*, this means it is enforced upon the ARCHITECTURAL DESIGN. The three remaining states have to do with 'canceling' earlier taken design decisions. Our core model, however, exhibits the notion of an ever-growing 'decision pool' that may or may not influence the architectural design. Rendering a decision *challenged* consists of taking a new decision that a previously taken architectural design decision should no longer influence the architectural design. ENFORCEMENT of such a decision upon the architectural design results in a change of the architectural design, thereby removing the influence of the earlier taken decision and effectively *rejecting* the earlier decision. Finally, *obsolesced* is similar to the *rejected* state, but without an explicit rejection of a decision. In other words, the decision is conceptually still in place, but – e.g., through evolution of the design – its influence on the architectural design has worn off. The *history* of a decision consists of all the 'state changes' that apply to that decision, including the introduction of new decisions to challenge or reject earlier decisions.

Relations The ontology further defines a number of relations between decisions. All these relationships are manifested in the 'decision loop' in the core model. Kruchten's notion of alternative design decisions that address the same issue (i.e., the relation *is an alternative to*) maps directly to the core concept of different proposed ALTERNATIVEs for a single DECISION TOPIC.

The relation 'decision A *constrains* decision B' implies that B is tied to A and must be in the same state as A. For instance, 'Must use J2EE' constrains 'use JBoss as Application Server'. In terms of the core model, the DECISION 'Must use J2EE' introduces a new DECISION TOPIC 'which application server to use?'. The ALTERNATIVE 'JBoss' could not have been chosen without the decision to use J2EE.

The relationship *bound to* is defined in terms of the *constrains* relation. Decisions A and B are bound to each other when A constrains B and B constrains A. This relationship cannot be expressed in our core model: it translates to a sequence in which Decision A taken for Decision Topic DT_A introduces a new DECISION TOPIC DT_B for which DECISION B is taken, which in turn introduces again the need to decide on DT_A. Unfortunately, Kruchten does not provide an example for this relation, but in our opinion it relates to a kind of 'atomicity' that would be expressed as a single decision in the core model.

The relation A *enables* B is a weaker form of *constrains*, since decision A does not introduce the decision topic DT_B for which B is taken. Instead, DECISION A leads to a CONCERN that is taken into account when ranking ALTERNATIVEs and subsequently taking decision B. Although A is not a necessary prerequisite for B, once A has been explicitly taken the alternative that becomes decision B will rank higher. An example of this relation is from the decisions ‘Use Java’ to ‘Use J2EE’.

In the relation ‘decision A *forbids* decision B’, decision A also leads to a Concern that is used in ranking of alternatives when taking decision B. The difference with ‘enables’ is that the chosen alternative for decision B is now has a negative impact on the ranking of Alternative B, i.e., B will never become a Decision.

The relation A *comprises* B is a stronger relation than *constrains*, in the sense that it does not only affect new Decision Topics introduced because of Decision A, but other Decision Topics as well. These Decision Topics were already present (or would be introduced) because of other Concerns. In a *constrains* relation, the affected Decision Topics would not be present without Decision A. An example of this relation is the decision ‘use middleware framework M’ that introduces a new concern C ‘can only use services from M’. When in the ranking of proposed alternatives for the Decision Topic ‘how to handle message passing’ concern C is taken into account, the only viable alternative is ‘Message passing must use messaging services from M’. Moreover, this alternative could not have been chosen without Decision A (i.e., the use of framework M). This is the main difference with the *enables* relation. In a *comprises* relation, decisions for multiple decision topics DT_{B_1}, $DT_{B_2}, \ldots DT_{B_n}$ can be traced back to a single concern resulting from Decision A.

Explicitly ignoring concern C in ranking alternatives for DT_{B_i} introduces an exception to Decision A, denoted by the relation ‘Decision B_i *overrides* Decision A’. When a concern that results from an earlier decision A has – by accident – not been used in the ranking of alternatives for decision B, a *conflict* between decisions A and B may arise. Note that the relations ‘B overrides A’ and ‘B conflicts with A’ are both mapped in the same way to the same concepts in the core model. The only difference between the two relations is the reason a Concern is not used in the ranking: on purpose in ‘overrides’ vs. accidental in ‘conflicts with’.

When decision A *subsumes* decision B, decision A is is more encompassing than decision B. An example of this relation is described by the following sequence. The decision ‘must code the system’ results in a Concern ‘which language to use to code the system’ which can result in either the decision ‘must code the system in Java’ or the decision ‘must determine in which language to code the subsystems’. In the latter case, new decision topics are introduced for ‘which language to use to code subsystem ‘X’’ for each subsystem. In the former case, these decision topics are not introduced but subsumed in the more general Decision Topic ‘which language to use to code the system’. The decision ‘code the system in Java’ therefore subsumes each of the decisions 'code subsystem 'X' in Java'. Kruchten describes how a decision B (code subsystem 'X' in Java) can later be generalized into decision A (code system in Java), obsolescing decision B (see states described above). Immediately taking Decision A 'prevents' the introduction of the new Decision Topics for subsystems X, Y, and Z.

Besides relations between decisions, Kruchten names several relations with 'external artifacts'. Design decisions *trace from* technical artifacts upstream: requirements and defects (i.e., CONCERNs in the core model), and *trace to* technical artifacts downstream: design and implementation artifacts (i.e., ARTIFACTs in the core model). They are also traceable to management artifacts, such as risks and plans (again CONCERNs). Finally, Kruchten notes that it may be useful to track which portions of the system are *not compliant with* some design decisions. In the core model, this non-compliance corresponds to a reflection in ARTIFACTs of an ARCHITECTURAL DESIGN upon which some ARCHITECTURAL DESIGN DECISIONs have not yet been enforced.

4.7 Core Model Applications

Besides providing a common frame of reference that allows discussions *about* architectural knowledge concepts (such as the ongoing discussion in this thesis), the core model can also be applied to *exchange* architectural knowledge. Two types of exchange come to mind: one internal to organizations (intra-organizational exchange), and one external (inter-organizational exchange).

4.7.1 Intra-organizational knowledge exchange: Project memories

Over the past decade, the notion of learning software organizations, i.e., organizations that improve and extend their software engineering knowledge through continuous learning and exchange of experience has gained considerable interest. Such learning needs can be addressed by systematic application of organizational learning principles (Dingsøyr et al., 2006), supported by organizational (project) memories (Althoff et al., 2000).

Project memories can support various software engineering activities, such as version control, change management, documentation of architectures and design decisions (Moran and Carroll, 1996), and requirements traceability (Ramesh and Jarke, 2001). Learning from and improvement of the software development process can be supported by predictive models that guide decision making based on past projects (e.g., (Rus and Lindvall, 2002; Ruhe, 2002).) Such decision guidance also needs the knowledge incorporated in a project memory.

In general, different types of knowledge should be present in a project memory: know-how, know-why and know-what (Ruhe and Bomarius, 1999). Existing notational and documentation approaches to software architecture typically focus on the components and connectors. These approaches fail to document the design decisions embodied in the architecture as well as the rationale underlying the design decisions. This means that often the architectural knowledge present in a software engineering project memory adequately covers know-what and know-how, but fails to address the know-why of a software architecture.

Failure of addressing know-why results in high maintenance costs, high degrees of design erosion, and lack of information and documentation of relevant architectural knowledge. The know-why - or rationale - of an architecture manifests itself through the design decisions that are incorporated in the architecture.

The core model of architectural knowledge captures the interdependencies between architectural design decisions, architectural descriptions and the various stakeholders of the system, and treats design decisions as first-class entities. This ensures that not only the software artifacts themselves are subject to an organization's knowledge management efforts, but also the technical, organizational, political and economic influences on these artifacts. Based on the core model, a network of design decisions can be instantiated. Combined with the associated design artifacts, this network could then serve as a project memory containing vital architectural knowledge, including the now oft-lacking architectural know-why.

4.7.2 Inter-organizational knowledge exchange: An architectural knowledge grid

In an inter-organizational setting, we can look at the core model from two perspectives, namely data integration and service integration. For data integration the core model becomes a reference model for exchanging architectural knowledge. For service integration, it provides the means to integrate the services that a grid infrastructure may provide.

Having a core model of architectural knowledge has a number of advantages. First of all, from a data integration perspective, the core model defines a vocabulary for architectural knowledge: the minimal set of common notions that is needed when architectural knowledge has to be made explicit. Terminology, processes, and concerns particular to a specific organization or domain can be expressed in terms of core model concepts. Metaphorically speaking, the organization-specific terminology lies like a shell around the core model. One can even envision multiple layers of shells, for instance when an organization defines its own methodology (the outer shell) as an extension to the IEEE-1471 standard (the inner shell). Thanks to this separation between core model and shells, we gain in terminological stability (the core model acts as a reference model used by all companies), extensibility (the architectural knowledge of new companies or domains can always be added as a new shell without any increase in complexity), and reuse (by adding new shells, the architectural knowledge vocabulary is incrementally enriched).

Moreover, with a shared core model it becomes easier to agree on a common terminology. This common terminology sticks to the essence and neglects the specific concepts delegated to the shells. These benefits are reaped whenever multiple departments in the same organization, or even different partner organizations, have to collaborate in the same software project: terminological misunderstandings are avoided.

Furthermore, from a service integration perspective the core model can be the means to integrate the services that a grid infrastructure may provide. These services may 'speak the same language' by exchanging data expressed in concepts from the core model. A direct benefit of this language uniformity is that the core model, being shared among multiple sites, realizes a more generic architectural style aimed at integration via an enterprise data model (Gorton, 2006): the enterprise data model (i.e., our core model) is the target format for all messages between the grid members, which transform their specific formats to the target format. Such transformations are defined as shells.

We envision architectural knowledge exchange in a grid-setting (Lago et al., to appear), an instantiation of the knowledge grid discussed in (Zhuge, 2004). The basic idea is sketched in Fig. 4.8. The model depicted in the center is the core model of architectural knowledge. Organization-specific models provide a specialization hereof. In the figure, *design principles* (a kind of ARCHITECTURAL DESIGN DECISION) are made available by RFA, while *quality criteria* (another kind of ARCHITECTURAL DESIGN DECISION) are made available by DNV (see also §4.5). Both organizations may offer visualization services (e.g., tabular versus graph-based) to visualize ARCHITECTURAL DESIGN DECISIONs. Suppose a third organization, X, is looking for ARCHITECTURAL DESIGN DECISIONs that are shared on the grid. Because of the specializations of the notion of an ARCHITECTURAL DESIGN DECISION at RFA and DNV, this query translates into a search for *design principles* at RFA, and *quality criteria* at DNV. Both results will be returned, even using the local visualization services of RFA and DNV.

4.8 Conclusions

In this chapter we have presented a model of architectural knowledge. This model is the result of the execution of an action research cycle. Experimentation with an earlier

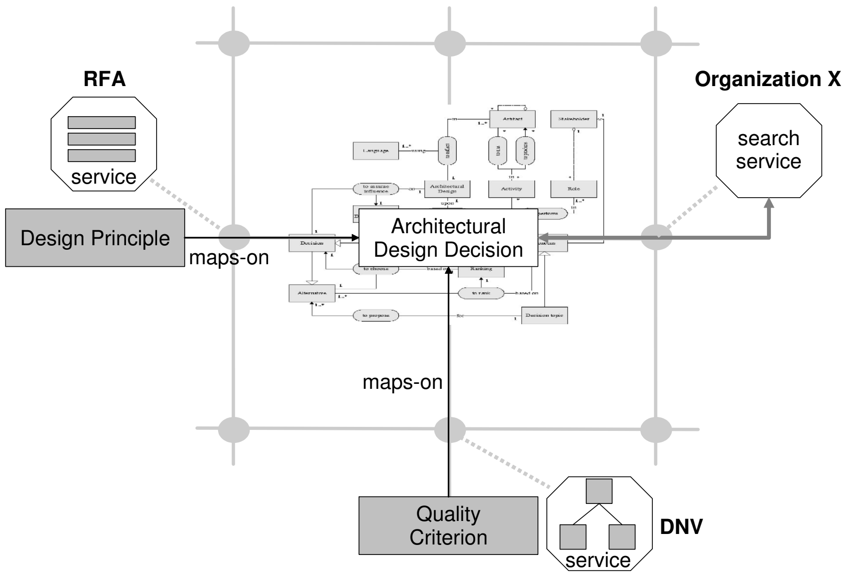


Figure 4.8: Architectural knowledge exchange in a grid-setting

version of the model identified mismatches between the model and industrial practice. Those mismatches have been overcome by ensuring the new version of our model of architectural knowledge is both complete and minimalistic. Because of the minimalistic aspect of the model, we refer to it as a ‘core model’ of architectural knowledge. The validity of the claim of completeness of our core model is made plausible by an attempt to falsify this validity using various sources from literature.

Application of the FRS method results in a model that captures both static and dynamic aspects. By capturing dynamic aspects of a domain, the resulting model not only focuses on what exists in a domain (e.g., design decisions, architectural descriptions), but also on how things happen (e.g., the decision making process). This allows us to model the architecting process as a whole, including dynamic relations between otherwise static objects.

Hofmeister et al. (2007) define architecting as an iterative process in which the architecture 'grows' over time as architects perform architectural activities, such as analysis, synthesis, and evaluation. In the core model we build further on this view, by considering the iterative nature of architecting as a 'decision loop'. We focus not only on the design decisions themselves, but also on the result of this iterative process – the architectural design – which is reflected in various design artifacts such as architectural descriptions.

The application of the core model to characterize the use of architectural knowledge in four industrial organizations brought the organizations tangible benefits: RFA moved a step further in improving its corporate terminology. By making explicit the missing link between the architectural design decisions (or rules) and the rationale behind their enforcement, VCL can improve compliance with corporate architectural rules. DNV can codify the knowledge about what is present in a software product and what is *expected* to be present, hence providing a better support for software project audits. PAV can now support its architects in annotating its documents with architectural knowledge in a more efficient way, thereby facilitating safeguarding the quality of the architecture.

5 Conclusions

In this part we have explored the domain of architectural knowledge management. Using both desk research and field research we answered three research questions, introduced in §2.2. In this part, we have established an understanding of what architectural knowledge is, how it can be managed, and what related challenges exist in (industrial) practice. In §5.1 we elaborate upon our contributions by revisiting and answering our three research questions. In §5.2, we discuss several challenges that need further investigation. Two of these challenges will be addressed in Parts II and III.

5.1 Contributions

5.1.1 What is architectural knowledge? (RQ-I.1)

Our research provides a dual answer to the question what architectural knowledge is. On the one hand, we investigated what the current view on architectural knowledge in literature is (Chapter 3); on the other hand, we have provided our own reference model of architectural knowledge (Chapter 4).

Our investigation of existing literature has shown that there is no single view on architectural knowledge, but that instead there are four prime ways in which one can look at architectural knowledge: the pattern-centric view, the dynamism-centric view, the requirements-centric view, and the decision-centric view. In all views, we can distinguish between tacit and explicit architectural knowledge, as well as between application-specific and application-generic architectural knowledge. Moreover, all views on architectural knowledge can be linked to the concept of architectural design decisions; constructing an architectural design is essentially a decision making process.

Architectural design decisions play a pivotal role in our own model of architectural knowledge, in the form of the 'decision loop'. In this model, (architectural) decision making is viewed as proposing, ranking, and selecting alternative solutions to address stakeholder concerns. Each chosen alternative, i.e., architectural design decision, may introduce new concerns and hence lead to subsequent architectural design decisions. A key property of our 'core model' of architectural knowledge is that other models can be expressed in terms of core model concepts. Hence, the core model can be used as a common frame of reference in discussions on and management of architectural knowledge.

5.1.2 How can architectural knowledge be managed? (RQ-I.2)

The question how architectural knowledge can be managed is answered in Chapter 3 through a combination of literature review and an investigation of the state of the practice in industry.

Our systematic review led to the recognition of different views on architectural knowledge, and the definition of four architectural knowledge categories. Each of the views on architectural knowledge has a corresponding philosophy on how such knowledge should be managed. These philosophies can be expressed as knowledge conversions between the different categories.

While many single-philosophy approaches to architectural knowledge management exist, a shift to more overarching approaches can be observed. It seems that a fifth, 'decisions-in-the-large' philosophy is emerging. In this philosophy, architectural knowledge is related to the whole architecting process, and not to either the problem domain, or the solution domain, nor to decisions per se ('decisions-in-the-narrow'). The decisions-in-the-large philosophy appears to be driven by the question how and when architectural knowledge can be used, e.g., for architecting, sharing, assessing, or learning.

Strategies to manage architectural knowledge can be broadly classified in 'personalization' and 'codification'. While many state-of-the-art approaches follow the codification strategy, industrial practice implicitly relies on personalization. The personalization strategy is hardly ever intentionally chosen. Codification has the potential to contribute to better management of architectural knowledge, given that important social aspects are taken into account to get such systems into use. However, there are good reasons for many organizations to opt for a personalization approach in order to facilitate innovative solutions with minimal bureaucracy. A hybrid strategy, which combines codification and personalization probably suits typical architecting activities best.

5.1.3 What are typical challenges to architectural knowledge management in practice? (RQ-I.3)

We answered the question what typical architectural knowledge management challenges are in Chapter 4. Those challenges were identified by characterizing the use of architectural knowledge in four industrial organizations. We identified four high-level challenges regarding architectural knowledge management in these four organizations:

- **Architectural knowledge sharing** is the challenge to improve the way architectural knowledge is shared within and between organizations;
- **Architectural knowledge discovery** is the challenge to enhance the discoverability of relevant architectural knowledge;
- **Architectural knowledge compliance** is the challenge to support alignment to common architectural rules;
- **Architectural knowledge traceability** is the challenge to enable effective navigation through an organization's body of architectural knowledge.

5.2 Discussion

The answers to the research questions lead to follow-up questions, some of which will be answered in this thesis and some of which warrant future research.

Regarding the appropriate choice of knowledge management strategy, we believe two phases of architecting should be distinguished: a creative first phase, and a rational second phase. The first phase is mostly related to the production of architectural knowledge, while the second phase targets architectural knowledge consumption.

In the first phase, the decision making process is a rather unstructured process in which the architectural solution space is explored and ideas are coined. While it could already very much benefit from codified knowledge (such as architectural styles, patterns, and tactics), this creative architecting phase seems to be particularly suited for a personalization strategy. Although it is important that architectural knowledge such as expertise, options under consideration, and the like can be located when needed, this need not necessarily be through structured knowledge repositories. In this phase, meetings between architects (and other stakeholders) and ad-hoc communication by means of for instance email discussions are little intrusive and pose no limitations on the options that can be considered. Architects might feel such limitations when forced to codify each and every option they consider.

In the second phase, the design space is outlined by approved architectural decisions, and a stable architectural design emerges. This phase lends itself for a rationalization of the earlier 'unstructured' decision making process. Traces from the first phase, present in for instance emails, discussion fora, and meeting minutes, can be used to 'reconstruct' the rationale for the current architectural design. In this phase, the rationalized decision making process can be codified by making explicit for instance the architectural decisions taken and other options considered (and rejected). This codified knowledge can then easily be consulted throughout the remainder of the development project – and possibly even be reused in similar future projects.

In the remainder of this thesis, we examine architectural knowledge management from the perspective of architects and auditors. Whereas architects can be seen as principal producers of architectural knowledge, auditors can be regarded as principal architectural knowledge consumers. Although this distinction is not clear-cut, and both parties may act as producers and consumers, the difference in focal points leads to different needs in architectural knowledge management.

Two of the challenges from §5.1.3 are directly related to the remainder of this thesis. Architects need to exchange the architectural knowledge they produce, hence architectural knowledge sharing is a primary challenge for architects. This will be the subject of Part II. Auditors need to find the architectural knowledge produced by others, hence architectural knowledge discovery is a primary challenge for auditors. This will be the subject of Part III. The other two challenges are outside the scope of this thesis, and have been addressed by other researchers of the GRIFFIN project.

Part II

Supporting Architects

By Rik Farenhorst

6 Sharing Architectural Knowledge

The architecting process is very knowledge-intensive in nature and it is crucial to effectively support architects with sharing architectural knowledge. This introduction chapter of Part II briefly outlines the problems and challenges related to sharing architectural knowledge and introduces our research contributions in this area. To this end, it lists the central research questions, elaborates upon the research methodology followed, and introduces the studies presented in Part II.

6.1 Introduction

Software architects are knowledge workers. They are the central stakeholder in the architecture process, and responsible for making the key architectural decisions, communicating with both business and technical stakeholders, and safeguarding the project's success. Based on sound judgment, they are expected to constantly balance organizational requirements, technical requirements, and constraints while taking and communicating the architectural design decisions for the architecture design problems at hand. Supporting architects in these knowledge-intensive tasks calls for effective support for sharing architectural knowledge.

In research and practice there is a broad consensus on the importance of sharing architectural knowledge, in particular for reusing best practices, obtaining a more transparent decision making process, providing traceability between design artifacts, and recalling past decisions and their rationale. The obvious advancements in this architectural knowledge community notwithstanding, some scoping and maturation would be beneficial to both practitioners and researchers. This ensures that the methods, tools and techniques developed actually help in solving real-world problems.

The research in this part is based on case study research at RFA, one of the industrial partners of the GRIFFIN research. In Part I we explained that application of our core model of architectural knowledge (see Chapter 4) highlighted that the primary challenge for RFA related to architectural knowledge sharing. In this part we further zoom in on this challenge. By conducting several case studies at RFA, we iteratively built up understanding on how to effectively support architects in sharing architectural knowledge. We thereby took a broader perspective on knowledge sharing than the inter and intra-organizational knowledge exchange through models that is discussed in Chapter 4; we looked at person-to-person knowledge sharing and related schemes as well.

The remainder of this chapter is organized as follows. In §6.2 we explain the motivation for this research by arguing why sharing architectural knowledge is important. In §6.3 we formulate our research questions. In §6.4 we elaborate upon the research methodology followed, followed by an overview of research methods used in §6.5. In §6.6 we provide an overview of the chapters in this part and their relation to the research questions from §6.3. In §6.7, finally, we list the prior publications upon which this part has been based.

6.2 Problem Statement

Rus and Lindvall (2002) stated that "the major problem with intellectual capital is that it has legs and walks home every day". This statements holds as well for software architecting. Architects often work on a project-basis, and often reside at the customer organization instead of at the internal architecture department. When a project is finished they usually quickly move on to the next. Consequently, there is little time to meet and discuss with colleague architects who work elsewhere, let alone to share useful architectural knowledge with them.

Sharing architectural knowledge benefits architects in many ways, the most important being the prevention of knowledge vaporization. Knowledge pertaining to software architectures that is not explicitly stored eventually vaporizes (Bosch, 2004). The software system then turns into a 'black box', resulting in high maintenance cost and high degrees of design erosion. Knowledge vaporization also leads to additional costs made each time knowledge has to be transferred within the organization. Architects have to rethink the reasons why certain decisions have been made, or why errors occurred, and they have to repeatedly follow the same paths to achieve similar results.

Explicit knowledge can be shared more easily between different stakeholders. This brings us to three additional reasons for sharing architectural knowledge:

- **To foster reuse.** Every project is different and almost no architecture is the same. Nonetheless, in many cases certain best practices such as architectural styles, tactics or patterns apply to categories of applications or to specific domains. Sharing experience with colleagues or other stakeholders is thus something architects should do to reuse important architectural knowledge assets. This not only prevents architects to 'reinvent the wheel', it also improves the overall expertise of architects in the organization, and may even help to create a community of architects who can contact each other to exchange ideas, solutions, and other experience.

- **To promote learning and training.** If architectural knowledge is better accessible to or more effectively delivered to people, this information could be put to good use, such as learning and training. Especially junior architects who arrive fresh in an organization benefit from this. Instead of digging through endless file shares and feeling lost in their new surroundings, effective sharing mechanisms can give them fast access to important information sources (artifacts or people).

- **To increase collaboration.** In a decade where trends such as globalization, offshoring, virtual organizations and collaboration within closed and open communities emerge, sharing knowledge on software architectures is becoming increasingly important. Stakeholders are often located at different places, which makes exchanging knowledge face-to-face hard. Nevertheless, to stay up-to-date, share expertise, and exchange ideas, they have specific knowledge needs. Especially current large-scale software development and the complex architectural problems related to this calls for collaboration of multiple architects, developers, etc.

RFA is a large software development organization in which many architects are involved in architecting activities. For this organization, sharing relevant architectural knowledge is crucial for the reasons described above. RFA's primary goal is to develop high-quality software systems, and managers in this organization acknowledge the need for efficient collaboration between stakeholders and maintaining and increasing the know-how of architects by learning, training, and knowledge reuse.

Although there was a clear need for effective sharing of architectural knowledge, at the start of the GRIFFIN project we quickly discovered that the status quo at RFA was not really close to meeting this goal. Significant issues with respect to sharing architectural knowledge were brought to light. A tool for reusing architectural best practices was seldom used, architects seemed to know little of each other's interest, expertise or experience, and a structured approach for knowledge management in the architecting process did not exist. As a result, architects of RFA often 'reinvented the wheel' instead of resorting to the body of knowledge available.

Overcoming the above issues marked the starting point for a number of consecutive studies conducted at RFA that form the core of Part II of this thesis. The overall goal of these studies was understanding how to effectively support architects in sharing architectural knowledge.

van den Hooff and Huysman (2009) stress the need for sharing knowledge and discuss two main approaches to managing knowledge sharing: an emergent approach focusing on the social dynamics between organizational members and the nature of their daily tasks, and an engineering approach focusing on managing interventions to facilitate knowledge transfer. The emergent approach, the prevalent approach to modern-day knowledge management literature, emphasizes that knowledge sharing cannot be forced because it results from an intrinsic motivation that is personally, subjective and socially determined. The engineering approach, however, has its value as well, because it – although not directly – seems to stimulate and create conditions for this emergent process (van den Hooff and Huysman, 2009). Our contribution is in line with the engineering approach; by focusing on engineering useful and appealing methods, tools and technologies, we aim to (intrinsically) motivate architects to share architectural knowledge more frequently and more effectively.

6.3 Research Questions

Corresponding to the problem statement outlined in the previous section, our main research question is:

RQ- II.1 *How to effectively support architects in sharing architectural knowledge?*

An introduction to the concept of architectural knowledge is presented in Part I in which we have discussed what architectural knowledge is (RQ-I.1). In this part we focus on how architects can be effectively supported to share such knowledge. To answer this question properly we need to grasp what kind of knowledge-intensive activities architects are involved in and what kind of knowledge sharing support they need during these activities. This leads to the following two research questions, as also depicted by the arrows in Fig. 6.1.

RQ- II.2 *What do architects do that involves architectural knowledge sharing?*

RQ- II.3 *What do architects need with respect to architectural knowledge sharing?*

In order to fully understand what the main activities of architects are (RQ-II.2), we need to take into account the context in which they operate. This context is often relatively 'constant', and hard to change. Therefore, before we can propose any solutions

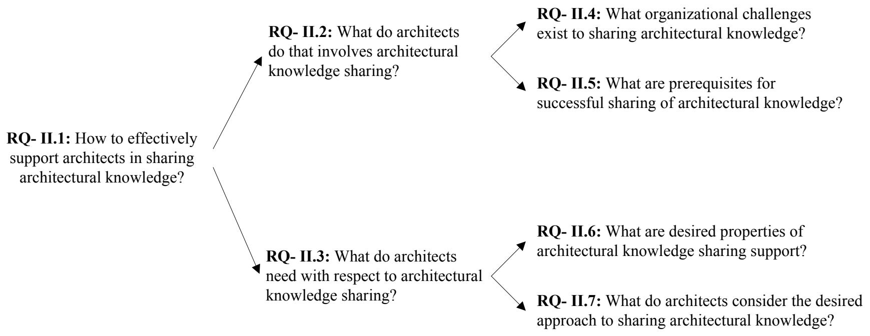

RQ- II.1: How to effectively support architects in sharing architectural knowledge?
RQ- II.2: What do architects do that involves architectural knowledge sharing?
RQ- II.3: What do architects need with respect to architectural knowledge sharing?
RQ- II.4: What organizational challenges exist to sharing architectural knowledge?
RQ- II.5: What are prerequisites for successful sharing of architectural knowledge?
RQ- II.6: What are desired properties of architectural knowledge sharing support?
RQ- II.7: What do architects consider the desired approach to sharing architectural knowledge?

Figure 6.1: Research questions of Part II

to effectively share architectural knowledge, it is important to sketch the boundaries in which we have to operate. Identifying these boundaries involves understanding organizational challenges related to sharing architectural knowledge. Based on these insights we aim to discover what organizational or technical prerequisites exist for successful sharing of architectural knowledge. We derive two research questions from RQ-II.2:

RQ- II.4 *What organizational challenges exist to sharing architectural knowledge?*

RQ- II.5 *What are prerequisites for successful sharing of architectural knowledge?*

Based on our understanding of what architects do and what challenges exist to sharing architectural knowledge, we focus on what we could do to make their life a bit easier in this respect (RQ-II.3). To fully understand what kind of (tool) support architects need we start with identifying desired properties such support should have. Based on a more elaborate investigation in practice we also elicit what architects consider the desired approach to architectural knowledge sharing. This prevents us from designing a solution that does not match the expectations of practicing architects. From RQ-II.3 we thus derive the following two research questions:

RQ- II.6 *What are desired properties of architectural knowledge sharing support?*

RQ- II.7 *What do architects consider the desired approach to sharing architectural knowledge?*

Summarizing the above, the answer to our main research question (RQ-II.1) is found by unifying the answers to a number of derived research questions central to this part (RQ-II.2 - RQ-II.7). The relations between these research questions are also illustrated in Fig. 6.1.

6.4 Research Methodology

This research was conducted mainly at RFA, one of the industrial partners of the GRIFFIN consortium. As discussed in §6.2 RFA's main problem was that sharing architectural knowledge was implemented quite ineffective and in an ad hoc matter. Between 2005 and 2009, we have worked together with architects of RFA to further analyze the status quo with respect to architectural knowledge sharing. During a number of studies we proposed, implemented and evaluated several improvements.

Our qualitative research over the past four years closely resembles a typical action research cycle. Action research is "grounded in practical action, aimed at solving an immediate problem situation while carefully informing theory" (Baskerville, 1999). A particular strength of the qualitative nature of action research is its value in explaining what goes on in organizations (Avison et al., 1999), which is exactly what we needed to do in RFA.

In its most simple form action research consists of a diagnostic and a therapeutic stage, but a more widely practiced and reported form is called canonical action research (CAR), which is based on a five stage cyclic model presented by Susman and Evered (1978). This model consists of five main stages:

1. **Diagnosing.** Identification of primary problems that are the underlying causes of the organizations desire for change. In this phase certain theoretical assumptions about the nature of the organization and its problem domain are developed.

2. **Action planning.** Specification of organizational actions that should relieve or improve the primary problems. In this phase a theory is formulated about how to address the earlier identified problems.

3. **Action taking.** Researchers and practitioners collaborate in the active intervention into the client organization, causing certain changes to be made.

4. **Evaluating.** Researchers and practitioners evaluate the outcomes of the action taken. This evaluation includes determining whether the theoretical effects of the action were realized, and whether these effects relieved the identified problems.

5. **Specifying learning.** The restructuring of organizational norms or theory to reflect the new knowledge gained by the organization during the research. This phase may also indicate further action research interventions that are worth diagnosing. While the activity of specifying learning is formally undertaken last, this reflection phase is usually an ongoing process that plays a role throughout all other four phases.

The specifying learning phase is a particularly important final phase, because it resembles the *double-loop* learning approach of Argyris and Schön (1978), as opposed to *single-loop* learning in which not the organizational norms but only the action strategy is adjusted (e.g., “let’s use yet another tool for...”). *Double-loop* learning is more creative and reflexive in nature, which lends itself much better to practitioners or organizations who need to make informed decisions or define proper strategies in rapidly changing and often uncertain contexts (Argyris and Schön, 1978). Therefore, for helping RFA with defining effective strategies for sharing architectural knowledge, enabling *double-loop* learning by a reflective specifying learning phase is crucial.

Between 2005 and 2009 we have conducted four case studies in RFA and a fifth study in order to validate a theory that was constructed based on these case studies. Each individual case study lasted around five months, and had its own scope, research questions, and methodology. When looking at the bigger picture, however, these case studies clearly resemble the aforementioned phases of an action research cycle. A schematic overview of this cycle is depicted in Fig. 6.2. In the following subsections we give an overview of each study in turn and explain to which action research phase it corresponds.

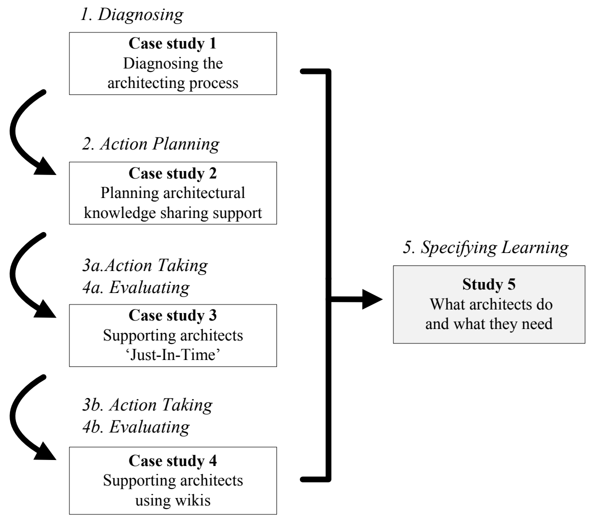

1. Diagnosing
Case study 1
Diagnosing the architecting process
2. Action Planning
Case study 2
Planning architectural knowledge sharing support
3a.Action Taking
4a. Evaluating
Case study 3
Supporting architects ‘Just-In-Time’
3b. Action Taking
4b. Evaluating
Case study 4
Supporting architects using wikis
5. Specifying Learning
Study 5
What architects do and what they need

Figure 6.2: Action research cycle

6.4.1 Case study 1: Diagnosing the architecting process

In the first case study, which was conducted early 2006, we explored the status quo in RFA with respect to architectural knowledge sharing. This study corresponds to a diagnosis phase of an action research cycle, in which the main problems and points for improvement are identified. In this case study we mainly focused on the architect's activities and his roles and responsibilities in the architecting process.

We quickly found that most mechanisms in place at RFA to share knowledge were rather ineffective. For example an existing system for storing architecture guidelines and best practices was barely used because architects deemed its added value rather low. In order to understand what caused the problems with these tools and to indicate some points for improvement, we zoomed in on four specific aspects of RFA's architecting process: the architecting environment the architects operate in, the decision making process of these architects, the architecture descriptions they produce, and the various stakeholder roles and responsibilities.

While studying the architecting process of RFA, our first discovery was that architects do much more than making design decisions or creating architectural descriptions. We found that architects are also responsible for communication with internal and external stakeholders, which showed itself through workshops, coffee room chats, requirements negotiations, etc. In addition, many projects require some quality monitoring of architects, during which they review existing architectural documentation, verify whether architectural guidelines or rules are adhered to, check whether an architecture solution conforms to reference architectures of customer organizations, and during which they assess the feasibility of the architectural solution.

The second main discovery during this case study was an apparent lack of motivation of architects in RFA with respect to pro-actively sharing architectural knowledge. It became clear that part of the reason for this was the fact that available knowledge management tools had a poor usability, steep learning curve and quickly outdated contents. Based on our diagnosis we defined a number of prerequisites for successfully sharing architectural knowledge. The importance of these prerequisites is motivated by demonstrating that they create some necessary incentives for architects to share architectural knowledge.

6.4.2 Case study 2: planning architectural knowledge sharing support

The second case study at RFA was conducted early 2007 and focused on establishing properties that characterize effective architectural knowledge sharing tool support. This case study makes for a typical action planning phase of an action research cycle, because in this study we formulated a theory to address the diagnosed problems of the previous case study.

We looked into knowledge management and software architecture literature to see what kind of dos and don'ts exist with respect to architectural knowledge sharing support. In parallel we evaluated existing tools in RFA to identify strengths and weaknesses. We culminated all insights into a number of desired properties for architectural knowledge sharing tools.

One of the main insights gained in this case study is that architects benefit from a so-called 'hybrid' strategy in sharing architectural knowledge. This means that a proper combination between a codification and personalization strategy needs to be found (Hansen et al., 1999). Other desired properties for architectural knowledge sharing tools we identified include support for stakeholder-specific content, easy manipulation of content and collaboration.

We used the above insights to design and implement a prototype web portal that adheres to all desired properties for sharing architectural knowledge. This part of the case study corresponds well to the description of an action plan in which the formulated theory is made concrete and executable. The 'action taking' phase itself, in which we used and evaluated the portal at RFA's architecting process, was part of the next case study.

6.4.3 Case study 3: Supporting architects 'Just-in-Time'

The third case study at RFA was conducted mid-2007. In this case study we used and tested the web portal developed in the previous study to improve the status quo at RFA with respect to architectural knowledge sharing. We evaluated with a number of architects what they liked and disliked about the portal. This case study is therefore a combination of the 'action taking' and 'evaluation' phases of action research.

Before we started testing our portal we examined in more detail existing tools for architectural knowledge sharing at RFA. We elicited from the architects what the limitations are of these existing tools and which requirements they have for a potentially new environment. This requirement elicitation was a continuation of the diagnosis we conducted during our first case study (cf. §6.4.1), but also a validation of the theory about effective tool support we established in our second case study (cf. §6.4.2).

The requirements elicitation part of this case study showed that architects need effective support for searching relevant information, managing documentation, and assistance to easily get in touch with colleagues or other stakeholders. When we aggregated these insights, we found that RFA's architects were apparently in need of *'Just-in-Time'* architectural knowledge, which we defined as access to and delivery of the right architectural knowledge, to the right person, at any given point in time. Increasing the access to and delivery of architectural knowledge will boost reuse of architectural knowledge, stimulates learning among architects, and assists in assessing quality (e.g., by pointing architects to reference architectures, best practice databases, and quality frameworks).

The testing of our web portal indicated that architects perceive it as a definite improvement over the existing tools at RFA. The fact that several community building – or personalization – mechanisms (e.g., yellow pages, discussion boards) were combined with codification techniques (e.g., repositories, project environments) offered the architects an all-round platform to share architectural knowledge in various ways. The intuitive user interface further helped motivate architects to actually use these tools, i.e., create the necessary incentives for sharing architectural knowledge.

The strengths of our portal notwithstanding, during our evaluation the architects also indicated a number of possible improvements. Interestingly, the architects did not ask for new kind of functionality or features, but instead wished for more elaborate support for some of the core parts of our portal, including more focus on collaboration, communication and integration with other tools. We decided to take this feedback seriously and planned for another case study of action taking and evaluation to address these requirements.

6.4.4 Case study 4: Supporting architects using wikis

The fourth case study at RFA was conducted in the first half of 2008. In this case study we explored the applicability of wikis as architectural knowledge sharing environment. Wikis are strong in community building and in supporting collaboration, and during the previous case study we found that architects needed more extensive support in these areas. In terms of placing this case study in an action research cycle it is similar to the previous one; the use and evaluation of a tool environment corresponds well to the ‘action taking’ and ‘evaluation’ phases of action research.

In this case study, we learned that wikis have several characteristics that make them suitable as knowledge sharing environment in the architecting process. Facilities to hold discussions, group editing features and the very intuitive user-interface make wikis a very lightweight yet versatile platform. In addition, the commercial enterprise wiki we tested offers various integration mechanisms to other tools (such as Sharepoint websites) and RFA’s file shares, which was greatly appreciated by the architects and was considered a clear improvement over the portal we had shown to them in the previous case study. They now finally had a single point of entry to a web of knowledge at their disposal, which easily combines management of very specific architectural knowledge (e.g., design decisions, best practices, patterns), but also related knowledge crucial for any project (project documentation, progress reports, handbooks).

Wikis are also quite suitable in creating a 'community of architects' in which everybody knows from each other where certain expertise, or experience resides. This helps architects in various activities such as conducting reviews, writing reports, or communication with stakeholders. The enterprise wiki we tested and evaluated offers various plugins which add various interesting features, such as modeling support, generation of statistics and overviews, and even text mining features in combination with database integration. We argue that these aspects can further support architects in their daily work by offering 'smart' or pro-active support when needed, in a non-intrusive way (e.g., automatically identifying patterns or rules from raw wiki content).

In this case study we found that wikis are able to create incentives for architectural knowledge sharing, because in a short amount of time a lot of architectural knowledge was contributed by the architects who participate in a pilot. We did learn, however, that certain so-called 'wiki patterns' need to be kept in mind to prevent things going out of hand (Mader, 2007). Examples are the appointment of wiki 'gardeners' who keep the wiki organized, or the establishment of templates to enable uniform codification of specific types of knowledge. These insights showed that a versatile tool in itself is not enough; process rules and guidelines are needed to make effective use of the tool and to position it correctly in the architecting process.

6.4.5 Study 5: What architects do and what they need

By combining insights gained during our all four case studies we have created a theory of what architects do and what kind of support they need for sharing architectural knowledge. This theory building step corresponds to the specifying learning stage, which denotes the ongoing process of documenting and summing up the learning outcomes of the action research cycle.

To validate our theory, in this final study that was conducted early 2009 we conducted large-scale survey research in four IT organizations in the Netherlands, including RFA. In total 279 practicing architects were asked about their daily activities, and how important they consider the various types of support for sharing architectural knowledge.

Our survey results showed that architects do make lots of architectural decisions, but neglect documenting them. Producing and subsequently sharing of architectural knowledge is clearly not one of their most popular activities. While in itself not very surprising, one would expect that support to help architects with this latter task is something they appreciate. The opposite is true. Architects rather stay in control and are not interested in automated or intelligent support. When it comes to consumption of architectural knowledge, however, support for effective retrieval of (stored) architectural knowledge is on the top of their wish list. This apparent contradiction suggests architects are rather lonesome decision makers who prefer to spend their working life in splendid isolation.

This study showed that for architectural knowledge sharing support to be effective, this support should stimulate architects in both producing and consuming architectural knowledge. Integrated, 'all-round', tools environments that implement a variety of architectural knowledge sharing use cases, seem promising because they could provide a balance between production and consumption of knowledge in the architecting process.

6.5 Research Methods

Software engineering research is often qualitative in nature (Glass et al., 2002) and ours is no exception. A difference, however, is that software engineering research usually does not consider social and organizational aspects. As indicated in §6.4 the GRIFFIN consortium enabled a tight collaboration between researchers and practitioners, making the choice to conduct action research a logical one.

Action research is more of a holistic approach to problem-solving, rather than a single method for collecting and analyzing data (O'Brien, 1998). It allows for various methods to be used, which are generally common to the qualitative research paradigm. Holz gives an overview of research method being used in computing (Holz et al., 2006). From this list, we list the ones applicable to our action research below. Table 6.1 depicts an overview of these research methods and shows to which action research phase they belong and which research questions we have answered using these methods.

Table 6.1: Overview of research methods

Study	**Action research phase**	**Research methods**	**Research questions**
1: Diagnosing the architecting process	Diagnosing	Case study Document analysis Exploratory survey Interviews Critical analysis of the literature	RQ- II.4, RQ- II.5
2: Planning AK sharing support	Action planning	Case study Interviews Critical analysis of the literature	RQ- II.6
3: Supporting architects 'Just-In-Time'	Action taking & evaluation	Case study Interviews Prototype experiments	RQ- II.7
4: Supporting architects using wikis	Action taking & evaluation	Case study Interviews Prototype experiments	RQ- II.7
5: What architects do and what they need	Specifying learning	Survey	RQ- II.2, RQ- II.3

- **Case studies.** As mentioned before we have conducted a series of four case studies in RFA. We use a standard definition of case study here: "the collection and presentation of detailed information about a particular participant or small group, frequently including the accounts of subjects themselves" (Holz et al., 2006). Key property of case study research is that it is exploratory in nature, which lends itself well for action research activities. We did not focus on the discovery of a universal, generalizable truth, nor did we typically look for cause-effect relationships. Instead, emphasis is put on exploration and description. As explained in §6.4 the exact scope and focus of each of the four case study was different.

- **Document analysis.** In order to retrieve more information on how architectural knowledge was captured in RFA in the first case study we used document analysis. This included examination of system software and documentation, technical papers, architectural standards, reference material, etc. During this analysis we gained initial insight into the terminology used and the kind of architectures that were developed.

- **Exploratory survey.** In contrast to a normal (or full) survey, this research method aims at conducting an exploratory field study in which there is no test of relationships between variables (Holz et al., 2006). In the first case study we used an exploratory survey about use cases for architectural knowledge.

- **Interviews.** This is a research method for gathering information by posing questions to people, using an interviewer. Interviews can be structured or unstructured with respect to the questions asked or the answers provided. We used semi-structured interviews in all four case studies. This means that a set of questions was made in advance and placed in a logical order by the interviewer, in order to have a high-level structure to steer the conversation. However, based on the answers of the interviewee sufficient freedom exists to ask follow-up questions or to focus on new unexplored topics.

- **Critical Analysis of the Literature.** This method is defined as "an appraisal of relevant published material based on careful analytical evaluation." (Holz et al., 2006). Although adopting an explicit knowledge management perspective to software architecting is rather novel, we were able to draw upon existing literature about knowledge management in software engineering, instead of having to start from scratch. Although relevant literature was studied during all five studies, a more critical review of relevant literature was done in the first and second case study.

- **Prototype Experiments.** In the third and fourth case study we constructed prototype tooling (respectively a web portal and a wiki) to support sharing of architectural knowledge. In both studies we have validated the added value of these tools by asking architects to experiment with these tools and to provide feedback. The prototype experiments we conducted involved using and evaluating the tools by practicing architects in a real-life setting, as opposed to experiments conducted in a laboratory setting.

- **Survey.** In our fifth study we conducted a large scale survey in which 279 architects participated. The survey was carefully designed and contained all standard phases and analyses of proper survey research, including theory formulation, survey design, question formulation, scale construction, hypothesis formulation and testing (cf. (Kitchenham and Pfleeger, 2001-2002)).

6.6 Outline of Part II

The following five chapters correspond to the five studies mentioned earlier. Taken together they form the action research cycle as depicted in Fig. 6.2 and they help answering the research questions listed in §6.3.

- In Chapter 7 we identify a number of organizational challenges related to architectural knowledge sharing (RQ-II.4). Based on this diagnosis we define a number of prerequisites for sharing architectural knowledge (RQ-II.5).

- In Chapter 8 we identify the desired properties of architectural knowledge sharing tools (RQ-II.6) by studying typical characteristics of architecting and what knowledge management techniques can be applied to the software architecture domain. A prototype web portal is designed that has all these properties.

- In Chapter 9 we investigate what architects consider the desired approach or strategy for sharing architectural knowledge (RQ-II.7). Based on these insights we experiment with the Just-in-Time web portal for architectural knowledge. This portal is constructed based on the design made in the previous chapter.

- In Chapter 10 we continue to elicit what architects need (RQ-II.7) by exploring the applicability of wikis as architectural knowledge sharing environment. We experiment with an enterprise wiki for architectural knowledge as improvement over the portal of the previous chapter.

- In Chapter 11 we identify what architects do and what they need with respect to sharing architectural knowledge. A theory of architecting activities (RQ-II.2) and support methods for sharing architectural knowledge (RQ-II.3) is constructed based on the earlier case studies, and this theory is validated through a survey.

To conclude Part II, in Chapter 12 we revisit the research questions addressed during the five studies of our action research cycle (RQ-II.2 - RQ-II.7). Together these research questions answer our main research question (RQ-II.1). In Chapter 12 the main contributions of Part II are culminated in a number of lessons learned.

6.7 Publications

Most of the research presented in Part II has either been published previously or is currently in press. The chapters in this part are based on the following publications.

Parts of Chapter 7 have been published as:

- Farenhorst, R. Tailoring Knowledge Sharing to the Architecting Process. *ACM SIGSOFT Software Engineering Notes*, 31(5), 2006.
- Farenhorst, R., P. Lago, and H. van Vliet. Prerequisites for Successful Architectural Knowledge Sharing. In *18th Australian Software Engineering Conference* (ASWEC 2007), pages 27–36, IEEE Computer Society, 2007.

Parts of Chapter 8 have been published as:

- Farenhorst R., P. Lago, and H. van Vliet. Effective Tool Support for Architectural Knowledge Sharing. In *1st European Conference on Software Architecture* (ECSA 2007), pages 123–138, Springer, 2007
- Farenhorst R., P. Lago, and H. van Vliet. EAGLE: Effective Tool Support for Sharing Architectural Knowledge. *International Journal of Cooperative Information Systems*, 16(3&4): 413–437, World Scientific Publishing Company, 2007.

Parts of Chapter 9 have been published as:

- Farenhorst R., R. Izaks, P. Lago, and H. van Vliet. A Just-In-Time Architectural Knowledge Sharing Portal. In *7th Working IEEE/IFIP Conference on Software Architecture* (WICSA 2008), pages 125–134, IEEE Computer Society, 2008.

Parts of Chapter 10 have been published as:

- Farenhorst, R. and H. van Vliet. Experiences with a Wiki to Support Architectural Knowledge Sharing. In *3rd Workshop on Wikis for Software Engineering* (Wikis4SE'08), Porto, Portugal, 2008.

Parts of Chapter 11 have been published as:

- Farenhorst, R., J.F. Hoorn, P. Lago, and H. van Vliet. What Architects Do and What They Need to Share Knowledge. *Technical Report IR-IMSE-003*, VU University Amsterdam, April 2009.
- Farenhorst, R., J.F. Hoorn, P. Lago, and H. van Vliet. The Lonesome Architect. In *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture* (WICSA/ECSA 2009). In press.

Finally, parts of the lessons learned and research methodology overview of this chapter have been published as:

- Farenhorst, R. and H. van Vliet. Understanding How to Support Architects in Sharing Knowledge. In *4th Workshop on SHAring and Reusing architectural Knowledge* (SHARK'09). IEEE Computer Society, 2009.

7 Organizational Challenges to Sharing Architectural Knowledge

In order to successfully share architectural knowledge, some organizational challenges need to be overcome. In this chapter we elaborate on these challenges by reporting on our first case study at RFA. Based on this diagnosis we define a number of prerequisites that need to be met to promote sharing of architectural knowledge. The importance of these prerequisites is motivated by demonstrating that they create some necessary incentives for sharing architectural knowledge.

7.1 Introduction

In recent literature warnings are presented for the fact that not all knowledge sharing implementations are successful automatically. Ghosh (2004) lists several factors that make knowledge sharing difficult, such as the fact that knowledge sharing is time consuming, and that people might not trust the knowledge management system. Another warning is that it is infeasible to strive for completeness. Lakshminarayanan et al. (2006) note that it is impossible to create a tool to reason about all the issues that an architect would normally need to consider. Finally, we should be aware of the fact that a lot of the available knowledge cannot be made explicit at all, but instead remains tacit in the minds of people (Nonaka and Takeuchi, 1995). Sharing this tacit knowledge is very hard, as stated by Haldin-Herrgard (2000).

The above warnings clearly indicate that successful architectural knowledge sharing involves more than just choosing the right software tool. We argue that successful architectural knowledge sharing can only be achieved if the necessary incentives are created. These incentives induce stakeholders to share valuable architectural knowledge, such as the major design decisions made, the underlying rationale for these decisions, and alternatives that were considered.

To find empirical evidence for the warnings sketched above, we conducted a first case study at RFA to study the state of the practice with respect to sharing architectural knowledge. In this chapter we report on this study, in which we explored current methods for sharing knowledge and the issues related to these methods. We quickly discovered that these issues were not mere technical in nature; various organizational challenges exist that need to be overcome to guarantee successful sharing of architectural knowledge.

To identify RFA's challenges with architectural knowledge sharing, we zoomed in on four specific aspects of its architecting process: the architecting environment the architect operates in, the decision making process of these architects, the architecture descriptions they produce, and the various stakeholder roles and responsibilities. We did so by conducting interviews, an exploratory survey and document analysis. In addition, we reviewed knowledge management literature to acquire insight on sharing knowledge in such an organizational context.

Our diagnosis of RFA's architecting process revealed a number of issues related to sharing architectural knowledge in this organization, which suggested that knowledge sharing practices in this organization were rather immature. We investigated how to improve this situation and identified a number of prerequisites for sharing architectural knowledge. We underline the importance of these prerequisites by demonstrating that they create incentives to architects for sharing knowledge.

The remainder of this chapter is organized as follows. In §7.2 we use insights gained from literature to identify a set of incentives for architectural knowledge sharing. §7.3 describes the case study conducted including the research methods used for our diagnosis of the architecting process of RFA. Based on this diagnosis, in §7.4 we discuss a number of issues pertaining to architectural knowledge sharing. In §7.5 we propose a set of prerequisites for sharing architectural knowledge, and demonstrate how these prerequisites create the necessary incentives for architectural knowledge sharing. Finally, §7.6 concludes this chapter by revisiting the research questions answered by this case study.

7.2 Incentives for Sharing Architectural Knowledge

In the architecting phase of software development, a lot of implicit and explicit knowledge is required to take appropriate architectural design decisions. This knowledge is usually scattered across the organization. Various stakeholders, such as software developers, architects, project managers, and maintainers, all possess knowledge that can be of value when developing the software architecture. We assert that sharing this knowledge between these stakeholders leads to a number of ‘intrinsic’ benefits. If stakeholders would recognize these benefits, it would induce them to share valuable architectural knowledge. In this sense, these benefits act as incentives for architectural knowledge sharing.

Based on insights gained from related knowledge management literature (Argote, 1999; Bresman et al., 1999; Cummings, 2003; Ghosh, 2004; Haldin-Herrgard, 2000), as well as our own experience in software architecture research, we have identified three main incentives for architectural knowledge sharing. We argue that these incentives need to be created in order to promote successful architectural knowledge sharing. Below, they are elaborated in turn.

1. **Establishment of social ties.** Research has shown that if stakeholders trust each other and the ‘competition’ between them is minimized, the motivation for cooperation and sharing knowledge is higher (Argote, 1999). Moreover, individuals will only participate willingly in knowledge exchange once they share a sense of identity or belonging with their colleagues (Bresman et al., 1999). Sharing knowledge with these colleagues may then further increase the trust between them, thereby improving relationships. Establishing social ties, either by an increase of trust or by less competition between stakeholders, therefore acts as an incentive for knowledge sharing.

   This incentive for knowledge sharing also applies to the software architecture domain. When typical stakeholders in the architecting process, such as project managers, developers, or architects, collaborate in the decision making process and exchange ideas or expertise, the social ties between them will become stronger.

2. **More efficient decision making.** According to Cummings (2003) a group’s performance increases when everyone in this group is informed of each other’s expertise. This is because such knowledge allows groups to engage in joint brainstorming sessions in which group members are able to explore new ideas and discuss difficult issues.

   This increase in group performance is something architects in a software architecture environment benefit from. For software architects it is often very important to not only get feedback from colleague architects, but also from developers or maintainers when developing the software architecture. Developers may have a lot of expertise and their insights and experience can benefit the architects in an early stage. The same holds true for maintainers that may have gathered knowledge on the actual pros or cons of past decisions. Architectural knowledge sharing creates a feedback loop between architects and other stakeholders in the architecting process. Having such a feedback loop greatly improves the efficiency of the decision-making process and is therefore an incentive for architectural knowledge sharing.

3. **Knowledge internalization.** When actively sharing knowledge, people are able to communicate valuable experience with each other. This experience is then stored as tacit expert knowledge in their minds. This process is called knowledge internalization (Nonaka and Takeuchi, 1995). Knowledge internalization refers to the degree to which a recipient obtains ownership of, commitment to, and satisfaction with the transferred knowledge. When knowledge is fully internalized by a recipient, it becomes theirs. Consequently, tacit knowledge increases the quality of the work and makes the work go smoothly (Haldin-Herrgard, 2000), and the more knowledge is shared, the more knowledge can be internalized by practitioners.

   Knowledge internalization greatly assists software architects in their daily practice. Having internalized valuable experiences aids them in avoiding sub-optimal solutions, but also enables them to learn from mistakes, such as ‘bad’ design decisions or architectures that have been developed in the past but now cause headaches. As a result, the software architectures that are being developed are based on a set of decisions that is more carefully considered, and therefore have a higher overall quality.

7.3 Diagnosing the Architecting Process

In the first case study at RFA, during which we were actively involved with architects and other stakeholders, we diagnosed how architectural knowledge is shared in practice. Although this organization acknowledges the importance of software architectures, discussions with architects and developers indicated that the organization struggles with how software architecture development - using methods, techniques, and tools - can best be fit in the overall development process, and how architectural knowledge can best be shared and (re)used.

Within the architecture department of RFA a tool is used that guides the architects in the architecting process. This tool was specifically designed to support architects in designing a software architecture, i.e., in taking the right architectural design decisions. To this end it contains of a set of guidelines to select the appropriate architectural solution based on the input provided by the architect. The input of this tool consists of answers to a number of questions that mostly focus on quality aspects of the system to be developed. The output of the tool is a generated document that is then further refined into a final version of the architecture description of the system.

One of the main triggers for conducting this case study was the fact that the aforementioned tool is seldom used by architects. Although envisioned as a suitable means to share experiences and best practices related to software architectures, apparently architects do not perceive its added value. Our main goal was to find the underlying reason for the lack of success of this tool, by focusing on the architecting process as a whole.

We used three main research methods for our diagnosis: an exploratory survey containing several use cases for architectural knowledge, a documentation study of standards, best practices and architectural descriptions, and finally a set of open interviews with various stakeholders of the architecting process.

- **Exploratory survey.** We used an an exploratory survey consisting of 27 potential use cases for architectural knowledge. A more elaborate description of these use cases is given in (van der Ven et al., 2006a). We asked 15 architects how important they consider these use cases, by letting them give scores on a scale from 1 to 5. We ranked the use cases by counting the total scores they received from the 15 architects. Among the five use cases that rank highest are three use cases that are clearly devoted to sharing architectural knowledge:

  – *Take a design decision based on explicit concerns of a stakeholder.*
  – *Explain to stakeholders (the impact of) a design decision that is taken.*
  – *Keep stakeholders up-to-date on a certain (new or changed) design decision.*

  This indicates that there is a clear interest in sharing architectural knowledge within this organization.

- **Documentation study.** In order to retrieve more information on how architectural knowledge is currently shared, we examined available documentation such as architectural standards and best practices, example architecture documents, and related functional and technical system documentation. During this documentation study we gained initial insight into the terminology used and the kind of systems that are developed.

- **Interviews.** We held 17 semi-structured interviews with various kinds of stakeholders, among which were architects, developers, maintainers, project managers, and software testers. The interviews were open and focused on how the interviewees perceive architectural knowledge in the organization. Since the interviewees were asked to elaborate their answers the interviews provided us with detailed information on how architects and other stakeholders use architectural knowledge, what their knowledge need is, and what the current limitations are regarding architectural knowledge sharing.

Using the survey, documentation study, and interviews as primary sources, we collected an extensive amount of information about the architecting process of RFA. We structured the collected information using four perspectives, based on a schematic outline of the architecting process that was presented in Chapter 4, Fig. 4.4. In that figure the stakeholders are positioned centrally. This corresponds to the notion that a software architecture should be stakeholder-driven. The stakeholders, in particular the architects, take decisions according to their roles. These decisions can be influenced by environmental events, i.e., events that are relevant but not always explicitly recognized as being of influence to software development, such as budget constraints. The decisions are reflected in design artifacts and architectural descriptions, e.g., documented views, but also 'lower-level' artifacts such as source code. A stakeholder responsible for an artifact changed due to a decision might find a need to take subsequent decisions. The responsibilities are defined by the stakeholders' role(s).

Based on the schematic outline of the architecting process, we define four perspectives that provide us with information on how software architecting is implemented in an organization. These perspectives focus on the environment, the decision-making process, architectural descriptions, and stakeholders' roles and responsibilities respectively.

1. The **architecting environment** perspective, which targets the broad context in which the software architectures are developed.

2. The **decision making** perspective, which puts the architectural design decisions and their rationale central.

3. The **architectural descriptions** perspective, which focuses on the design artifacts that are constructed.

4. The **stakeholder roles and responsibilities** perspective, which examines the stakeholders and their roles.

Viewing RFA's architecting process from these four perspectives allowed us to retrieve information specific to important aspects of the architecting process, and to capture the recursive nature of this process, which is described by Hofmeister et al. (2007). Other approaches tend to focus on either the decision making process, without explicitly looking at the process in which the decisions are being taken (e.g., (Kunz and Rittel, 1970) and its descendants), or on the design artifacts (e.g., (IEEE 1471)).

In the remainder of this chapter we use our observations with respect to architectural knowledge sharing at RFA to determine a set of prerequisites that address the knowledge sharing issues. In the following subsections we summarize the findings by focusing on each of the four different perspectives in turn.

7.3.1 Architecting environment

Architect: "Even though it was clear that the system's performance was worse when using this type of database connections, the customer strictly referred to its reference architecture that forbids any alternatives. Further discussion on this issue was not possible."

RFA has a large organization as its main customer. This customer organization has its own architecture department that has strict control over the system development. Based on this customer organization's reference architecture, constraints have been defined for individual systems. Often there is little room for deviating from these constraints.

Because of the existing systems that are already in place in the customer organization, little or no systems are developed based on a 'greenfield' situation. Often, new systems have to replace or extend an existing system. RFA had developed a knowledge repository to guide architects in creating an architecture, but this repository is mainly intended for 'greenfield' system development. The guidelines used by the repository do not comply with the reference architecture of the main customer. Architects at RFA acknowledged that if crucial architectural knowledge about the reference architecture is added to the repository, the quality and efficiency of decision making will increase. Nevertheless, as of yet no effort has been taken to implement this. Therefore, when using the repository, chances are that architectural solutions proposed are in conflict with the constraints defined by the customer.

One can argue that among the most interesting architectural knowledge worth sharing is that of the customer organization. The fact that this is not done in the current repository is one of the main reasons for architects not to use it. This was confirmed in interviews held with four different architects.

7.3.2 Decision making

Developer: "It is often the case that me and my colleagues take the major design decisions, while the architect's main function is that of a scribe. The architecture documentation he creates reflects the decisions we have taken."

We found out that currently in RFA the architects do not have overall control over the decision making process. Developers often possess more technical expertise and are more up-to-date with the functional and technical constraints posed by the customer. Therefore, they make most of the technical design decisions. In addition, since the customer organization dictates certain rules that have to be adhered to, already several decisions have been made at the start of the architecting process. Architects and developers maintained in the interviews that they have only limited freedom in taking design decisions. They decide mainly on more localized and less cross-cutting issues, such as the kind of development tools to be used.

Above observations are in contrast with the statement that the architect is taking the main design decisions (Eeles, 2006b). Architects in RFA do not create much architectural knowledge to share with other stakeholders; they act more as mediators between the customer organization and the development teams. Architects are responsible for the overall quality of the system to be built, and therefore need to solve conflicting decisions, make the right tradeoffs, and check whether development teams comply to the architecture. Therefore, for them it is important that knowledge created within the customer organization or within the development teams is shared. Both project managers and architects confirmed in the interviews that this feedback loop would allow architects to build up expertise, gain practical insights, and define best practices for future software architectures.

7.3.3 Architectural descriptions

Maintainer: "Up until this interview I didn't know there existed an architecture description of the system."

Not all stakeholders agree with the level of detail that is to be used in the views of the architectural description. Developers and maintainers indicated in the interviews that they consider most of the views too high-level. Project managers on the other hand think the architectural descriptions are clear and nicely suited for non-technical people as well.

Architectural knowledge is not only found in the standard architecture documentation, but also scattered around in technical and functional design documents used by developers and maintainers. Many interviewees stressed the need for collaboration and sharing important information between architects, developers and maintainers.

Although some views in the architecture description are targeted at the maintainers, maintainers are unaware of the information that is described for them. In a meeting with five maintainers it became clear that none of them had seen the architecture description of the system they were maintaining before.

To sum up, architectural descriptions can be a powerful means to summarize and share essential information on the system that is developed. However, a necessary precondition then is that the granularity of the information matches the expectations and the knowledge needs of the receiver. In this case study we observed that this is currently not the case in RFA. Developers and maintainers consider the architecture description too high-level, whereas the architects are under the impression that specific viewpoints are of the right level of detail. We thus believe that knowledge sharing could be more effective if the information need of the stakeholders involved is used as a basis when describing the architecture.

7.3.4 Stakeholder roles and responsibilities

Architect: "Basically, the architecture description is finished as soon as the project manager and lead developers give their approval. After this, my job ends and I am no longer responsible for keeping the architectural description up-to-date."

In RFA architects are not responsible for maintaining the architecture document after the system has been developed. Since no other 'owners' of this document exist, it soon becomes outdated. This is one of the reasons for the fact that architectural descriptions are not often read by developers and maintainers.

The architect does not fulfill a prescriptive role in the architecting process that allows him to take all major architectural design decisions, but rather a descriptive one in which he is the mediator between the customer organization and the technical stakeholders. However, the knowledge repository introduced in §7.3.1 is designed to be prescriptive, and supports architects in making decisions. Because of this mismatch, the repository turned out to be unpopular among architects and seldom used.

7.4 Issues Related to Sharing Architectural Knowledge

Based on the results described in the previous section, we conclude that the available mechanisms for sharing architectural knowledge are not optimally implemented at RFA. From the case study results we have distilled six issues related to sharing architectural knowledge. Fig. 7.1 depicts a object-interaction diagram of RFA's current architecting process that was constructed using the FRS method introduced in Chapter 4. This diagram helps visualizing the issues by highlighting the five different stakeholders of the architecting process and how they communicate. The rectangular elements represent entities; the elliptical ones denote actions on or with these entities. Arrows indicate the creation of new instances of an entity. Actions and entities that are linked together can be read as sentences, e.g "An architect takes an architectural design decision". The six issues identified are:

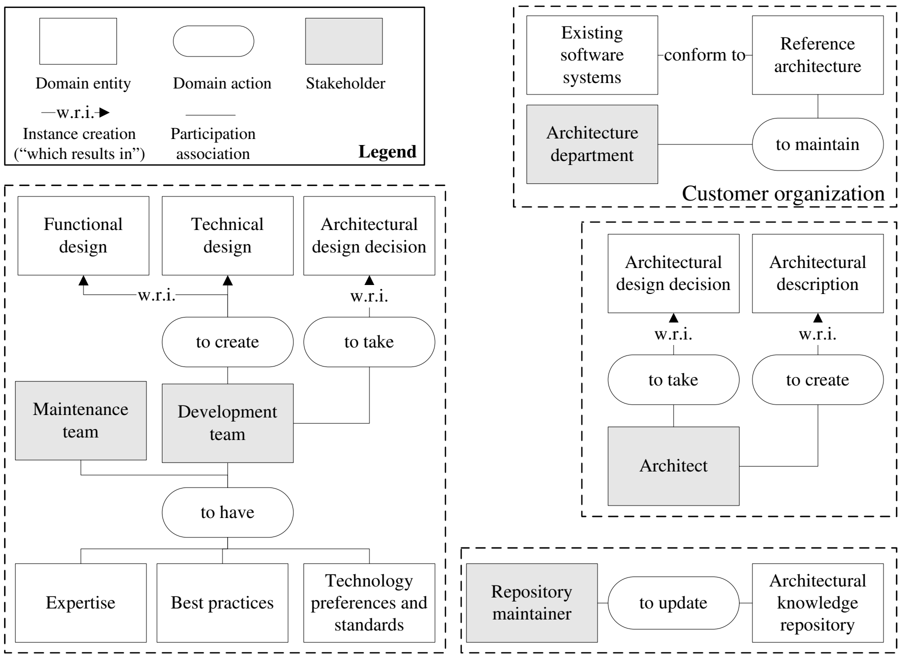


Figure 7.1: Current architecting process

1. **No consistency between architecture and design documents.** There is no alignment between the architecture descriptions and the functional design and technical design documents used by developers and maintainers. Because of the lack of alignment, valuable architectural knowledge might be dispersed in RFA without the architects knowing it. Consequently, it is hard to judge whether the architectural description conflicts with the design that is preferred by the developers. This lack of alignment is visualized in Fig. 7.1 by the absence of a direct connection between the functional and technical design documents shown at the left part and the architectural description shown at the right part. Architectural knowledge in a technical design document that needs to be shared therefore has to travel through two different stakeholders to reach the architecture description.

2. **Communication overhead between stakeholders.** Developers occasionally have to explain the architects technical decisions more than once. The reason for this is that decisions made in earlier meetings, including the rationale for these decisions, are not adequately stored in the architecture description. This knowledge sometimes dissipates quickly. Consequently, architects need to meet again with the developers to get this knowledge explicit at a later point in time. This problem of communication overhead is depicted in the right part of Fig. 7.1 by the lack of an explicit connection between design decisions taken by developers and the architectural description created by the architect.

3. **3. No explicit collaboration with maintenance teams.** Although maintainers are targeted in the architectural documentation, they are not involved as a stakeholder in the architecting process. No active discussions between architects and maintainers take place and the requirements of the maintenance teams are not taken into account during architecture development. The lack of a connection between the maintenance team and the architect in Fig. 7.1 illustrates this problem.

4. **4. No feedback from developers to architects.** Developers sometimes wear the hat of the architect and also make design decisions. However, architects are not informed on the decisions made by the developers unless explicit meetings take place. There is no mechanism in place that allows developers to share what they are doing. Therefore, it is very difficult for the architect to find out what kind of technical issues are encountered or what detailed decisions are taken. The lack of feedback from developers to architects is reflected in the center of Fig. 7.1 by the lack of communication between the development team and the architect.

5. **5. No up-to-date knowledge from development teams in repository.** The architectural knowledge repository contains little to no information on the 'best

practices', technology standards, and expertise currently available at development teams. Therefore, the repository is unable to advise architectural directions that match with the development processes. This problem is visualized in Fig. 7.1 by the fact that design decisions made by the developers are disconnected from the knowledge repository.

6. **6. No up-to-date knowledge from main customer in repository.** The architectural knowledge repository also lacks up-to-date knowledge on the customer organization's reference architecture. Therefore, the repository cannot give architectural directions that automatically comply with the constraints posed by this reference architecture. This is illustrated in Fig. 7.1 by the isolated reference architecture in the top right corner.

7.5 Prerequisites for Successful Architectural Knowledge Sharing

The six issues listed in §7.4 suggested that in RFA architectural knowledge sharing practices were rather immature. We have investigated how the architecting process of this organization needs to be changed in order to improve architectural knowledge sharing. We propose four prerequisites that have to be met in order to change the architecting process in such a way that all six architectural knowledge sharing issues are addressed. We argue that meeting these four prerequisites leads to an improved architecting process in which architectural knowledge is successfully shared.

The four prerequisites are listed below. Two of them are illustrated in more detail in Fig. 7.2 and Fig. 7.3 by showing the differences between the current and the improved architecting process. The thick boxes and lines in this figure highlight the domain entities and actions that relate to the issues in the current process and the solutions to these issues in the improved process.

1. **Alignment between design artifacts.** Architectural descriptions need to be aligned with other design documents. This can be done by enriching the architectural description with links to relevant (lower level) design documents, allowing developers or more technical stakeholders to more easily find their way in the set of documentation. Keeping these links up-to-date ensures that the architecture description always reflects the current state of the system, rendering it a good starting point for stakeholders that want to know something about the system. This prerequisite deals with issue #1: *'No consistency between architecture and design documents'*. In the left part of Fig. 7.2 is visualized how this

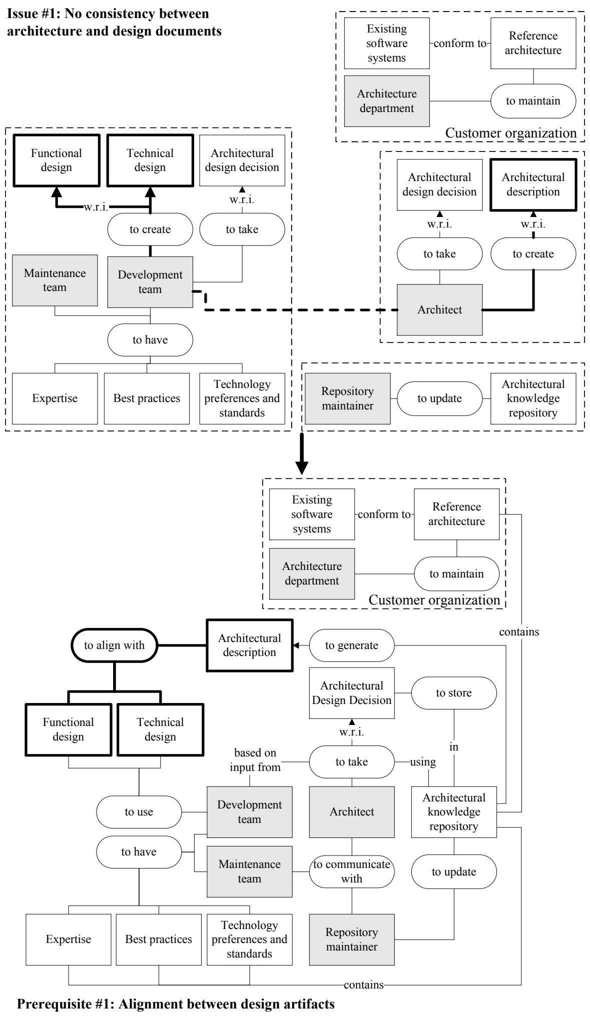


Figure 7.2: Addressing architectural knowledge sharing issue #1

is done. At the top left of this figure the current situation is depicted. There is a clear distance between the architectural description used by the architect on the one hand, and the functional and technical design documents used by the development team and maintenance team on the other hand. The absence of a direct connection between these documents is highlighted by the thick dashed line. This line indicates that there is no check for consistency possible between these documents. In the improved situation, depicted in the lower left part of the figure, the alignment between design artifacts enables this consistency check.

2. **Traceability between architectural decisions and descriptions.** If all architectural design decisions are documented using specific templates (e.g., using the one proposed by Tyree and Akerman (2005)), including considered alternatives and the rationale for the decisions, architectural descriptions provide a good summary of the decision-making process that leads to a certain architecture. Documenting the valuable architectural knowledge prevents its dissipation. As a result, communication between architects and other stakeholders, such as developers, will improve since the current state of the architecting process is known at all times. Consequently, discussions do not need to be held more often than necessary. This prerequisite therefore addresses issue #2: *'Communication overhead between stakeholders'*.

3. **Architects fulfill a central role.** The architects need to fulfill a central role in the architecting process. This guarantees better communication with all involved stakeholders through frequent formal and informal meetings, direct involvement of developers in decision-making, and better collaboration with the maintenance teams. This prerequisite addresses issues #3: *'No explicit collaboration with maintenance teams'* and #4: *'No feedback from developers to architects'*.

4. **Central architectural knowledge repository.** A central architectural knowledge repository allows for storing valuable input on the decision-making process from all stakeholders involved. This improvement is visualized in the right part of Fig. 7.3. In the top right of this figure, the current situation is depicted. Here, the architectural knowledge repository is isolated and valuable knowledge, such as experience from developers or knowledge stored in the reference architecture within the customer organization, is not contained in this repository. In the improved situation this knowledge is stored in a 'central' architectural knowledge repository. The thick lines indicate that this repository is positioned as a central storage facility for architectural knowledge. This prerequisite therefore directly addresses issues #5: *'No up-to-date knowledge from development teams in repository'* and #6: *'No up-to-date knowledge from main customer in repository'*.

In the discussion of the four prerequisites for successful architectural knowledge sharing, we have indicated how the issues found at RFA are addressed. The mapping

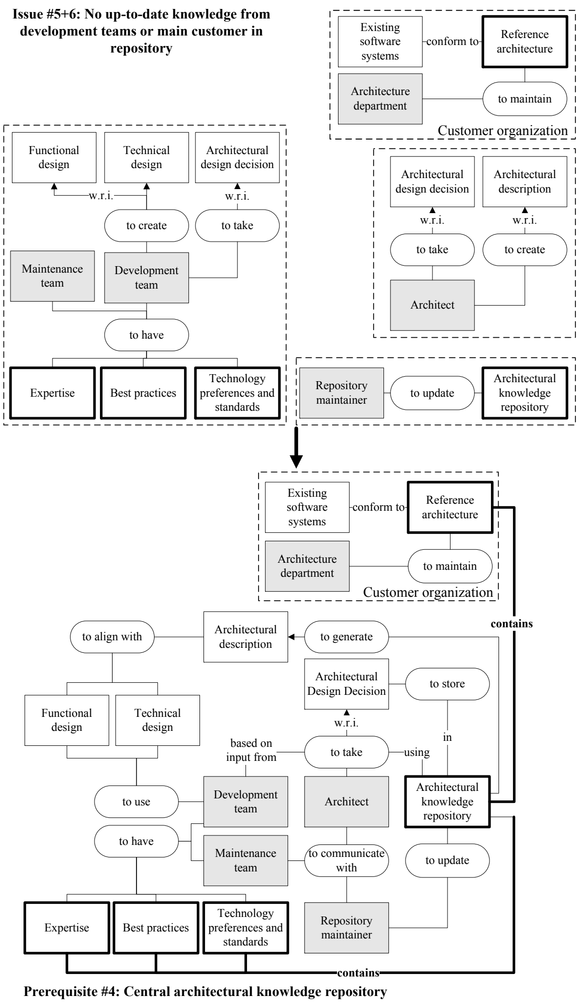


Figure 7.3: Addressing architectural knowledge sharing issue #5+6

between the six identified issues and the prerequisites that address these issues, is further illustrated in the left part of Fig. 7.4.

We argue that architectural knowledge sharing in practice significantly improves if all four of these prerequisites are met. In order to show the importance of these prerequisites, the right part of Fig. 7.4 illustrates that these prerequisites create the identified incentives for sharing architectural knowledge. These incentives are created as follows. Ensuring *'alignment between design design artifacts'* and *'adding traceability between architectural decisions and descriptions'* both improve the quality of knowledge that is made explicit and subsequently shared. As described in §7.2, this positively affects the amount of knowledge that is internalized by stakeholders, hence creating the *'knowledge internalization'* incentive.

The incentive *'establishment of social ties'* is created by allowing the *'architects to fulfill a central role'* in the architecting process. The improved communication initiated by architects can create a bond between stakeholders that induces architectural knowledge sharing. The internal competition is minimized as architects stress the need for cooperation, and by working more closely together the trust between stakeholders also increases. More extensive knowledge sharing establishes social ties, as explained in §7.2.

Finally, implementing a *'central architectural knowledge repository'* enables a feedback loop between developers, maintainers and architects. All stakeholders use

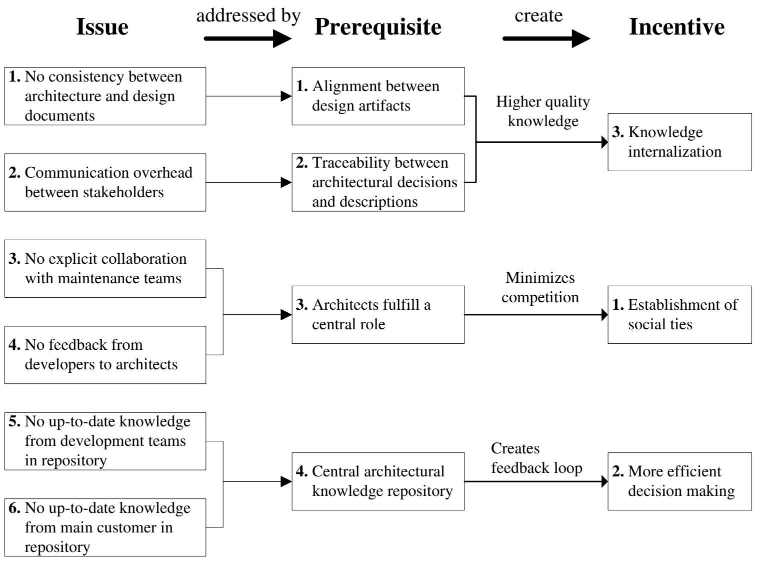

Issue
addressed by
Prerequisite
create
Incentive
1. No consistency between architecture and design documents
2. Communication overhead between stakeholders
3. No explicit collaboration with maintenance teams
4. No feedback from developers to architects
5. No up-to-date knowledge from development teams in repository
6. No up-to-date knowledge from main customer in repository
1. Alignment between design artifacts
2. Traceability between architectural decisions and descriptions
3. Architects fulfill a central role
4. Central architectural knowledge repository
Higher quality knowledge
Minimizes competition
Creates feedback loop
3. Knowledge internalization
1. Establishment of social ties
2. More efficient decision making

Figure 7.4: Mapping prerequisites to the identified issues and incentives

the central repository to monitor the status and progress of the architecture, and are able to participate in the decision making process by sharing expertise, best practices and common sense. This improves the overall group performance. As explained in §7.2 the existence of this feedback loop results in *'more efficient decision making'*.

In this section we have shown that the issues identified during the case study are all addressed if methods and means to share architectural knowledge meet the proposed prerequisites. Next to this, the fact that the four prerequisites map onto the three, context-independent, incentives identified in §7.2, denotes the value of these prerequisites; not only in the context of RFA, but in general. Sharing architectural knowledge is crucial in any architecting process; creating the necessary incentives motivates stakeholders to do so in an efficient and successful way.

7.6 Conclusions

In Fig. 7.5 our research contribution is visualized schematically. In this figure, the numbers between the parentheses denote the main results presented in this chapter.

Based on a literature review on knowledge management principles, combined with our own experience in software architecture research, we identified **(1)** three incentives for sharing architectural knowledge: the establishment of social ties, more efficient decision making, and knowledge internalization. In addition, in this first case study we diagnosed how architectural knowledge is shared in RFA. From this diagnosis, we identified **(2)** a set of issues pertaining to sharing architectural knowledge. In order to address these issues, we proposed **(3)** a set of prerequisites for architectural knowledge sharing.

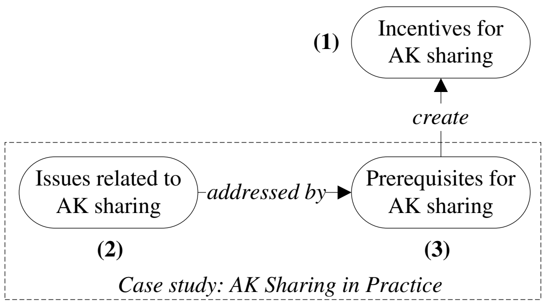

(1)
Incentives for AK sharing
create
Issues related to AK sharing
addressed by
Prerequisites for AK sharing
(2)
(3)
Case study: AK Sharing in Practice

Figure 7.5: Research contribution

We argue that successful architectural knowledge sharing is only possible if these prerequisites are met. We demonstrated this claim by showing that these prerequisites not only address the identified issues, but also create the identified incentives for sharing architectural knowledge. This demonstration indicated that these prerequisites are crucial to foster successful architectural knowledge sharing; not only in the context of RFA, but in general. To answer research question RQ-II.5 we learned that successful sharing of architectural knowledge requires 1) alignment between design artifacts, 2) traceability between architectural decisions and descriptions, 3) architects who fulfill a central role in the architecting process, and 4) a central architectural knowledge repository.

To diagnose the challenges related to architectural knowledge sharing, we have modeled the architecting process of RFA using four different perspectives. This allowed us to diagnose the issues related to architectural knowledge sharing. These issues are all organizational challenges that need to be addressed in order to guarantee effective sharing of architectural knowledge, so the identification of these issues has helped us answering research question RQ-II.4. Next to acting as a diagnosis instrument, our modeling approach served as an implementation instrument to help us suggest how the architecting process of RFA should be improved.

Our next goal is to define and implement methods and tools for sharing architectural knowledge. The results of this chapter indicate that in order to make these methods and tools effective, it is important that this further enables the proposed prerequisites for architectural knowledge sharing. In the next chapter, we study desired properties of tools to support sharing of architectural knowledge more elaborately.

8 Effective Support for Sharing Architectural Knowledge

In this chapter we define desired properties for effective support for sharing architectural knowledge. These properties were characterized based on an investigation of the main characteristics of architecting, plus application of relevant knowledge management best practices. We also highlight the design and implementation of a web portal that conforms to all defined properties. This conformance makes the portal a good step toward effective tool for sharing architectural knowledge.

8.1 Introduction

In the previous chapter we focused on typical organizational challenges in the architecting process. Now that we have a better understanding on prerequisites for successful architectural knowledge sharing, in this chapter we examine how tools can best be employed to facilitate architects in architectural knowledge sharing, and assess what kind of knowledge sharing strategy fits the architecting process best. We thus take a more solution-oriented stance towards sharing architectural knowledge, which fits well with our planning stage of our action research cycle (see Fig. 6.2).

The need for sharing architectural knowledge was already observed during the first case study conducted at RFA, as reported in Chapter 7. During that study we found that architects rely heavily on communication to work adequately and to produce high-quality results, and that many formal and informal discussions take place in the coffee room, at the hallway, or during other social events or meetings. However, despite the need for sharing architectural knowledge, software architects in practice often stick to familiar technologies such as office suites for their daily tasks. Since such tools are not explicitly aimed at sharing architectural knowledge, most of this knowledge remains implicit.

In this chapter we propose foundations for *effective* tool support for architectural knowledge sharing. To this end, we define a set of properties architectural knowledge sharing tools should have. These properties are derived by defining typical characteristics of the architecting process, and by taking into account best practices from the knowledge management domain. For the former, we combine our observations of software architecture practice with a review of software architecture literature. For the latter – because architecting is a knowledge-intensive process – we review knowledge management literature.

Based on the set of desired properties, we asses the conformance of a number of architectural knowledge sharing tools to these properties. The results indicate that not all of these properties are adequately supported. In an attempt to improve the status quo, we propose EAGLE: an architectural knowledge web portal. This portal offers a hybrid architectural knowledge management approach and to indicate its potential effectiveness, we show that it conforms to all aforementioned desired properties.

The remainder of this chapter is organized as follows. In §8.2 we describe our observations of architectural knowledge sharing during a second case study at RFA. In §8.3 we combine these observations with a review on state-of-the-art software architecture literature in order to define the main characteristics of software architecting. In §8.4 we elaborate on best practices known from knowledge management literature. Based on these best practices, in §8.5 we define a set of desired properties of architectural knowledge sharing tools. In §8.6 we examine existing software architecture tools and investigate how well these tools adhere to the defined properties. In §8.7 the design and implementation of EAGLE is presented, which is an architectural knowledge portal that implements all aforementioned properties. Finally, in §8.8 we present our long-term vision on EAGLE's application as blackboard system for architectural knowledge sharing.

8.2 Architectural Knowledge Sharing in Practice

One can only understand what architects really need by asking them. This principle motivated us to closely investigate the software architecture practice in our research on architectural knowledge sharing tool support. This investigation helped to explain why only few of the software architecture tools proposed by academics seem to really make it to software architecture industry.

To improve our initial understanding of the architecting process built up by the first case study, during our second case study at RFA, we closely monitored the architecting activities undertaken, the various roles architects have, their information needs, and the tools they use for their daily tasks. This study fits well with the ‘action planning’ phase of an action research cycle, since we diagnosed the architecting process of RFA further, based on which we planned which improvements were possible.

In RFA software architectures play an important role. Architectural descriptions are used as basis for development, and architectural guidelines need to be adhered to during system development and maintenance. In most software development projects architects work together in teams. One of the reasons for working in a team is that not all architects are skilled in both business and technology related aspects, while in most projects both these aspects play an important role. Consequently, there is a need to communicate and share information – both among architects and other stakeholders.

Architects in RFA have acknowledged that it is often hard to share information in a structured way. As a result, information sometimes gets lost when work is transferred from one department to the other, leading to redundant work and a lower overall quality of the architecture. Sharing relevant information is further constrained by deadline pressures posed by most projects the architects work on. There is usually insufficient time to explore all possibilities or alternatives, or to validate the results gained from brainstorm sessions with colleagues or customers.

Below we elaborate upon our observations with respect to the three prominent architectural knowledge sharing tools available in RFA: an expertise site, a knowledge repository, and a knowledge maps system.

- **Expertise site.** For the architecture department of RFA an architecture expertise website has been created. This site uses Microsoft Sharepoint as underlying technology and offers functionality such as discussion forums, and news updates. However, during our investigation we found that in its current form the expertise site does not appeal to architects. Main reason for this lack of appeal is the fact that there is very little content and only a small amount of people are registered users. Furthermore, editing or adding information to the site is non-intuitive and the relatively old information is difficult to retrieve. The expertise site is merely used as a document repository where presentations and white papers are stored. Architects indicated that they visit the expertise site only occasionally, and this is partly due to its limited content, but also because they lack a real community feeling in the architecting department in the first place.

- **Knowledge repository.** RFA has developed a knowledge repository that harbors a set of guidelines used to guide an architect in creating an architecture. After the architect answers a number of predefined questions, the repository uses its guidelines to advise the architect about the architectural solution. Unfortunately, in practice this repository is hardly used by architects. During semi-structured interviews with several architects the reasons for this problem became apparent: the tool is highly prescriptive and the architects' perception is that this limits the freedom they have in devising a solution. Moreover, the guidelines stored in the repository are outdated and the list of questions is rather large, and therefore time consuming. For these reasons, instead of using the repository architects prefer to edit existing architectural descriptions, since that yields results of similar quality while saving time. Many architects indicated that they have tried out the repository once or twice, after which they concluded the added value of using it was limited.

- **Knowledge maps.** RFA has also started an organization-wide initiative that aims to connect knowledge and knowledge workers throughout the organization. This intranet system is an information place where various information sources within the organization are combined and presented to the user. Users are able to fill in so-called knowledge maps that contain their areas of interest or expertise. Knowledge maps should make it easy to find colleagues based on their expertise. Unfortunately, the knowledge maps system is not that successful. The user interface is non-intuitive and slow, and on certain areas of interest, such as software architecture, no knowledge maps can be defined. For these reasons, the knowledge maps system is considered rather useless by the architects.

An often heard complaint during the interviews held with the architects was that there are too many systems that contain useful information. As a result, people often decide to just ask a colleague directly, or stop looking at all, since they do not know which system harbors what knowledge, and thus what is the right place to look. Architects indicated that a more central location that acts as glue between existing tools and that could be used as starting point for various knowledge-related tasks, would be a much welcomed alternative for the current state of practice.

8.3 Characteristics of Architecting

Software architecting is a knowledge-intensive process in which many design decisions are taken. These decisions are taken by carefully considering the available solutions, after which the best alternative is chosen. Architects often apply proved solutions, such as tactics or patterns, to guarantee a high-quality architectural design.

As described in §8.2, RFA is struggling with how to effectively use tools to share architectural knowledge. The architectural knowledge sharing tools in place do not match the needs of the architects. To accommodate architects and other stakeholders in their knowledge needs, *effective* architectural knowledge sharing tool support is needed. Effective here means that the tools align to what architects in practice do, and that the time and effort required for using the tools is limited.

In order to properly define properties of effective architectural knowledge sharing tools, we investigate what a typical architecting process entails. In this section we use software architecture literature to define five characteristics of the architecting process. We do not claim that our set of characteristics is complete, but we believe that by focusing on a broad set of literature we have covered the essential properties of architecting. Please note that not all these properties are unique to architecting; some of them could easily apply to other software engineering disciplines as well, such as detailed design or testing. The lack of uniqueness and completeness notwithstanding, studying characteristics of the architecting process allows us to define how architectural knowledge sharing tool support could be designed more effectively.

- **Architecting is consensus decision making.** Architecting can be viewed as a decision making process that not only seeks the agreement of most stakeholders, but also resolves or mitigates the objections of the minority to achieve the most agreeable solution (Eeles, 2006c). Often various stakeholders with different needs and concerns are involved in the architecting process. This is acknowledged by Bass et al. (2003), who present the Architecture Business Cycle to define architecting as a feedback loop between architects, the stakeholders and the architectural solution itself. Although the architects take the final design decisions, they often do so in accordance with the other important stakeholders. This decision making process lends itself for knowledge sharing initiatives that allow stakeholders to actively participate in this process. Architectural knowledge sharing tools should enable architects to efficiently work together in a team. Due to the size and complexity of most software systems, it is often infeasible for one architect to be responsible for everything alone. This focus on teamwork is especially true in global software engineering environments. Consequently, the 'architect role' is often fulfilled by multiple collaborating architects.

- **Architecting is iterative in nature.** Due to the consensus-driven decision making characteristic, architectures are not designed overnight, but rather in an iterative way. Hofmeister et al. (2007) illustrate this iterative nature of architecting by the concept of a backlog that is implicitly or explicitly maintained by architects. This backlog contains smaller needs, issues, problems they need to tackle and ideas they might want to use. Such a backlog drives the workflow, helping the architect determine what to do or decide next. Conceptually the architecture is finished when this backlog is empty. However, as long as the backlog has open issues it is worth relating these issues to the current state of the architecture design. Architects can then judge how these issues could best be addressed while maintaining the important qualities of the architectural design. Architectural knowledge sharing tools therefore should support traceability between knowledge entities, such as architectural decisions and identified problems.

- **Architecting is a creative activity.** Architects are responsible for reflecting the design decisions taken in comprehensive architectural models, by selecting the most suitable views and viewpoints or architecture description language. During these activities the creativity of the architect plays a crucial role (Eeles, 2006a). This is particularly true when dealing with novel and unprecedented systems. In such cases, there may be no codified experience to draw upon. Knowledge sharing tools need to take this characteristic into account and should support the architect's creativity instead of constraining it. This means that methods and tools probably work better if they are more descriptive in nature.

- **Architecting impacts the complete life-cycle.** Many architects would agree on the statement that an architecture is never finished, but rather stays alive throughout the life of the software system. During maintenance and system evolution an architecture plays an important role in safeguarding architectural qualities. If relevant architectural knowledge is not stored correctly knowledge vaporization may be the result, turning an architecture into a black box (Bosch, 2004). Knowledge sharing tools should therefore make available important architectural knowledge to various stakeholders, such as developers and maintainers, instead of only targeting the architects.

- **Architecting is constrained by time.** The previous characteristics of architecting show that architecting is a creative consensus-driven and iterative decision making process. In practice, however, a heavy constraint on these characteristics is the available time that architects have. Often, 'time to market' forces architects to choose for suboptimal solutions. This phenomenon was one of the conclusions of a recent workshop about sharing and reusing architectural knowledge: "in practice, architects find only one solution and not multiple alternatives to choose from" (Lago and Avgeriou, 2006). This is due to the hard constraints in industrial practice (e.g., time to market or budget) that forces architects to intuitively come up with a single solution based on their existing application-generic knowledge. In effect, this results in the architects not exploring the solution space and potentially missing alternative solutions.

8.4 What to Learn from Knowledge Management?

The recent emphasis on architectural knowledge has sparked discussions on how to capture and manage such knowledge, such as the know-why and know-how of architectural solutions, in an appropriate way. To answer this question, software architecture researchers benefit from best practices (such as methods and tools, methods) of the knowledge management domain, such as reported in (Lindvall et al., 2001).

The characteristics of architecting described in the previous section show that architecting is a creative, iterative decision making process often done in collaboration with colleagues and other stakeholders in the lifecycle. The fact that architects only have a limited amount of time to complete this decision making process further shows the need for effective architectural knowledge sharing support. In this section we elaborate on best practices for knowledge sharing based on a critical analysis of knowledge management literature. Based on this literature analysis we investigate how the architecture domain could learn from this established field. Please note that in this chapter we restrict ourselves to knowledge sharing factors related to tool support. Various social (Kankanhalli et al., 2005), organizational and cultural (Cummings, 2003), or personal factors (Nonaka and Takeuchi, 1995), also heavily influence the success of knowledge sharing, but are not explored in this study. For a discussion on how to create incentives for sharing architectural knowledge, we refer to Chapter 7.

Knowledge in software engineering is diverse and its proportion immense and steadily growing. Improved use of this knowledge is the basic motivation and driver for knowledge sharing in software engineering (Rus and Lindvall, 2002). Only since the early nineties have the knowledge management and software engineering communities begun to grow together (Aurum et al., 2003). Since then, various knowledge sharing tools have been proposed in the software engineering domain, leading to concepts such as the Experience Factory (Basili et al., 1994), experience management systems (Seaman et al., 2003), learning software organizations (Althoff et al., 2000), and Software Engineering Decision Support (Ruhe, 2002). In addition, considerable attention has been paid to the concept of design rationale (Dutoit et al., 2006), and various approaches based on this concept have been proposed, such as IBIS (Conklin and Burgess-Yakemovic, 1991) and QOC (MacLean et al., 1996).

Literature warns that not all knowledge sharing implementations are automatically successful. Ghosh (2004) lists several factors that make knowledge sharing difficult, such as the fact that knowledge sharing is time consuming, and that people might not trust the knowledge management system. Another warning given is that striving for completeness is infeasible. Lakshminarayanan et al. (2006) argue that it is impossible to create a tool to reason about all the issues that an architect would normally need to consider. In addition, we should be aware of the fact that a lot of the available knowledge cannot be made explicit at all, but instead remains tacit in the minds of people (Nonaka and Takeuchi, 1995). Sharing such tacit knowledge is very hard (Haldin-Herrgard, 2000). The potential limitations of knowledge sharing notwithstanding, we believe it is crucial to assist architects in practice with their daily work. Tools should assist architects in their knowledge-intensive tasks, by enabling them to discover, share, and manage architectural knowledge.

In order to design successful tools for knowledge sharing, a strategy needs to be chosen. Hansen et al. (1999) distinguish two main knowledge management strategies: codification and personalization. Whereas codification is aimed at systematically storing knowledge so that it becomes available to people in the company, the personalization strategy focuses on storing information *about* knowledge sources, so that people know who knows what. In the architecting process, some architectural knowledge might benefit from a codification strategy, whereas other types of knowledge could be better shared using personalization approaches. A hybrid approach, first coined by Desouza et al. (2006), is therefore worth considering (cf. Chapter 3, §3.4). Such a hybrid approach provides a balance between formalized and unstructured knowledge. According to Hall (2001), such a balance is an important prerequisite to stimulate the usage of tools.

To define in more detail how a hybrid architectural knowledge sharing approach should look like we can draw on a study about knowledge sharing by van den Brink (2003). He describes that four steps need to be executed in order to create "an interconnected environment supporting communication, collaboration, and information sharing within and among office and non-office work activities; with office systems, groupware, and intranets providing the bonding glue". Firstly, information and explicit knowledge components must be stored online, indexed and mapped, so people can see what is available and can find it (e.g., using digitally stored documents or yellow pages). Secondly, communication among people needs to be supported, by assisting in the use of best practices to guide future behavior and enable sharing of ideas (e.g., emails, bulletin boards, or discussion databases). Thirdly, tacit knowledge needs to be captured using for instance communities of practice, interest groups, or competency centers (e.g., groupware and electronic whiteboards). Lastly, methods are required that offer a virtual space in which a team can collaborate interactively, irrespective of geographic distribution of the team members or time. To enable the four steps described above, van den Brink (2003) defines three categories that form technological enablers for knowledge sharing: knowledge repository (for sharing explicit knowledge); knowledge routemap (for sharing explicit and tacit knowledge), and collaborative platforms (for sharing tacit knowledge).

The need for a combined approach that stimulates the collaboration of architects and that supports sharing both tacit and explicit knowledge, is also acknowledged by Röll (2004). He describes seven knowledge work processes that range from finding codified information to establishing social networks and collaborating in communities. The author states that these knowledge processes can not be seen independently, but often are interrelated. For example, an architect searching for information on security might a) initiate a search on this specific topic, b) negotiate with colleagues about the meaning of what was just found, c) create new ideas based on the discussions and by using common sense, and d) try to maintain a social bond with these colleagues at the same time.

Based on the above, we argue that architectural knowledge sharing tools can best follow a hybrid approach that combines codification and personalization methods, and that also stimulates collaboration between the stakeholders of the architecting process. More stable knowledge – such as best practices and architectural tactics – could be codified in a repository, less formalized knowledge could be spread in the organization more effectively using knowledge routemaps, and a collaborative platform allows architects and other stakeholders to work together on an iterative decision making process.

8.5 Desired Properties of Architectural Knowledge Sharing Tools

Based on the five characteristics of software architecting (§8.3) and best practices from knowledge management literature (§8.4) we define seven desired properties of architectural knowledge sharing tool support.

1. **Stakeholder-specific content** Because *Architecting impacts the complete life-cycle* various stakeholders are involved in the decision making process, and all these stakeholders need specialized views on the available content, such as open issues, approved decisions, or scheduled meetings. Architectural knowledge sharing tools should make it possible to distinguish between certain types of knowledge. Users of the tool can then choose what architectural knowledge they want to retrieve. The search functions should therefore match with the profiles of different users, such as developers, maintainers, architects, or project managers.

2. **Easy manipulation of content** Since *Architecting is iterative in nature*, architects follow a continuous iterative decision making process. Easy manipulation of content will keep the decision making process up to speed, whereas more rigid tool support that does not allow easy manipulation could instead slow it down.

3. **Descriptive in nature** Because *Architecting is an art*, architects should not be constrained too much in their tasks. Tools that support architectural knowledge sharing should not prescribe *how* architects should use those tools, for example by offering an abundance of predefined models, guidelines, or templates. Instead the tools should allow a descriptive perspective on the available architectural knowledge that does not limit the architects' creativity.

4. **Support for architectural knowledge codification** Since *Architecting is constrained by time*, architects could be helped by quickly finding relevant architectural knowledge. For certain types of knowledge that is not subject to frequent changes, a codification strategy probably works best. Architects can then easily retrieve solutions that have proven themselves in the past, and reuse these solutions accordingly. This property also relates to the knowledge repository category of van den Brink (2003), and the statement of Hall (2001) that there should be a proper balance between formalized (i.e., codified) and less structured (i.e., personalized) knowledge.

5. **Support for architectural knowledge personalization** Because *Architecting is consensus decision making*, architectural knowledge is not always immediately 'stable' enough to codify, because until consensus has been reached, decisions could change. For such knowledge, a personalization strategy could prove useful to enable architects to find who knows what, in a similar way as the knowledge routemaps proposed by van den Brink (2003). Personalization techniques are also valuable to support the discussions and negotiations between stakeholders (Röll, 2004).

6. **Support for collaboration** Because *Architecting is consensus decision-making* architectural knowledge tool support should explicitly support collaboration between different users. This property relates to the groupware criterion proposed by van den Brink (2003), as well as to the collaboration requirement of Röll (2004). This property enables architects to actively involve all important stakeholders in the decision making process. Since most architects are specialized in certain areas, tool support that supports collaboration also allows architects to use a 'divide and conquer' approach whenever possible.

7. **Sticky in nature.** This property could be seen as orthogonal to the others in the sense that it is an essential property to motivate people to start using an architectural knowledge sharing tool in the first place. With this we mean that people should be motivated to start using the tool, as elaborated upon by Boer et al. (2002), who describe several motivational factors. In addition, Hall (2001)

argues that the user interface should be attractive and user friendly to stimulate usage of the tool. To prevent users from neglecting the tool after having played with it once, according to Bush and Tiwana (2005) special features should be incorporated in the tool to let it obtain a certain level of 'stickiness'. Tools that are sticky motivate users to keep coming back to it, increasing the chance of widespread adoption in practice.

8.6 The Status Quo of Architectural Knowledge Sharing Tools

Based on the seven identified desired properties of architectural knowledge sharing tools, in this section we assess the status quo of tool support in the software architecture domain. We have selected a set of tools that represent the current state of the art in architectural knowledge management tools, since all these tools have recently been introduced in software architecture literature and they all explicitly target architectural knowledge management as well.

The academic tools that are assessed include Archium, ADDSS, DGA DDR, and PAKME. Archium is a tool environment proposed by Jansen et al. (2007) that is aimed at establishing and maintaining traceability between design decision models and the software architecture design. Capilla et al. (2006) have proposed a web-based tool called ADDSS for recording and managing architectural design decisions. Falessi et al. (2006) have devised a specific design decision rationale documentation technique, which is driven by the decision goals and design alternatives available. Hence, it is called the Decision Goal and Alternatives (DGA) DDR technique. Ali Babar et al. (2005) have proposed a Process-based Architecture Knowledge Management Environment (PAKME) that allows storing generic architectural knowledge (such as general scenarios, patterns, and quality attributes), and project specific architecture knowledge (such as concrete scenarios, contextualized patterns, and quality factors). To form a balanced set of tools from academia and practice, we add to our assessment RFA's three architectural knowledge sharing tools: the knowledge repository, expertise site and knowledge maps system. More details on these three tools are described in §8.2.

The results of the assessment are reflected in Table 8.1. For the assessment of the tools of RFA we were able to draw on our observations in this organization. For the assessment of the academic tools we used published literature about the tools as primary source of information. We have based the scores on our interpretation of the tools, but acknowledge that it is possible that the tools have evolved recently. Please note that we do not intend to give a strict judgment on the tools, but rather indicate

Table 8.1: Status quo of architectural knowledge sharing tools

	SA			*KM*			
Desired property / **Software arch. tool**	Stakeholder-specific content	Easy manipulation of content	Descriptive in nature	Support for AK codification	Support for AK personalization	Support for collaboration	Sticky in nature
Archium	-	+	+	+	-	-	?
ADDSS	-	-	+	+	-	-	?
DGA DDR	-	-	+	+	-	-	?
PAKME	-	+	+	+	-	+	?
PGR knowledge repository	-	-	-	+	-	-	-
PGR expertise site	-	+	+	+	-	-	-
PGR knowledge maps	-	-	+	-	+	-	-

whether they conform to the defined properties. If they do, this is reflected in Table 8.1 with a '+' score; if they do not, this has resulted in a '-' score.

Stakeholder-specific content is a property that we haven't found explicit support for in any of the studied tools. Archium is designed for a single user who could use the tool to establish and maintain traceability between design decision models and the software architecture design. The authors of ADDSS mention multi-perspective support as one of the envisioned features of ADDSS, but in the current prototype this is not yet implemented. DGA DDR and PAKME also do not mention stakeholder-specific content as a feature, but rather focus on the codification of the architectural knowledge in general. The same goes for RFA's knowledge repository, and the expertise site, although the latter allows users to find a lot of different types of information, but this information is not tailored to users. RFA's knowledge maps system is suffering from the same limitation since users can find experts on certain topics based on the knowledge maps, but the view they are able to obtain on this information is not customizable.

Easy manipulation of content is incorporated in three of the tools we studied. The language used in Archium offers explicit support for addition and modification of architectural design decisions. In PAKME, a maintenance component provides various features to modify, delete and instantiate different artifacts. This component also includes repository administration functions. RFA's expertise site in theory allows various stakeholders to change or update knowledge at a regular basis, although architects indicated that such changes are not always "easy" to make. In RFA's repository and knowledge maps system modifying content is non-intuitive and time-consuming, resulting in a negative score for this property. With both DGA DDR and ADDSS we believe the underlying model used in these tools is rather formal and limited in scope. As a result, chances are that architects feel too constrained by having to comply to these models. An issue specific to ADDSS is that architecting is not assumed to be iterative. This focus becomes apparent in ADDSS because the user can add the design decisions taken and then add a view to these decisions to gain traceability. However, changing the design decisions is only possible by starting a new iteration, and if this is done a new view also has to be included to prevent loss of traceability. Between iterations no connections are possible, so manipulating existing architectural knowledge is hard.

Descriptive in nature is something most tools in our investigation are, except RFA's repository that strongly prescribes the solution based on the questions that need to be filled in. This is related to the fact that most of the tools also score well on the *support for AK codification* property. Most tools follow a typical codification strategy. Users are able to add information to the tool that is then stored for future retrieval. The one exception to this approach is RFA's knowledge maps systems, which offers a mechanism for people in the organization to find each other. This *support for AK personalization* is unfortunately not well covered by all other tools. A related critique on the tools studied is that they also score low on explicit collaboration aspects, since they are built around the assumption that architects mainly codify information for reuse purposes. Although RFA's knowledge maps system allows experts to find each other, it does not offer structured means to let these experts collaborate. As a result most of the investigated tools get a '-' score on the *support for collaboration* property. The single exception is PAKME, which is built on top of an open source groupware platform to provide collaboration using content management, project management and the like.

To conclude our assessment, we have investigated whether the various tools are *sticky in nature*. To properly determine this for ADDSS, DGA DDR, Archium and PAKME, hands on experience is required, hence the question marks in Table 8.1. For the other three tools, we can assess the stickiness based on the interviews we held with architects of RFA. The results of these interviews indicate that none of the three architectural knowledge sharing tools in RFA is sticky, since users are not motivated to keep using the tools: the knowledge repository is outdated and offers little added value; the expertise site is not flexible in adding, manipulating or retrieving architectural knowledge; the knowledge maps are very rigid and do not offer specific topics on software architecture, rendering this system useless for the architects.

In summary, we conclude that most of the architectural knowledge sharing tools we studied are descriptive in nature and focus primarily on codification. To improve the status quo, more emphasis should be put on stakeholder-specific content, support for personalization, explicit support for collaboration, and general characteristics that make tools appealing and sticky in the long run.

8.7 EAGLE: Effective Tool Support for Sharing Architectural Knowledge

In our effort to arrive at effective tool support for sharing architectural knowledge, we have designed and implemented an architectural knowledge portal that acts as an Environment for Architects to Gain and Leverage Expertise (EAGLE). A first prototype of EAGLE has been subject to a pilot in the architecture department of RFA. During the trials the architects were enthusiastic about EAGLE's potential. More extensive experimentation of EAGLE is part of a next case study, elaborated upon in Chapter 9.

In the following subsections we will respectively discuss in depth the architecture design of EAGLE, its graphical user interface, the main architectural knowledge modules it contains, and our long-term vision on the potential of this portal. EAGLE is designed to incorporate all seven desired properties introduced in §8.5, so that it is an effective means to support architectural knowledge sharing. To propose evidence to this claim, wherever appropriate, the characteristics and features of EAGLE are linked to these properties using italic text. An overview of the desired properties and how these are addressed by EAGLE is provided in Table 8.2.

8.7.1 Architectural design

The central vision behind EAGLE was to create an integrated environment that allows architectural knowledge sharing in various ways. In addition, we aimed at providing stakeholder-specific content. A frequently occurring mistake in organizations is that they provide all employees with the same content. We believe that architects would like to have more flexibility, so that they can manage exactly the architectural knowledge they need.

Heavily driven by the integration and flexibility requirements, we have chosen a web based portal as our underlying framework to build EAGLE. The advantage of web based over standalone applications is that a large group of users can be reached, while the maintenance of the application can be performed centrally. Advantages of portals over 'normal' websites include that a portal acts as one single access point to various

Table 8.2: Desired properties addressed by EAGLE

Desired property / **EAGLE**	Stakeholder-specific content	Easy manipulation of content	Descriptive in nature	Support for AK codification	Support for AK personalization	Support for collaboration	Sticky in nature
Graphical User Interface	√	√	√				√
Notifications & subscriptions							√
Best practices repository				√			
Document repository				√			
Yellow pages					√		√
Discussion boards						√	√
Project environments				√	√	√	
Blogs						√	

functionality, provides easy access to internal and external information sources, and that it offers a personal, adaptable environment. A portal is therefore the ideal means to turn EAGLE in aforementioned integrated environment for architectural knowledge sharing. The architectural design of EAGLE is depicted in Fig. 8.1.

EAGLE is a client-server application. At the server side an Apache Webserver communicates with a MySQL database in which all architectural knowledge is stored. The database uses a specific datamodel to capture various architectural knowledge entities, such as design decisions, concerns, and rationale. Various PHP scripts reside at the web server to operate on this knowledge. For each of the architectural knowledge modules one or more PHP scripts are created to handle client requests on the one hand, and to communicate with the database on the other hand.

For the client side of EAGLE, we selected a suitable open source framework: Portaneo, a Rich Internet Application.[1] Portaneo is highly modifiable, has a flexible plugin system – making EAGLE highly extensible – and, above all, is free. These characteristics made it a better choice than existing commercial software such as Microsoft Sharepoint[2], because with Portaneo we are able to experiment more easily with the portal, while using a minimum of resources. Moreover, we felt that building a portal from

[1]http://www.portaneo.com/solutions/en/

[2]http://www.microsoft.com/sharepoint/default.mspx

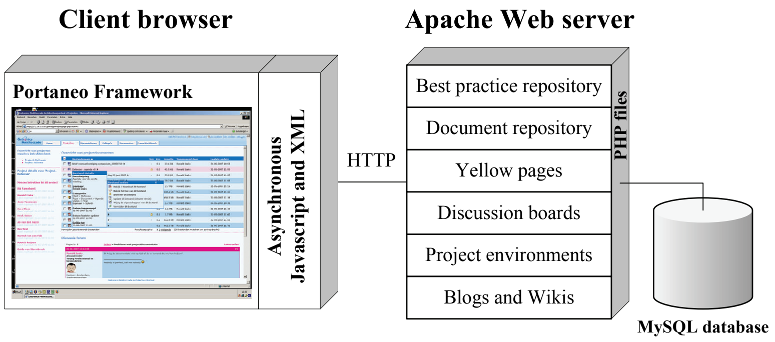


Figure 8.1: Architecture of EAGLE

scratch using open source enabled us to tailor the portal towards specific architectural knowledge sharing functionality as much as possible. A last reason to opt for Portaneo is its relatively large user community and the fact a standard is used for content management. As a result, various modules available on the Internet can be incorporated in the portal[3].

The client browser contains the portal functionality organized in several tabs and menus on a web page. All tabs and menus communicate directly with the architectural knowledge modules that reside on the web server. A more detailed discussion on the functionality of the various modules can be found in §8.7.3 The current version of EAGLE is optimized for use with Internet Explorer, but other web browsers can be used as well.

Communication between the web browser and web server takes place using the standard HTTP protocol. However, instead of using a traditional web application model where the browser itself is responsible for initiating requests to, and processing requests from, the web server, we use asynchronous Javascript and XML – also known as Ajax – to act as an intermediate layer to handle communication requests (Zakas et al., 2006). This communication layer, that is depicted on the right side of the client browser in Fig. 8.1, is really just a JavaScript object or function that is called whenever information needs to be requested from the server. Instead of the traditional model of providing a link to another resource (such as another web page), each link makes a call to the communication layer, which schedules and executes the request. This request is done asynchronously, which means that code execution does not wait for a response before continuing. When the communication layer receives the server response, it goes into

[3]See http://www.google.com/ig/directory?synd=open for the list of available modules.

action, often parsing the data and making several changes to the user interface based on the information provided.

Asynchronous communication involves transferring much less information than a traditional web application model. Consequently, user interface updates are faster, so that users are able to do their work more efficiently. Moreover, it looks more professional when only those parts of a web page are updated that are actually used at that moment. For these reasons EAGLE mostly uses asynchronous communication. Synchronous communication is only used when the portal is start up for the first time – which means that the whole page has to be loaded by the browser – or for operations such as file transfers.

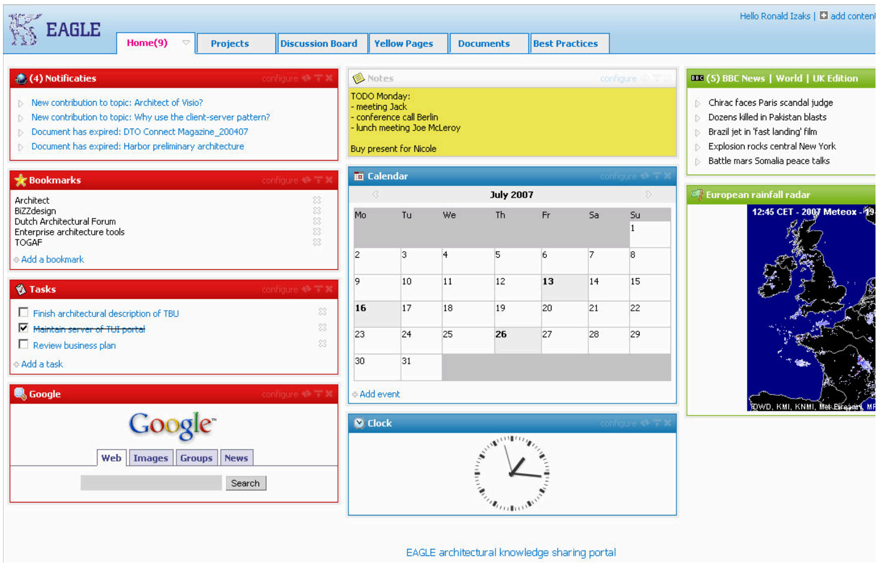
8.7.2 Graphical user interface

As already pointed out in the previous section, EAGLE is a highly modularized portal. This property is reflected in the graphical user interface. Users are able to select a number of 'GUI modules' that are organized in their browser screen. There are two types of GUI modules: 1) small modules containing little information of which several appear on one page, and 2) page-wide modules which are denoted as tabs. A personalized portal therefore consists of multiple tabs, each of which contains one of the listed type of modules.

Due to the flexibility users have in organizing their view on the existing content, and the fact that various types of architectural knowledge are distinguished in the underlying datamodel, EAGLE scores well on the *stakeholder-specific content* property. Due to the module standard used, users can also import functionality from the Internet, apart from choosing among the main architectural knowledge modules (which are elaborated upon in the next section).

EAGLE does not pose restrictions on how it should be used. It primarily acts as a gateway to various types of architectural knowledge, both in codified and personalized forms. As described in §8.2 this integration aspect was one of the features architects of RFA would greatly appreciate. Because no restrictions are put on its use, EAGLE conforms to the *highly descriptive in nature* property.

Fig. 8.2 depicts a screenshot of the start page of EAGLE. In this screenshot aforementioned modules are organized in a number of tabs (on the top of the screen), as well as in a number of main blocks in the center of the screen. The tabs give access to the various architectural knowledge modules described in §8.7.3. The blocks in the center of the start page give access to additional useful knowledge sources. These are typical knowledge sources that architects at RFA – and arguably users in general – request on a regular basis, such as the weather report, a search bar, and a calender. The fact that EAGLE integrates architectural knowledge sharing features with these



(4) Notificaties
- New contribution to topic: Architect of Visio?
- New contribution to topic: Why use the client-server pattern?
- Document has expired: DTO Connect Magazine_200407
- Document has expired: Harbor preliminary architecture

Notes
TODO Monday:
- meeting Jack
- conference call Berlin
- lunch meeting Joe McLeroy
Buy present for Nicole

(5) BBC News | World | UK Edition
- Chirac faces Paris scandal judge
- Dozens killed in Pakistan blasts
- Brazil jet in 'fast landing' film
- Explosion rocks central New York
- Battle mars Somalia peace talks

Bookmarks
Architect
BIZZdesign
Dutch Architectural Forum
Enterprise architecture tools
TOGAF
Add a bookmark

Calendar
July 2007

Mo	Tu	We	Th	Fr	Sa	Su
						1
2	3	4	5	6	7	8
9	10	11	12	**13**	14	15
16	17	18	19	20	21	22
23	24	25	**26**	27	28	29
30	31					

Add event

European rainfall radar
12:45 CET - 2007 Meteox - 19
DWD, KMI, KNMI, Met Eireann, MF

Tasks
- [ ] Finish architectural description of TBU
- [x] ~~Maintain server of TUI portal~~
- [ ] Review business plan
Add a task

Google

Clock

EAGLE architectural knowledge sharing portal

Figure 8.2: EAGLE's start page

sources, further adds to EAGLE's attractiveness, because it combines access to various knowledge, both work-related and private needs. Hence, we argue that EAGLE is rather *sticky in nature*, thereby conforming to this property as well. Furthermore, editing architectural knowledge in all of EAGLE's modules is easy due to the intuitive user interface, ensuring *easy manipulation of content*.

8.7.3 Architectural knowledge modules

In this section we will elaborate upon the key modules of the architectural knowledge portal, most of which are currently implemented in a working prototype. Before we discuss the various modules themselves, we first pay attention to an overarching trait of EAGLE, which is the support for notifications and subscriptions.

The implemented subscription mechanisms allow people to subscribe to specific architectural knowledge in discussion forums, but also to non-architectural news such as the news headlines or the weather report. In the right side of Fig. 8.2 this is visualized: here the user has subscribed to the BBC news website using RSS feeds. RSS feeds are employed to push relevant architectural knowledge to certain stakeholders, or to notify subscribed stakeholders if new architectural knowledge emerges. RSS feeds are complementary to the more traditional pull mechanism of using the best practices database. Users can subscribe to certain topics of interest and get updated without having to search the portal themselves, which is a lightweight approach to share architectural knowledge among relevant stakeholders.

The notifications are all shown in a comprehensive way in one of the panes of the portal's start page; in Fig. 8.2 the notification pane is found in the top left part of the page. Two types of notifications are supported in the current version of the portal: 1) checking the expiration dates of documents, and 2) informing users of contributions of discussion topics on which they have subscribed to. Since the first service performs an action that is not initiated by a user, the portal executes a script every night which checks the expiration date of documents and adds notification records in case of expired documents. For the second service, users need to be subscribed to discussion topics.

The subscription and notification mechanisms ensure that users can decide what information is displayed to them and in what way. Furthermore, a big advantage of notifications is that users are triggered to regularly check whether there is new information for them, thereby further increasing the *stickiness* of the portal.

Best practices repository

To store best practices or other reusable assets, the portal contains a best practices repository. This repository is one of the architectural knowledge modules that explicitly *supports architectural knowledge codification*. Architectural knowledge is codified in predefined formats, and could be retrieved for various purposes, such as reusing past design decisions, or to find out what guidelines exist on a certain topic.

The main contribution of the best practices repository is that it supports the decision making process of architects. Architects can fill in questions to specific architectural knowledge topics, after which the tool offers a number of alternative solutions. Without posing too much restrictions, the tool thus offers insights in the design space, which is especially useful for the less experienced architects.

Document repository

Documentation is stored in EAGLE's document repository. Although primarily intended for documents containing architectural knowledge, it is also possible to store all kinds of additional useful material, such as budget plans, memos, or corporate guidelines. As a result, similar to the best practice repository this module offers explicit *support for architectural knowledge codification*.

At first sight the document repository is not that different from existing content management systems. Two main features, however, distinguish it from the more standard systems. Firstly, an underlying category model enables classification of documents in various architectural knowledge categories. The implemented search function, allows searching documents by selecting on or more categories. Secondly, it is possible to assign an expiration date to documents. When the expiration date is reached, the owner of the document is signaled using the notification system discussed before, after which he needs to either update the document or delete it. This feature prevents that outdated documents reside in the repository, so that the overview and management of important documents is improved.

Please note that this module of the portal is not specific to architectural knowledge in the sense that it does not (yet) contain searching within documents. Nevertheless, architects – and other related stakeholders such as managers – can benefit from a proper way of organizing documentation, since the amount of documentation usually produced in the architecting process is rather large.

Yellow pages

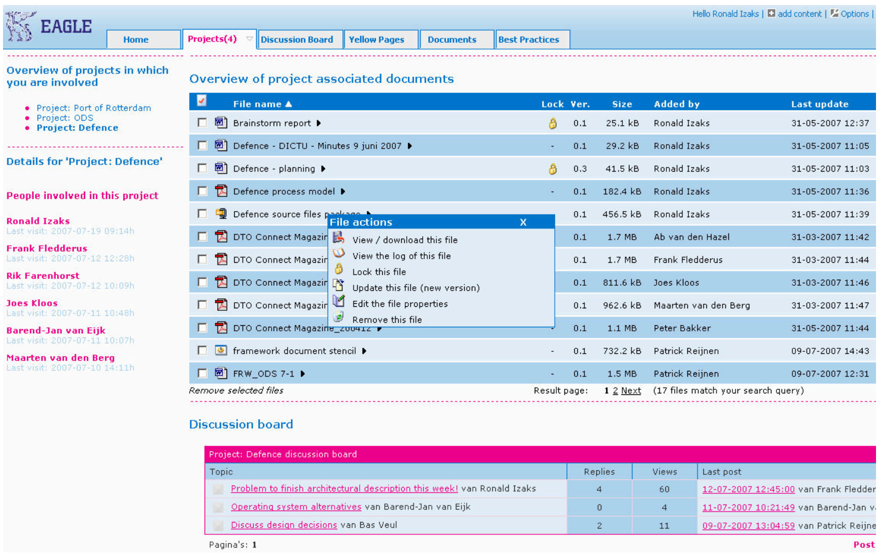
EAGLE allows architects to quickly find information about each other using the yellow pages module. Architects can obtain an overview of all other portal users. By selecting the name, a more detailed information page is shown with the credentials and contact information of that person. Also detailed architectural knowledge, such as the expertise areas of the user, and which projects and activities he is assigned to, is accessible through the yellow pages. This allows retrieving 'who is doing what', and 'who is knowing what', both of which are valuable information for architects.

The yellow pages system is a good example of *support for architectural knowledge personalization*, because the actual architectural knowledge itself is not presented, but it is indicated who possesses the knowledge. This functionality thus allows users to easily find colleagues based on experience, interests or projects on which they work, after which these colleagues could be contacted by sending them a message using the portal, or more traditionally by simply calling or emailing them.

By connecting people in the architecting process, we increase 'team building' in the organization, and foster discussions that can result in higher quality solutions. This approach also conforms to the *sticky in nature* property, because people like to share their ideas more easily if they have a 'group feeling', and creating such an environment ensures a certain degree of stickiness.

Discussion boards

The functionality offered by EAGLE's discussion boards is rather straightforward. It provides *support for collaboration* between people that are otherwise disconnected (for



Overview of projects in which you are involved

- Project: Port of Rotterdam
- Project: ODS
- **Project: Defence**

Details for 'Project: Defence'

People involved in this project

Ronald Izaks
Last visit: 2007-07-19 09:14h

Frank Fledderus
Last visit: 2007-07-12 12:28h

Rik Farenhorst
Last visit: 2007-07-12 10:09h

Joes Kloos
Last visit: 2007-07-11 10:48h

Barend-Jan van Eijk
Last visit: 2007-07-11 10:07h

Maarten van den Berg
Last visit: 2007-07-10 14:11h

Overview of project associated documents

File name ▲	Lock	Ver.	Size	Added by	Last update
Brainstorm report	🔒	0.1	25.1 kB	Ronald Izaks	31-05-2007 12:37
Defence - DICTU - Minutes 9 juni 2007	-	0.1	29.2 kB	Ronald Izaks	31-05-2007 11:05
Defence - planning	🔒	0.3	41.5 kB	Ronald Izaks	31-05-2007 11:03
Defence process model	-	0.1	182.4 kB	Ronald Izaks	31-05-2007 11:36
Defence source files package		0.1	456.5 kB	Ronald Izaks	31-05-2007 11:39
DTO Connect Magazin		0.1	1.7 MB	Ab van den Hazel	31-03-2007 11:42
DTO Connect Magazin		0.1	1.7 MB	Frank Fledderus	31-03-2007 11:44
DTO Connect Magazin		0.1	811.6 kB	Joes Kloos	31-03-2007 11:46
DTO Connect Magazin		0.1	962.6 kB	Maarten van den Berg	31-03-2007 11:47
DTO Connect Magazine_200412	-	0.1	1.1 MB	Peter Bakker	31-05-2007 11:44
framework document stencil	-	0.1	732.2 kB	Patrick Reijnen	09-07-2007 14:43
FRW_ODS 7-1	-	0.1	1.5 MB	Patrick Reijnen	09-07-2007 12:31

Remove selected files Result page: **1** 2 Next (17 files match your search query)

File actions X
- View / download this file
- View the log of this file
- Lock this file
- Update this file (new version)
- Edit the file properties
- Remove this file

Discussion board

Project: Defence discussion board

Topic	Replies	Views	Last post
Problem to finish architectural description this week! van Ronald Izaks	4	60	12-07-2007 12:45:00 van Frank Fledder
Operating system alternatives van Barend-Jan van Eijk	0	4	11-07-2007 10:21:49 van Barend-Jan v
Discuss design decisions van Bas Veul	2	11	09-07-2007 13:04:59 van Patrick Reijne

Pagina's: **1** **Post**

Figure 8.3: Project environments: documentation, discussions, and yellow pages

example due to physical distance) and it offers a lightweight way to discuss ideas or to share experience. An intriguing feature, however, is that if a user has subscribed himself to a discussion forum, as soon as somebody posts a message in a discussion topic, a notification is sent to him. This tracking mechanism allows keeping up-to-date on the progress and issues of activities and tasks users are involved with.

As an additional feature, depending on how many posts a users makes on a forum, his status is changed. As a result, heavy users of the discussion forums are rewarded by a higher status, which is depicted as avatar below their name. Such support for reputation tracking appeals to users and motivates them to keep using the portal, hence increasing its *stickiness* (Bush and Tiwana, 2005).

Project environments

EAGLE offers the architects the opportunity to organize their work on a per project basis in so called project environments. Essentially, a project environment combines functionality offered by three other architectural knowledge modules: document management, discussion boards, and yellow pages. This combination is clearly visible in Fig. 8.3 where at the left side information regarding the associated project stakeholders can be found (yellow pages), in the center the project documents are listed (document repository), and at the bottom the projects' discussions are shown (discussion board). An added benefit of this integrated functionality is that it supports *codification* (document management), *personalization* (yellow pages), and also *collaboration* (discussion board). To ensure that classified project information is contained, administrators of EAGLE can use access control structures to only allow specific stakeholders to access certain project environments.

Blogs

Blogs are employed to allow designers and architects to easily communicate and collaborate. As a result, other stakeholders can quickly acquire information about the current status of the project, such as the design decisions that have been made, alternatives that have been considered, etc. One important motivation for people to blog is the ability to create a community feeling (Nardi et al., 2004). To foster communication between architects, and to motivate them to share architectural knowledge, such a community feeling is essential. This may also further stimulate *collaboration*.

8.8 Vision: Towards a Blackboard System

In the previous sections we have elaborated upon EAGLE's key architectural knowledge modules. Our long-term goal is, apart from improving these modules, to turn EAGLE into a typical blackboard system. A blackboard system is a common approach to assist problem solvers in cooperation and communication (Englemore and Morgan, 1988). With various background services – called 'knowledge sources' in the Blackboard paradigm – architects can be supported in decision-making, architectural knowledge retrieval, etc. Since the current design of EAGLE already uses a central database where all architectural knowledge is stored, the foundations for such a blackboard system are already met. Features to further improve architectural knowledge sharing can be defined on top of this basic infrastructure. One of these features is that of text mining, see for example (Fan et al., 2006). Text mining support can be utilized to enrich unstructured architectural knowledge present in EAGLE.

To visualize how different services can collectively operate on the global database (i.e., the blackboard), consider the following hypothetical scenario that illustrates the interplay between the blog, RSS feeds, and a (future) text mining service, which is also depicted in Fig. 8.4.

A developer and architect who both work on project X have a discussion about how to best implement a public key infrastructure and they discuss various alternative

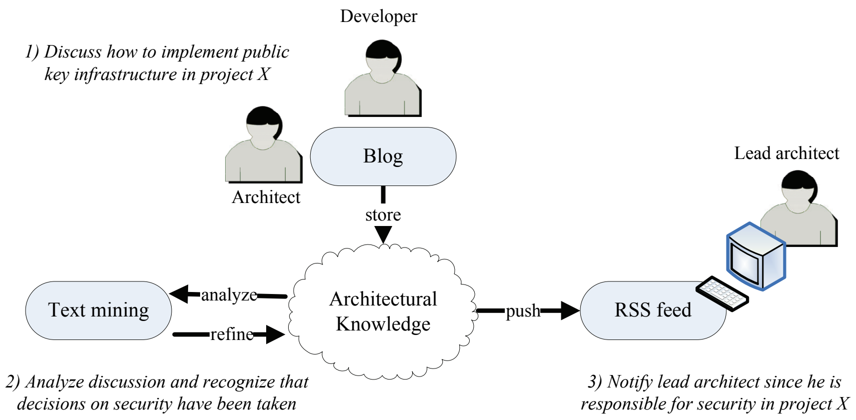

Developer
1) Discuss how to implement public key infrastructure in project X
Architect
Blog
store
Lead architect
Text mining
analyze
refine
Architectural Knowledge
push
RSS feed
2) Analyze discussion and recognize that decisions on security have been taken
3) Notify lead architect since he is responsible for security in project X

Figure 8.4: Interplay of architectural knowledge sharing services

solutions. Since they reside at different locations, they use a blog for their discussion. The blog, being one of EAGLE's architectural knowledge modules, integrally stores the discussion. This unstructured information is not very meaningful to stakeholders who are not directly involved in the discussion, but a structured summary is valuable to the lead architect of the project, since in our example he is ultimately responsible for the architecture design in Project X, including security. To this end, a text mining service, also part of EAGLE, opportunistically analyzes the blog discussion and determines that a decision on security has been made. Consequently, a summary of this architectural knowledge is sent to the lead architect using a RSS feed. Although the lead architect is unaware of the blog discussion that has taken place, he is able to determine whether the decision made by the architect and developer is the correct one. If this is not the case, he could intervene in time, preventing possible security problems later on in the project.

We strive to gradually develop EAGLE as an integrated portal that supports the community feeling between stakeholders in the architecting process. A survey carried out by Clerc et al. (2007a) indicated that architects consider themselves a middleman between various stakeholders such as business management and the customer on the one side, and technical oriented developers and stakeholders on the other side. EAGLE puts the architects centrally in the architecting process, by offering easy access to all kinds of codified and personalized architectural knowledge. The support for collaboration, the stakeholder-specific content, the easy manipulation of content, and the fact that EAGLE is both descriptive and sticky, further add to the effectiveness of this tool.

To arrive at an integrated environment, we should also explore the options for how we can further integrate EAGLE with existing tools that architects use in the architecting process, such as office tools or modeling tools. Architects can then use this integrated environment for all their daily tasks, which further motivates them to use the portal.

8.9 Conclusions

Software architecting is a knowledge intensive process. Consequently, tool support for architectural knowledge sharing has various benefits, such as reusing best practices, teaching staff, and support efficient collaboration between stakeholders. However, our observations in software architecture practice show that architects often stick to traditional tools, such as office suites, for their daily work, thereby missing the opportunity to effectively share architectural knowledge.

In this chapter we have defined seven properties that architectural knowledge sharing tools should have to be effective. This directly answers research question RQ-II.6. These properties have been defined by drawing on experience and literature in both the software architecture and knowledge management domain. By viewing software architecture from a knowledge management perspective we were able to determine which best practices from this field apply to the architecting process. Although the identified properties are essential from an architectural knowledge management perspective, it should be noted that some of these properties might to a certain extent apply to other Software Engineering disciplines as well.

Based on the properties, we have assessed a number of existing software architecture tools. The results of this assessment indicate that the status quo of architectural knowledge sharing tool support lacks full conformance to the seven desired properties.

To improve the status quo, we have presented the design and implementation of EAGLE, an architectural knowledge portal. This portal offers a hybrid architectural knowledge management approach and supports collaboration between stakeholders in the architecting process. To this end, it incorporates lightweight features using state-of-the-art techniques, such as blogs, RSS feeds and text mining. We have shown that our portal conforms to all identified properties, thereby increasing the chance for successful widespread adoption in software architecture practice. Because of EAGLE's conformance to all seven desired properties of architectural knowledge sharing tools, we argue that it is a better alternative compared to the tools discussed in §8.6, and a good step towards effective tool support for sharing architectural knowledge.

Our next step is on assessing the contribution of the prototype of the portal in practice. To this end, as part of the next case study at RFA we will ask a rather large group of architects to experiment with EAGLE and report their experiences.

9 Just-in-Time Support for Architects

In this chapter we extend our search for effective tool support by examining the intended, actual, and desired approach to sharing architectural knowledge according to architects at RFA. This study indicates that architects are best supported by an integrated tool environment that supports Just-in-Time architectural knowledge. Since EAGLE, our web portal, seems to conform well to these requirements, we experimented with it at RFA to further assess its practical value.

9.1 Introduction

Observations in Chapter 8 showed that architects in industry have yet to meet a tool environment that matches their knowledge needs. The main problem seemed to be a misalignment between the knowledge managed by these tools and what architects in practice really need for their daily tasks, which could be due to the specialized nature of these tools. In this chapter, we report on a third case study at RFA in which we delved further into this problem and gained understanding on how to overcome it. To this end, we assessed the architects' satisfaction with existing tools that support knowledge sharing in this organization, followed by the identification of their requirements for an improved tool environment.

In this third case study we focused on respectively the intended approaches to sharing architectural knowledge at RFA, the actual way this is done, and the desired situation according to the architects. We found that architects are not particularly concerned with specialized architectural knowledge reflected in meta-models, templates or process guidelines. Instead, they seemed primarily interested in support for *'Just-in-Time (JIT) architectural knowledge'*, which we define as access to and delivery of the right architectural knowledge, to the right person, at any given point in time. Such architectural knowledge may include updates on major decisions made or discussions held, but also contact information or expertise of important stakeholders. Since architecting is such a knowledge-intensive decision-making process, Just-in-Time architectural knowledge is vitally important for architects to ensure high-quality results.

The preference for Just-in-Time architectural knowledge seemed to match the functionality offered by EAGLE, the web portal we introduced in Chapter 8. To assess the practical value of EAGLE, we experimented with it and evaluated whether it could deliver Just-in-Time support to architects. This study therefore aligns well to typical *action taking* and *evaluation* phases of our action research cycle discussed in Chapter 6.

Our experimentation indicated that EAGLE's integrated functionality supports architects in their decision making process, by providing easy access to the right architectural knowledge at any given point in time. Experimentation with our portal also indicated that it is a definite improvement over existing tools at RFA.

The remainder of this chapter is organized as follows. In §9.2, we argue why Just-in-Time architectural knowledge is important. In §9.3, we outline the research design of this case study, which consisted of a problem-oriended and solution-oriented part. The results of the former part, in which we identify requirements for architectural knowledge sharing tool support, are discussed in §9.4. Results of latter part, which focuses on the application of EAGLE to enable Just-in-Time architectural knowledge, are described in §9.5 and in §9.6.

9.2 The Importance of Just-in-Time Knowledge

Many practitioners and researchers of the knowledge management community argue that instead of browsing numerous documents and other knowledge sources, ideally people like to compile, capture and receive a smaller and readily digestible volume containing only the really relevant knowledge needed at that moment. The concept of furnishing or making accessible the right knowledge to the right person at any given point in time is known as *"Just-in-Time Knowledge Management"* (Conteh et al., 2006). The importance of Just-in-Time knowledge is further stressed by Kerschberg and Jeong (2005), who argue that effective decision-making demands that the decision-makers are able to "sift through the mountains of data to find the right knowledge nuggets at the right time".

Access to and delivery of relevant knowledge at the right time is particularly important for software architects. This need for Just-in-Time knowledge follows from the fact that software architecting inherently is a decision-making process. This insight has matured over the past few years, as illustrated by the rather dominant decision-centric view on architectural knowledge, as discussed in Chapter 3.

During their decision-making process, architects are in a constant need for access to relevant architectural knowledge in order to make well-founded design decisions. Architects often maintain, implicitly or explicitly, a ‘backlog’ of smaller needs, issues, problems they need to tackle, and ideas they might want to use in the architecting process (Hofmeister et al., 2007). This backlog drives the workflow, helping the architect to determine what to do next. We argue that working on the backlog demands support for JIT architectural knowledge, i.e., access to and delivery of the right architectural knowledge, for the right person, at any given point in time. This way, architects can better discuss open issues, inform other stakeholders, or retrieve specific expertise.

Support for JIT architectural knowledge can be eased by using tools, so that it becomes easier to sift through the vast amounts of architectural knowledge available. Over the past few years, several tools have been proposed to support knowledge sharing in the architecting process, most of which focus specifically on managing architectural design decisions (Jansen et al., 2007; Capilla et al., 2007) and rationale (Tang et al., 2007; Falessi et al., 2006; Ali Babar and Gorton, 2007). All these tools follow a typical codification strategy, which aims to systematically store knowledge in predefined formats so that it can be easily found and reused. However, in order to support access to other kinds of architectural knowledge, such as expertise or experience of colleagues, codification alone does not suffice; architectural knowledge that is hard to articulate is easier shared using a personalization strategy. When using this latter strategy, not the knowledge itself, but information about its source or ‘owner’ is stored, after which they can use their personal network to share knowledge.

The importance of personal networks in knowledge sharing is also noted by Huysman and Wulf (2006), who conducted studies on practices of knowledge sharing in industry. They found that when sharing experience, people prefer to look for support from personal networks rather than from electronic networks to gain knowledge about the knowledge. This way, experience – or other tacit knowledge – does not need to be transformed into explicit knowledge to be shared. They argue that knowledge sharing tools should provide an infrastructure for establishing, maintaining or intensifying relationships in communities. Translating this requirement to the architecting process, we argue that JIT architectural knowledge is best supported by tools that not only codify important architectural knowledge, but also help stakeholders to find each other, so that architectural knowledge can be shared using personalization techniques as well.

9.3 Research Design

The question central to this third case study at RFA is what architects’ specific architectural knowledge sharing needs are, and how best to fulfill these needs. Although architects in this organization have access to several tools that support sharing architectural knowledge, the earlier case studies at RFA indicated that they struggle with how best to use them in their daily work.

The third case study consisted of two main parts. The first part was more problem-oriented and diagnostic in nature. The main objective of this part was to analyze how architectural knowledge sharing can best be supported in RFA. For this analysis we used a user-centered design method, which is designed around the assumption that people usually consider it easier to indicate what they dislike, instead of only pointing out positive aspects (Hoorn, 2006). Since the architects at RFA already have architectural knowledge sharing tools at their disposal, we were able to use this method to analyze the quality of these tools and identify possible room for improvement. Our analysis consisted of three consecutive steps, which are elaborated in turn below.

1. An analysis of the **intended** approach to architectural knowledge sharing. Over the past few years, four different tools have been introduced at RFA's architecture department to support architectural knowledge sharing. To elicit the original requirements of these tools, we have conducted semi-structured interviews with four managers from this department who have been responsible for introducing the tools, and who are now responsible for their maintenance. Based on these interviews we could determine how these tools should ideally support architectural knowledge sharing. The results of this first step are elaborated in §9.4.1.

2. An analysis of the **actual** approach to architectural knowledge sharing. The first step helped us to determine how the existing tools should ideally support the architects in sharing architectural knowledge. In this second step, we verified how well the intended support is actually perceived by the architects themselves. The results of this step are elaborated in §9.4.2.

3. An analysis of the **desired** approach to architectural knowledge sharing. In the second step the architects indicated the limitations and issues of the existing architectural knowledge sharing tools. In this third analysis step the same architects were explicitly asked how this situation could best be improved, i.e., how they perceive the ideal tool support for sharing architectural knowledge. This elicitation helped us to identify a set of desired features, based on which we were able to identify a number of requirements that future tools supporting architectural knowledge sharing should meet. All these requirements are further elaborated in §9.4.3.

The second part of the case study was solution-oriented. The requirements distilled during the first part of the case study mapped well with the functionality of EAGLE, our architectural knowledge portal introduced in Chapter 8. In §9.5 we sketch how EAGLE may act as therapy to the problems identified in the first part of this case study. To evaluate whether the practical value of our web portal was acknowledged by RFA's architects, i.e., whether it matches their desired approach to sharing architectural knowledge, we let the architects experiments with it. The key results of this experimentation exercise are elaborated upon in §9.6.

9.4 Diagnosis: Architectural Knowledge Sharing Approaches in Practice

In this section we elaborate upon the diagnostic part of the third case study at RFA. In the following three subsections we respectively discuss the analysis results of the intended, actual and desired approach to architectural knowledge sharing in RFA.

9.4.1 Intended approach to architectural knowledge sharing

In this first analysis step we analyzed the four different tools available in RFA to support architectural knowledge sharing. We interviewed the four managers who have been responsible for introducing the tools, in order to retrieve the original requirements of these tools. We classified these requirements as depicted in Fig. 9.1. In the remainder of this section we discuss the main requirements in more detail.

- **Best practice repository.** RFA has developed a knowledge repository that is primarily intended to support the construction of architectural descriptions. This support requires that the architects are offered guidance in their decision-making process. The repository allows storing architectural best practices so that these can be reused in future projects. Example best practices include references to conflicts between technology platforms, reference architectures from customers, or trade-offs between quality criteria. After answering a number of predefined questions, the architect is assisted by the repository, which uses its best practice to advise the architect about the architectural solution. During the interview with the manager responsible for the repository, we elicited that reusability of architectural knowledge is envisioned as main strength of the repository. Reusing best practices helps architects to more efficiently arrive at the most suitable architectural solution.

- **Expertise site.** This intranet website uses Microsoft Sharepoint as underlying technology. Its main purpose is to support community building among the architects at RFA. Four sub-requirements were identified during the the interview held

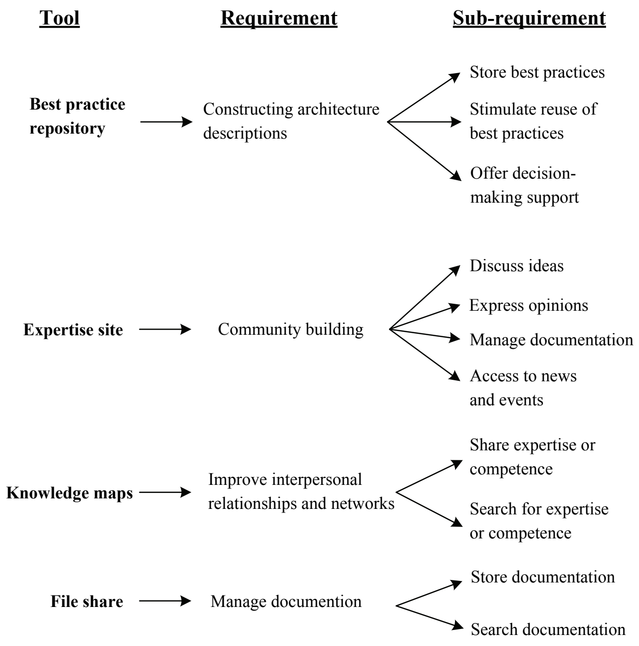

Tool
Requirement
Sub-requirement
Best practice repository
Constructing architecture descriptions
Store best practices
Stimulate reuse of best practices
Offer decision-making support
Expertise site
Community building
Discuss ideas
Express opinions
Manage documentation
Access to news and events
Knowledge maps
Improve interpersonal relationships and networks
Share expertise or competence
Search for expertise or competence
File share
Manage documention
Store documentation
Search documentation

Figure 9.1: Intended approach to sharing architectural knowledge

with the manager of the Expertise website: the ability to discuss ideas, the possibility to express opinions, a means to manage internal and external documents, and access to news, events, or other external information sources.

- **Knowledge maps system.** RFA has also developed an organization-wide knowledge maps system that aims to connect knowledge and knowledge workers. To meet this requirement, the system offers a place where users publish their expertise with respect to architecture-related topics, by filling in detailed user profiles. Users can use these profiles to search for colleagues with specific expertise or competences.

- **File share.** In addition to the other three – more specialized – architectural knowledge sharing tools, the architecture department of RFA uses a standard file share to manage all documentation. The original requirements of this system mentioned in the interview with its manager are nothing more than storing and searching for documents that contain relevant information for the architects.

9.4.2 Actual approach to architectural knowledge sharing

In this analysis step we interviewed eight architects from the total of 15 within the architecture department of RFA. This selected group of interviewees included junior and senior architects with various specialisms. The requirements and sub-requirements identified during the previous step acted as starting point for these interviews. We asked the architects what they liked and – more importantly – what they disliked about the requirements of the four existing tools for architectural knowledge sharing.

The architects were not really satisfied with the best practices repository. Due to a very non-intuitive user interface and low performance, using the tool is a time-consuming task. Moreover, in its current form the tool does not offer much support to decision-making. Although it offers storage for best practices, it does not indicate to the user which best practice is best to follow in a particular situation. Architects therefore see little value in the current implementation of the tool. In addition, the reusability of the repository is low because the content is outdated, and because adding or modifying the best practices is also time-consuming and error-prone, the costs for keeping the content up-to-date outweigh the benefits.

The Expertise site did also not particularly please the architects. This site, which is built as an intranet website, is not well accessible and its performance on RFA's network is low, too. As a result, the Expertise site is not often visited by the architects. Consequently, new discussion topics are seldom started, because architects doubt whether anybody will read them anyway. Another main problem of this tool is its non-intuitive user interface, which makes publishing knowledge on the site especially cumbersome. Architects therefore often resort to traditional communication means, such as email or phone, to communicate their ideas and experience.

The architects were particularly harsh on the knowledge maps system. In their opinion the main problem with this tool is that it lacks efficient search mechanisms. Consequently, the architects consider it difficult to quickly find the right knowledge workers within the organization. Likewise, they doubt whether their knowledge profile would be read often by colleagues. Due to the perceived low return on investment, architects often skip filling in such a profile, which was deemed a very time-consuming process, too.

The file share is used as the primary way of document management in RFA. Nevertheless, the architects are not really positive about its implementation. The architects' main problem with this tool is not in storing the documents (this is done using the standard Windows Explorer in Windows), but in retrieving them. Except for a standard folder structure there is no way to add meta-data. Moreover, the standard search functionality in Windows is not very flexible, which makes retrieving the right document a painful task.

In addition to the issues specific to the four existing tools, the architects reported one major problem of the current situation: the abundance of different information sources. As a result, architects have difficulty to easily retrieve specific architectural knowledge needed at a particular point in time, because they do not know where to start looking. It is not clear which source to trust more. A lack of trust and overview also results in a lack of motivation of contributing architectural knowledge to these knowledge sharing tools. After all, where can you best publish your knowledge?

From the above analysis we conclude that there is quite a mismatch between the intended and actual use of the four tools. All four tools have specific flaws that hinder widespread success and the lack of integration between the tools confuses architects which tools to use in which situation.

9.4.3 Desired approach to architectural knowledge sharing

During the interview round with the eight architects we also elicited their desired way of sharing architectural knowledge. We followed a similar approach as while identifying the intended approach to architectural knowledge sharing (see §9.4.1), only this time we focused on what the architects consider important requirements for any (future) architectural knowledge sharing tool. These requirements are further decomposed into sub-requirements whenever possible, after which we ranked them in order of importance based on how often they were mentioned by the interviewees. The resulting ranked classification of requirements is depicted in Fig. 9.2.

1. **Integration.** The requirement considered most important by the architects is that a tool environment should offer a central point of access to the various types of functionality available. This central point of access should be both attractive and intuitive. Attractiveness is key in the sense that it increases the chance for the tool's widespread adoption. Intuitiveness decreases the learning curve and makes using the tool fun. In addition, the architects noted that customization is important, because not all users have the same knowledge needs. Depending on current experience or interests, you might want to adapt the content shown or the user interface itself to your liking. Another requirement mentioned is that if new architectural knowledge emerges, a notification should be sent. This improves the overview users have on newly published architectural knowledge, which keeps them up-to-date. Finally, to further add to the integration strength, the architects also mentioned the need for links to external information sources, such as white papers, seminars and trainings, or other corporate communication.

2. **Project view.** The architects indicated that one major improvement for the current situation would be the support for a project view that enables management of

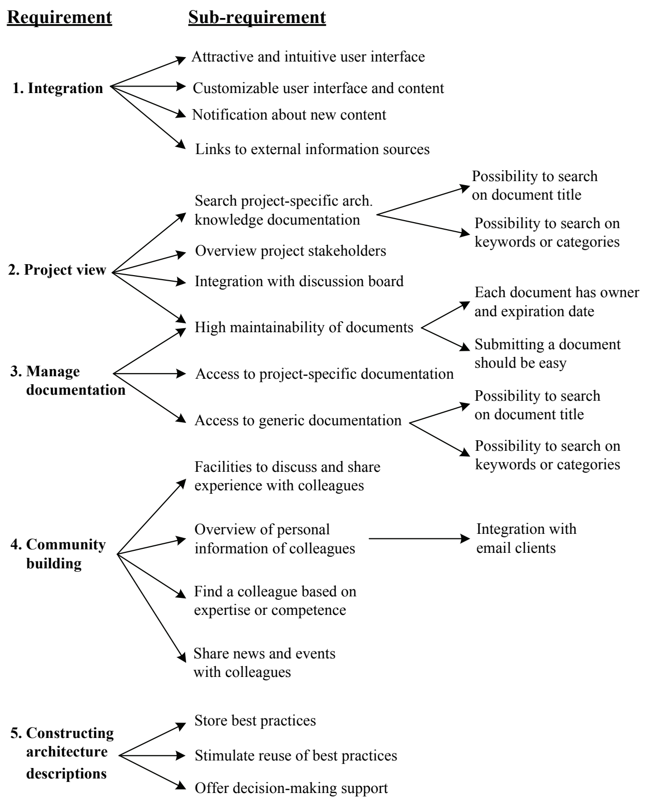

Requirement
Sub-requirement
1. Integration
Attractive and intuitive user interface
Customizable user interface and content
Notification about new content
Links to external information sources
2. Project view
Search project-specific arch. knowledge documentation
Possibility to search on document title
Possibility to search on keywords or categories
Overview project stakeholders
Integration with discussion board
High maintainability of documents
Each document has owner and expiration date
Submitting a document should be easy
3. Manage documentation
Access to project-specific documentation
Access to generic documentation
Possibility to search on document title
Possibility to search on keywords or categories
4. Community building
Facilities to discuss and share experience with colleagues
Overview of personal information of colleagues
Integration with email clients
Find a colleague based on expertise or competence
Share news and events with colleagues
5. Constructing architecture descriptions
Store best practices
Stimulate reuse of best practices
Offer decision-making support

Figure 9.2: Desired approach to sharing architectural knowledge

project-specific architectural knowledge. The main advantage of such a project view is that it offers a central point of access to easily search all architectural knowledge related to a particular project. For stakeholders that join a project at a later point in time, such a central point of access is helpful to quickly become acquainted with the ins and outs of the project. A sub-requirement that follows from this search requirement, however, is that the maintainability of documents is high. In addition, architects required that the project view should contain information about the project stakeholders. This information may include standard personal contact information, but also more architectural knowledge related content such as expertise areas of people. Finally, the architects indicated a need for discussion board functionality to be used by project stakeholders, so that issues, design decisions or conflicts can be quickly communicated.

3. **Manage documentation.** Related to the previous category is support for managing documentation. The difference with the project view is that the scope may be (much) broader, including all sorts of company documents. As with the project view requirement, searching documents was considered of prime importance by the architects, since this is one of the things that is currently implemented poorly. Consequently, sufficient meta-data has to be added to the documents in order to support intelligent search queries.

4. **Community building.** In contrast with the need for document management is the architects' wish to support building a community within their department. Although the architects acknowledge the power and importance of traditional conversations and meetings – both formal and informal – with respect to tool support they reckon it would be very helpful if there were facilities in place that help people to connect with each other. Consequently, requirements in this category include support for discussions and sharing expertise, but also overviews of 'who knows what' and 'who is doing what' in the organization. Finally, the ability to share news and events with colleagues would further add to the community feeling.

5. **Constructing architecture descriptions.** The last main category relates to one of the primary deliverables of the architects in RFA: architecture descriptions. These documents usually contain a variety of architectural knowledge, and usually take multiple days or weeks to construct. All sort of automated support during the process of making well founded decisions, followed by reflecting these decisions in the architecture description is highly appreciated.

If we compare the requirements classifications of the intended and desired approach to architectural knowledge sharing (see Fig. 9.1 and Fig. 9.2), we can make a few interesting observations. First of all, the requirements related to *'constructing architecture descriptions'* are mentioned both in the intended approach as in the desired approach. Obviously, the architects still like the underlying concepts, but are unhappy with the way the current tools implement these concepts. Secondly, although *'manage documentation'* was already an original requirement, architects at RFA take a consumer perspective and desire more focus on access to stored documents, instead of just storing them. For *'community building'* we observe the opposite trend. Here, the architects put more emphasis on publishing architectural knowledge, such as ideas, news, and other information; something which was poorly implemented in the current Expertise site. In addition, the sub-requirement related to finding colleagues based on expertise or competence suggests that architects not only rely on codification mechanisms, but also desire personalization strategies to share architectural knowledge. This wish for 'hybrid' architectural knowledge sharing is further stressed by the *'project view'* requirements, that indicate a need for both codification (e.g., document management) and personalization (discussion boards) techniques. Finally, the desire for *'integration'* is something that was obviously overlooked when designing the four existing tools.

9.5 Therapy: Architectural Knowledge Portal

The requirements identified in the previous section provide us with a good overview of the architectural knowledge sharing needs of the architects in RFA. The most important requirement is that of an integrated environment to share architectural knowledge. In addition to this need for integration, we conclude that architects are in need for what we defined earlier as Just-in-Time architectural knowledge. Requirement categories 2 till 5 of Fig. 9.2 demand various mechanisms to get easy access to available architectural knowledge. As discussed in the previous section, a hybrid strategy is needed to support both codification and personalization of architectural knowledge. The project view and community building requirements further show the need architects have for a tool environment that supports them in using their personal networks. Meeting these requirements demands specific personalization techniques.

The identified requirements are largely in line with the vision and scope of EAGLE, the architectural knowledge portal of which the architecture design and main modules were discussed in Chapter 8. First of all, EAGLE's modules enable both codification and personalization of architectural knowledge, a desired requirements of RFA's architects. Moreover, EAGLE's project environment (see Fig. 8.3), was one of the key requirements identified in §9.4.3. In a project environment in EAGLE, architectural knowledge is available in various forms, such as a list of the major project deliverables (center), a list of involved stakeholders (left), and an integrated discussion board where project stakeholders can discuss open issues (bottom). If necessary, access control measures can be used to ensure that only specific architects have access to the architectural knowledge stored.

Document management is supported by EAGLE by the document repository plugin. Instead of merely storing the documents, additional meta-data can be added to the underlying data model and documents can be classified using a tailored architectural knowledge category model that we designed together with the architects of RFA. Consequently, advanced search functionality is offered, such as searching for "all documents about Project X that have the status Final" or "all documents related to security written by John Doe". Using this search functionality, architects can quickly retrieve the documents that match their need.

Architectural best practices are stored in a repository that is added as a plugin to EAGLE. In this repository, architectural knowledge is codified in predefined formats, and could be retrieved for various purposes, such as reusing past design decisions, or to find out what best practices exist on a certain topic. In order to overcome the issues with the repository that were mentioned in §9.4.2, we have made the repository more intelligent, better maintainable, and better-looking.

Whereas the document repository and best practices repository are good examples of plugins that follow the architectural knowledge codification strategy, EAGLE also supports architectural knowledge personalization to fully comply to the community building requirements. To this end, it contains a 'yellow pages' plugin. On the yellow pages architects can get an overview of all other architects. By selecting the name, a more detailed information page is shown with personal information and contact information of that person. We are currently extending this information with more detailed information, such as the expertise areas of the architect, and which projects and activities he is assigned to. This allows retrieving knowledge about 'who is doing what', and 'who is knowing what' in the organization. Although at first sight this information is not directly pertaining to the architecture being designed, it can still be valuable information for architects, because it might tell them who to contact if they require help with specific architectural topics.

All the plugins mentioned above are accessible from EAGLE's start page. This start page acts as central point of access, and offers an intuitive user interface to ensure easy navigation. In addition, the portal also incorporates functionality to add personalized links to various information sources. Various RSS feeds can be loaded in the portal, allowing architects to access all sorts of non-architectural information via the portal as well, such as the daily news headlines, the weather report, etc. This coherence between all knowledge – architectural or not – is in line with the integration requirements identified in §9.4.3.

In addition to the main plugins described above, EAGLE has three main features, which will be elaborated upon below in turn:

1. **Integrated functionality.** EAGLE offers a central access point to various types of functionality by means of a start page. From this start page all important functionality can be accessed by the architects by one mouse click, after which they can quickly retrieve the architectural knowledge they need, using codification techniques, personalization techniques, or a combination.

2. **Stakeholder-specific content.** EAGLE offers an intuitive and attractive user interface. Since architects are already familiar with web pages, navigating the

portal is easy. Both the user interface and the content can be customized by architects. Different architects can thus focus on different types of architectural knowledge. A lead architect supervising a project for example would be mainly interested in what all architects are currently working on, and what their specific expertise areas are. A security architect on the other hand wants to be kept posted on specific developments in his domain, so he would be interested in documentation, discussions or news feeds related to this topic.

3. **Notifications and subscriptions.** EAGLE has a built-in subscription and notification system. Architects can subscribe to specific architectural knowledge topics (e.g., a topic of a discussion forum) or artifacts (e.g., a document). As soon as relevant architectural knowledge is published (e.g., another architect posts a message on the forum) or changed (e.g., a document expires or is replaced by a newer version) a notification is sent to all subscribed architects. The subscription and notification mechanisms allow users to keep up-to-date and provides access to relevant architectural knowledge at any point in time.

We argue that the above three features together ensure that EAGLE offers support for what we defined as Just-in-Time architectural knowledge. The integrated functionality provides access to *'the right architectural knowledge'*. The support for stakeholder-specific content ensures that *'the right person'* finds what he wants. Finally, the subscription and notification mechanisms allow architects to stay up-to-date by delivering the relevant architectural knowledge to them when needed.

9.6 Experimentation

In order to assess the practical value of EAGLE, we let 11 architects of RFA experiment with it. Among these 11 architects were the eight we had interviewed earlier during this case study, plus three additional ones. These latter three architects were included because we deemed them as more objective, so that the assessment results are more representative for the whole population.

The experimentation consisted of executing predefined scenarios that mapped on the requirements identified in §9.4.3. The architects had to execute each scenario using EAGLE (e.g., the scenario *"retrieve the newest version of the technical design of Project X, using the document repository plugin."*), after which they had to give scores for the implementation using a 5-points Likert scale. When comparing the scores from the three new architects with those of the eight others, we did not see any significant differences. The main results of the experiment are discussed below.

In its current form EAGLE is already an improvement over the existing tools that were in place in RFA. Most architects (82%) indicated that the document management properties of the portal are an improvement over the existing fileshare. Because of the categorization model and metadata that can be added to documents, retrieving documents is much easier. However, architects mentioned that a change in mindset is required before everyone is used to the new way of uploading and tagging documents. In addition, they noted that disciplined use of the portal remains an important prerequisite to success.

Although the majority of architects (91%) was particularly fond of the integration aspects of EAGLE, in which document management, project environments, discussion boards and personal contact information is integrated, they wanted the portal to integrate even more with existing tools of the department, such as email clients (send emails to colleague, send invitations for meetings, attach documents to emails, store documents from emails in the repository), calendars (todo lists in the portal), or project tools (assign people to tasks or activities using the portal).

All architects liked the way the combination of the notifications and subscriptions of EAGLE work. They deemed it considerably useful to stay up-to-date on architectural knowledge available that might be of interest. The fact that EAGLE has different types of notifications (e.g., 'document expired', 'new forum post') is highly appreciated, and the fact that architects are free to subscribe to architectural knowledge reflected in various ways (e.g., news, discussion boards, documents) is liked as well. Some architects mentioned that the notification and subscription system might also add to the attractiveness of the portal, in the sense that architects are motivated to visit it on a regular basis (to see if new relevant architectural knowledge is present). It therefore appears that EAGLE is to a certain extent 'sticky' to its users, which is considered an important prerequisite for successful adoption of knowledge management tools in practice (Bush and Tiwana, 2005) and identified as desired property of such tools in Chapter 8.

EAGLE's emphasis on providing access to the organization's vast amount of architectural knowledge is appreciated by all the architects. This portal in its current form supports access to and delivery of the "*right architectural knowledge on the right time*", and leaves sufficient freedom to the architects on how to visualize this knowledge. Apart from this support for Just-in-Time architectural knowledge, EAGLE emphasizes the social capital, i.e., supporting sharing in a community as opposed to individually consuming knowledge. As a result, by improving collaboration between architects of RFA, our portal is a good first step to create a real 'community of architects'.

9.7 Conclusions

In this chapter we have investigated what are the typical architectural knowledge needs of architects at RFA, and how these needs can best be fulfilled. We have identified five main requirements for an architectural knowledge sharing environment: 1) integration, 2) project views, 3) manage documentation, 4) community building, and 5) constructing architecture descriptions. Based on these requirements - that also form an answer to research question RQ-II.7 – we have concluded that architects are best supported by an integrated tool environment that supports Just-in-Time architectural knowledge.

To meet the above requirements we have experimented with EAGLE, our architectural knowledge web portal. Main features of this portal include integrated functionality to retrieve architectural knowledge, support for stakeholder-specific content, and a notification and subscription system. Architects can use EAGLE to connect to colleagues or other involved stakeholders by retrieving ‘who is doing what’ and ‘who knows what’. In addition, codified architectural knowledge in a document repository or best practice repository can easily be accessed using advanced search mechanisms. Finally, collaboration is explicitly supported by EAGLE’s discussion board and project environment.

By offering an integrated environment for architects that incorporates various functionality to easily get access to available architectural knowledge, we argue that EAGLE is able to deliver the continuous flow of relevant information that architects need when working on their backlog (Hofmeister et al., 2007). Moreover, our portal’s functionality to share architectural knowledge follows a hybrid architectural knowledge sharing strategy, combining both codification and personalization techniques. Because explicit attention to personalization is incorporated in the portal, it supports architectural knowledge sharing by focusing on social capital (Huysman and Wulf, 2006).

A last important characteristic of EAGLE is the balanced focus on architectural knowledge consumption (i.e., retrieving architectural knowledge), and production (i.e., publishing architectural knowledge). This distinguishes EAGLE from existing architectural knowledge tools, which often focus solely on the producing side (Avgeriou et al., 2007b).

Experimentation with EAGLE further indicated that it is already a definite improvement over the existing tools within RFA. Nevertheless, as future work we plan to extend our portal (e.g., with additional plugins) that further ease sharing of architectural knowledge. One of these extensions, as discussed in Chapter 10, features the use of wikis, allowing architects to easily produce architectural knowledge.

A final insight, of which the software architecture research community should take notice, is the fact that architects – at least the ones at RFA – apparently are not particularly interested in very ‘specialized’ architectural knowledge support, such as detailed meta-models, templates or process guidelines. They seem to already have their techniques and processes in place to design and maintain software architectures. What *is* important, however, is the facilitating support during their everyday decision-making process by means of continuous access to and delivery of relevant architectural knowledge.

10 Wiki Support for Architects

In this chapter we experiment with an enterprise wiki environment to assess how suitable it is in supporting architects in sharing architectural knowledge. Our experiments show that wikis are particularly strong in creating a 'community of architects, because they offer support for managing both architectural and 'non-architectural' knowledge. Moreover, the wikis versatility – demonstrated by its integrated and diverse functionality – makes it a worthy competitor of the web portal presented in earlier chapters.

10.1 Introduction

Wikis are becoming increasingly popular knowledge management systems in both software engineering research and practice, indicated by a number of recent papers about this subject (Bachmann and Merson, 2005; Schuster et al., 2007). We are particularly interested in the applicability of wikis in the software architecting process. During our fourth case study at RFA, we therefore experimented with creating and using an enterprise wiki platform in the architecture department of this organization. By working together with, and interviewing several architects, we were able to acquire a solid understanding of a wiki's potential in the software architecting process. In terms of placing this case study in an action research cycle it is similar to the previous one described in Chapter 9; the experimentation and evaluation of a tool environment corresponds well to the *action taking* and *evaluation* phases of action research.

Our experimentation showed that wikis can well be employed as environment for sharing architectural knowledge. One of its main strengths is strong community building and communication support, something our EAGLE web portal fell a bit short on, as reported in the previous chapter. Another strong point is the broad range of plugins available to increase the functionality of the platform. This makes a wiki an interesting platform to create a blackboard system – a vision described in Chapter 8 – that included various services to assist architects in decision making, architectural knowledge retrieval, and other knowledge-intensive activities.

In an attempt to more precisely define the potential merit of wikis in software architecting, in this chapter we reiterate over some main characteristics of software architecting. Based on this characterization we analyze what contribution a wiki could offer to a typical architecting process. To place our analysis results in context we compare the strengths and weaknesses of a wiki to some existing tools that support architectural knowledge management, including EAGLE. The comparison shows that whereas most other tools are stronger in codifying specific architectural knowledge concepts, a wiki prevails as versatile and all-round knowledge management environment. To assist researchers and practitioners in setting up such an environment, we propose a number of do's and donts for wiki usage in the software architecting process.

In §10.2 we reiterate over theories presented in earlier chapters to outline what knowledge needs architects typically have. In §10.3 we study in detail how suitable wikis are in addressing these knowledge needs using experimentations with an enterprise wiki. Based on these experiments in §10.4 we discuss how suitable wikis are for sharing architectural knowledge by comparing the wiki to other tools and by formulating a number of dos and don'ts.

10.2 Knowledge Needs of Architects

As explained before, software architects are typical knowledge workers. They are considered the 'spider-in-the-web' of the software development process. While communicating with both business and technical stakeholders, they need to negotiate and find a balance between these stakeholders' needs and concerns, take the important architectural design decisions, reuse existing architectural styles and patterns, evaluate and review architectural solutions and create architectural standards and guidelines.

To better understand the knowledge needs of architects, in this section we first reiterate over some trends in software architecting that have been introduced in earlier chapters. Then, based on these trends, we elaborate upon the knowledge needs of architects by focusing on 'architectural knowledge' and 'non-architectural knowledge', respectively.

With respect to how the software architecture field developed over the past few years, four trends can be observed:

- **Increased collaboration.** In recent years, software projects have increased considerably in size and so have the number of architects working on these projects.

In these projects collaboration takes place between various (types of) stakeholders, which was confirmed during our first case study at RFA (see Chapter 7). Architectural discussions are held between large groups of stakeholders, who vary in background and expertise. Consequently, the knowledge flows in the architecting process are diverse and frequent, which calls for proper support to enable efficient communication and collaboration between team members.

- **Focus on decision making.** Architecting is a typical consensus decision making process, as explained in Chapter 8. In order to take major architectural decisions, reaching consensus between the main stakeholders is vital. Architectural decisions can have a relatively large impact on the final system, because they constrain lower-level design issues. This trait makes these decisions often irreversible, making the need to formulate and communicate a proper rationale for these decisions even higher. Therefore, architects spend significant amount of time in negotiating and discussing (both verbally and by means of documentation) with the parties involved, and in making tradeoffs between various design alternatives to reach consensus. During this process the architect also needs to guarantee that decisions taken do not conflict with company regulations, reference architectures or other standards and rules.
- **Distributed development.** Architecting is often done by architects who are spread over multiple teams in one organization. When the organization has multiple sites, online communication replaces traditional coffee room talks, which are not an option anymore to share valuable knowledge. Global Software Development (GSD) affects the way the architecting process is organized even further. Architectural knowledge management needs to take into account the challenges that arise as a result of the geographical, temporal, and socio-cultural distance innate to GSD (Clerc, 2008).
- **Need for reusable assets.** With the maturation of the software architecture field in both research and practice, architectural solutions (e.g., patterns and styles) are better formulated, using well-documented architectural views become common practice, and architectural standards and principles are more frequently embraced by the stakeholders. Consequently, architects increasingly benefit from such reusable assets; they do not always have to reinvent the wheel, but instead can learn from prior experience. In order to reuse such architectural knowledge, retrieving the right knowledge at the right time is considered crucial (see Chapter 9).

In a modern-day architecting process that is characterized by the trends described above, architects produce and consume much knowledge. To accommodate architects in all their knowledge-intensive activities, proper support for knowledge sharing should focus on both 'architectural knowledge' and 'non-architectural knowledge':

- **Architectural knowledge.** As presented in Chapter 3, architectural knowledge can be classified using two dimensions: tacit vs. explicit and application-generic vs. application-specific. Application-specific architectural knowledge is important knowledge to understand why an architecture for a software system is the way it is. It is often defined as the set of design decisions, including the rationale for these decisions, together with the resulting architectural design. Application-generic architectural knowledge is knowledge that is more often internalized as tacit knowledge and built up by the architect as experience, domain knowledge or expertise. It includes architectural styles, patterns, and tactics, and knowledge about when and how to use specific technologies, tools and methods. Application-generic architectural knowledge enables practitioners in producing or consuming application-specific architectural knowledge (e.g., taking a high-quality architectural decision based on a proper rationale) and is therefore also very important, as confirmed by research of Clements et al. (2007) about the duties, skills and knowledge of architects.

- **Non-architectural knowledge.** Apart from the need for architectural knowledge, architects also require 'non-architectural' knowledge to perform their daily tasks, just like other IT professionals do. This non-architectural knowledge is very broad but equally crucial for the architecting process as architectural knowledge is. Non-architectural knowledge includes process management information, information about the organization and the context and domain, information about the business processes of the internal and customer organization, knowledge about both architectural and non-architectural standards (e.g., TOGAF, Sabanes-Oxley), and knowledge about the current status in the organization (e.g., 'who is doing what' or 'who knows what').

Both types of knowledge need to be managed and shared to effectively assist architects in their daily work. Tools reported in literature often focus on modeling, retrieving and editing architectural design decisions, rationale and related architectural concepts. Examples include the tools that were examined in Chapter 8: ADDSS (Capilla et al., 2006, 2007), DGA DDR (Falessi et al., 2006), PAKME (Ali Babar and Gorton, 2007), and Archium (Jansen et al., 2007). However, support for collaboration, decision making and distributed development is not always strongly supported by these tools, which raises the question whether they are suitable as overall knowledge sharing platform in software architecting. In previous chapters we have discussed EAGLE, a web portal for architectural knowledge. Whereas this portal offers a range of methods for sharing knowledge, architects indicated that with respect to collaboration and communication support, more advanced support would be welcomed. It is therefore only a logical choice to investigate how well wikis meet these requirements.

10.3 A Wiki in the Architecting Process

Ever since the advent of Wikipedia, the interest in wiki-based systems is growing steadily. Numerous wiki software packages have been proposed and commercial organizations are increasingly interested in setting up collaborative work-spaces where people meet, produce and consume knowledge. Within the software engineering community the interests in wikis is growing as well. Bachmann and Merson (2005) describe their experience with applying a wiki environment for supporting architecture documentation. Schuster et al. (2007) propose a wiki-based tool to document architectural design decisions.

During the fourth case study at RFA we experimented with an enterprise wiki environment in the architecture department. Initially, we wanted to develop a wiki-plugin to EAGLE, the portal experimented with in the third case study. After some investigation we found that a more feature-rich and user-friendly option was to take an existing enterprise wiki tool, and further tune it to create an architectural knowledge sharing environment. This way a lot of 'out-of-the-box' collaboration and community-building features of the enterprise wiki environment could be reused, thereby improving exactly those characteristics of EAGLE that called for improvement (cf. Chapter 9, §9.6).

Our wiki is built using the Confluence Enterprise wiki software [1]. In the next four subsections we elaborate upon our experience by focusing on the context in which the wiki was used, the adoption strategy we followed, and how architectural and non-architectural knowledge was managed by the wiki, respectively.

10.3.1 Case study context

The experiments were conducted at one of the architecture departments of RFA. Architects of this department work on architectural solutions for local and national government. The department consists of around 30 enterprise and IT architects, who work on various projects at customer organizations and internal projects. At least once a week the architects visit the department's office for meetings and sharing experience and ideas with colleagues. Often just coffee room talks and small (often informal) meetings were used to this end, but also some repositories and an intranet website exist

[1] http://www.atlassian.com/software/confluence/

in which architectural knowledge is stored. None of the architects had been using a wiki environment before, so apart from being familiar to Wikipedia, they were new to this type of knowledge management platform.

The choice for the Confluence Enterprise wiki had been taken jointly by us (the researchers) and the management of the architecture department. Prior to our experiments the unit manager had asked one architect to study available wiki environments (e.g., MediaWiki, XWiki, wiki support in MS Sharepoint 2007) for their suitability. In a comparison that was made Confluence turned out to be the most promising wiki environment, mostly due to its ‘rich text editing’ features and the fact that a Confluence wiki is highly customizable using a plugin-system. Compared to e.g., MediaWiki (the environment on which Wikipedia is hosted) in Confluence wiki pages can be written in a way similar to MS Word (a tool frequently used by architects). With respect to the integration, Confluence made it possible to link from the wiki to existing Sharepoint repositories full of corporate information.

By choosing Confluence we ensured that the wiki was not becoming yet another stand-alone knowledge management environment, like several smaller systems that already existed in the architecture department of RFA. From earlier research we know that stand-alone tools might become difficult to maintain, are hard to keep up-to-date and confuse practitioners, who have difficulty determining which one to use for what purpose and where to find what architectural knowledge (cf. Chapter 9).

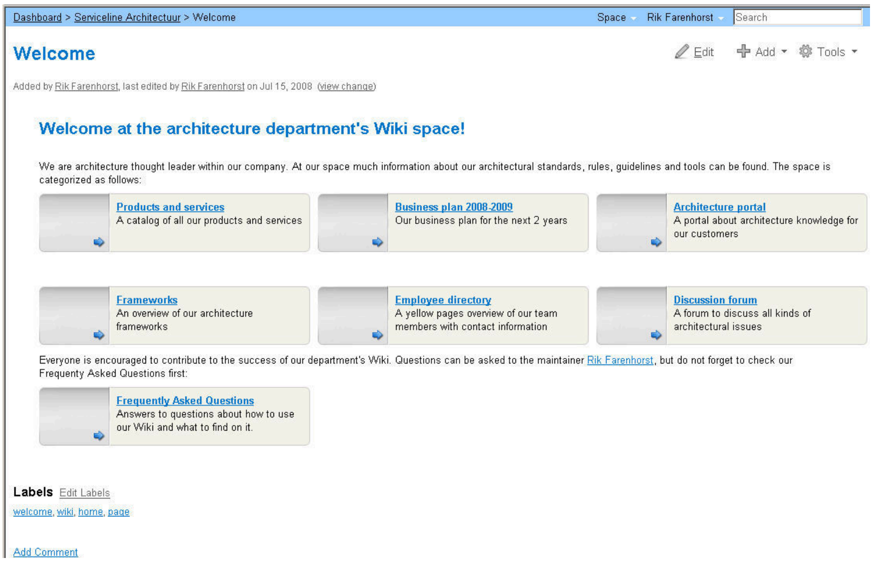
10.3.2 Adoption strategy

In order to increase the adoption of the wiki in RFA, we decided to start with generating “critical mass” on the wiki with three architects, before showing the wiki to the whole team. By following this Critical Mass pattern (Mader, 2007) we automatically used additional wiki patterns as well, such as the Champion pattern (one of us acted as wiki champion), the Moderator pattern (the three architects were appointed Moderators of specific parts of the wiki), and the FAQ pattern (we made a page with frequently asked questions to help wiki users with basic functionality and features). More useful information on these (and other) patterns can be found in (Mader, 2007) or online [2].

After working for two months in a core team, which consisted of the champion and three moderators, the wiki had been filled with initial content, the layout had been changed to match corporate styles, and the functionality and quality of Confluence had been thoroughly tested. A screenshot of the wiki’s start page is depicted in Fig. 10.1.

The next step was informing the other colleagues of the wiki’s existence. To this end an email was sent among the team members with the wiki’s address, login details

[2]http://www.wikipatterns.com



Welcome

Added by Rik Farenhorst, last edited by Rik Farenhorst on Jul 15, 2008 (view change)

Welcome at the architecture department's Wiki space!

We are architecture thought leader within our company. At our space much information about our architectural standards, rules, guidelines and tools can be found. The space is categorized as follows:

Products and services
A catalog of all our products and services

Business plan 2008-2009
Our business plan for the next 2 years

Architecture portal
A portal about architecture knowledge for our customers

Frameworks
An overview of our architecture frameworks

Employee directory
A yellow pages overview of our team members with contact information

Discussion forum
A forum to discuss all kinds of architectural issues

Everyone is encouraged to contribute to the success of our department's Wiki. Questions can be asked to the maintainer Rik Farenhorst, but do not forget to check our Frequenty Asked Questions first:

Frequently Asked Questions
Answers to questions about how to use our Wiki and what to find on it.

Labels Edit Labels
welcome, wiki, home, page

Add Comment

Figure 10.1: Screenshot of the start page of the wiki

and some general introduction to the wiki's aim and scope. This introduction email resulted in an immediate boost of wiki activity. Several architects created a personal wiki page within a day, and started adding comments to existing content on the wiki. Others uploaded documents to the wiki in an attempt to share their recent work to the group. It is fair to say that the enthusiasm was wide-spread.

The enthusiasm of the wiki reached other departments within RFA as well. Several colleagues from other departments came by for a demonstration of its capabilities, and became excited as well. We provided several of them with their own 'wiki spaces' so that they could further experiment or play with the wiki themselves. After half a year management of RFA noticed the wiki's popularity and decided to acquire a company-wide license and to place the maintenance of the wiki under the responsibility of the central IT department. As a result in the near future the wiki will be used in a more professional way throughout RFA.

10.3.3 Support for architectural knowledge

We have used the wiki for storing and sharing both architectural knowledge and non-architectural knowledge. Parts of the wiki were filled with reusable architectural principles and rules (one per wiki page) after which an overview page was generated using the 'table of contents' plugin. The same approach has been used for architectural knowledge concerned with architectural technologies and patterns.

It should be noted, though, that the architects had some trouble with structuring the architectural knowledge in a suitable way. Being used to writing documents and storing these in repositories with some meta-data, it took a mindset change before some architects understood that storing the knowledge in text-format directly in wiki pages, and labeling these pages for retrieval purposes is all there is to it in a wiki environment. The knowledge entities can be edited directly in wiki-markup, and the documents are redundant, although they might be archived (i.e., added as attachment) for users who wish to read more about the topic.

Unfortunately, the approach sketched above was not that easy. Adding meta-data such as specific properties of architectural decisions was quite cumbersome. Confluence does not provide much support for this. It does, however, offer some plugins that address this topic. One of these is a plugin that allows connections to SQL databases, but we found that this connection is read-only. Consequently, one cannot update database tables from within the wiki pages, which limits the integration with relational databases significantly. There is also a meta-data plugin in Confluence, but this offers only limited functionality compared to what specific databases based on their own data model could offer. Storing architectural design decisions was therefore not an easy task.

As a solution to model architectural knowledge in the wiki we are currently devising templates to be used by the architects to further codify the architectural knowledge concepts in a uniform way and to ensure that this knowledge is sufficiently enriched with meta-data that is stored with it. Further experimentation is needed to see if these templates offer sufficient guidance to the architects. We also plan to investigate whether we could 'borrow' best practices from the semantic wiki community (see e.g., (Schaffert et al., 2008)) to handle architectural knowledge concepts in a more mature way.

Another nuisance when working with the wiki was the support for images. Architects often use diagrams and models to depict their architectural views (Clements et al., 2002). They sometimes use specialized modeling tools to this end, but often also construct diagrams in MS Word or Visio. When storing architectural documentation in wiki pages the architects found that each image needs to be added as separate attachment and that editing images from within Confluence is not possible. Since architectural views are update frequently – based on stakeholder discussions and negotiations – this is a drawback when using the wiki as main storage place for architectural deliverables.

10.3.4 Support for non-architectural knowledge

To our surprise one of the main strengths of the wiki was not its support for managing architectural knowledge, but how it offers a central storage place for non-architectural knowledge that architects produce and consume during their daily activities. Architects put meeting minutes on the wiki, but also a holiday roster, and a list of the team members' birthdays. They also constructed a 'yellow pages' system by creating a wiki page for each team member, including contact information, current tasks and activities, and interests and expertise. From what we observed it is exactly these 'related' knowledge pages that made the wiki fun to use and to visit regularly. It attracted the architects to the platform, it decreased the emails being sent with attachments significantly, and it even started some healthy competition between a few architects, who checked the wiki's activity monitor on a daily basis to see who had the honor of being the 'most active contributor'.

10.4 Wikis for Architectural Knowledge Sharing?

Based on our experience with using a wiki in RFA, we are able to indicate the main strengths and weaknesses of this tool for its use in the architecting process. Compared with EAGLE our wiki offers increased support for collaboration and communication, both of which were requested by the architects of RFA during the previous case study. Moreover, having a intuitive user interface and strong usability, our wiki can be used as encyclopedia of architectural knowledge, as well as a community where architects can easily find each other. The easy integration with Microsoft Office tools (e.g., transforming a Word document in one or more wiki pages with just one mouse click), and corporate repositories (Confluence offers a Sharepoint connector to sync wiki content with Sharepoint sites) further places the wiki as central knowledge management environment for architects and other interested stakeholders. This integration with commercial tools would be rather costly to implement ourselves as extension to EAGLE.

In order to determine whether wikis are suitable to share architectural knowledge, in the next subsection we compare wikis to some state-of-the-art architectural knowledge management tools. For this comparison we zoom in on the four characteristics of the architecting process discussed in §10.2. This allows us to better define the relative strengths and weaknesses of a wiki environment when used as architectural knowledge management platform.

In §10.4.2 we list a number of dos and don'ts for using wikis in the software architecture domain. During our experimentation we found that several personal and organizational factors impact the success of a wiki in the software architecting process either positively or negatively. Researchers or practitioners who wish to set up a wiki environment themselves can use our dos and don'ts to increase the chance of success.

10.4.1 Comparison with other tools

As discussed in §10.2, in the architecting process:

1. A lot of *collaboration and communication* takes place due to the large software development teams and the projects these teams work on.
2. *Consensus decision making* is one of the primary activities that stakeholders spend their time on.
3. *Online communication* becomes more and more important due to distributed development which inhibits traditional coffee room talks.
4. *Retrieving architectural assets* is crucial to enable reuse and to deliver high-quality architectural solutions.

Below we compare architectural knowledge-specific tools (ADDSS, PAKME, DGA DDR, and Archium) with the Confluence wiki by zooming in on these characteristics of architecting.

When taking a close look at the existing architectural knowledge management tools listed before, we see that these offer specific support for codifying and retrieving architectural knowledge concepts. In Archium, ADDSS and DGA DDR the main concepts are design decisions, whereas in PAKME also patterns and styles are included. All of these tools use tailored databases to store and edit the knowledge and use an underlying data model to define the knowledge they focus on. Retrieving architectural knowledge assets is therefore well supported by these tools. As we have described in §10.3 the wiki does not have strong support for modeling architectural knowledge concepts out-of-the box, which makes it harder to retrieve such knowledge using the wiki than when using one of the architectural knowledge-specific tools.

When looking at the support for online communication, the wiki stands out. Current versions of ADDSS, DGA DDR, PAKME and Archium run locally on a pc and therefore cannot be collectively used by practitioners working on different sites. The wiki on the other hand is web-based and aimed at targeting communities. It supports concurrent editing of content, version management, and users can easily provide comments on wiki pages created by others.

As described in §10.3, a major strength of the wiki is its support for managing non-architectural knowledge, such as meeting minutes, plannings, deadlines, progress reports, etc. This non-architectural knowledge can be stored in the various wiki pages, and can be very supportive for stakeholders involved in consensus decision making. They could use the wiki as central archiving environment for all information used in software architecture projects. This makes the wiki a much more ‘all-round’ tool than the architectural knowledge-specific tools, which offer only codification techniques for architectural knowledge concepts.

Another drawback of the architectural knowledge-specific tools is that they fall short when it comes to supporting collaboration. In the architecting process architects typically work on designing and describing or communicating architecture solutions. For the description part the wiki offers a nice groupware platform where architects can collaboratively produce new information, review each others work, etc. And since architecture projects are growing larger and larger, involving more and more stakeholders, groupware support is much welcomed. In addition, the architectural knowledge-specific tools tend to have a rather high learning curve, whereas the wiki’s usability resembles both text-editors and websites, to which architects are already familiar.

10.4.2 Dos and don’ts for wikis in the architecting process

If we summarize the comparison described in §10.4.1 we conjecture that the wiki is strong in its support for collaboration, non-architectural knowledge and online communication, whereas the architectural knowledge-specific tools offer more thorough support for the retrieval of reusable architectural knowledge assets. Based on these results we conjecture that a wiki can make a fine architectural knowledge management environment.

Keep in mind though that a successful wiki is something that takes time to set up. In this section we provide some dos and don’ts when setting up a wiki environment in the architecting process. These dos and don’ts have been defined based on 1) multiple interviews and discussions with the architects at RFA who were most active on the department’s wiki, and 2) a review of available literature on wiki patterns (Mader, 2007), based on which we analyzed which ones are most significant in the software architecting process.

1. **[Do] start small.** Define a startup period in which preparations on the wiki environment begin. Produce an initial set of wiki content with a small group of people, who use this startup period to become familiar with the wiki system and the ins-and-outs of wiki markup. Appoint one or more wiki champions who will promote the wiki to colleagues and assign moderators who will safeguard the

quality of the knowledge stored in the wiki. Make sure that the champions and moderators have a clear idea on their responsibilities. Only when all the above has been arranged, the wiki should be announced to the whole team or department. Having more knowledge available from the start will induce architects to use the wiki as primary knowledge environment (cf. Chapter 7).

2. **[Do] allow and stimulate all types of knowledge.** Do not restrict the wiki to only very specialized architectural knowledge such as design decisions, architectural principles or standards. As discussed, a wiki is particularly strong in managing non-architectural knowledge, so make sure that users have sufficient freedom to manage such knowledge in the wiki as well.

3. **[Do] strive for integration.** To increase the chance that users consider the wiki an integral part of their daily work, integration with existing systems such as file shares, a Sharepoint site, etc. is crucial. Ideally the wiki is positioned as core system for codifying both non-architectural and architectural knowledge, and when necessary, links and traces to additional systems are made.

4. **[Don't] impose too much structure.** It might be very tempting to introduce lots of templates, rules and guidelines for users to follow or use. However, one of the wiki principles is that a wiki should become a self-organizing system in which users incrementally improve the quality and correct mistakes. Introducing templates, e.g., to model architectural knowledge concepts is a good thing to do, but do not overdo it, since it might kill the 'fun' of working with a wiki.

5. **[Don't] restrict access.** Try to keep the wiki contents open to as many users as possible instead of restricting access to specific stakeholders depending on their role. By granting access to as much architectural knowledge as possible, reuse is stimulated and the chance of users correcting or improving knowledge grows. Classified information defies the wiki philosophy of collaborative editing, so if access to specific architectural knowledge needs to be restricted, consider storing it on other systems instead of on the wiki.

Please note that four of the five dos and don'ts listed above are not necessarily 'architecting-specific', and could well apply to other domains as well. The one that stands out, though, is the fact that the scope of the wiki should encompass both architectural knowledge and non-architectural knowledge. This permits the wiki to become an all-round knowledge management environment that supports the architecting process in a more mature way compared to the other tools.

10.5 Conclusions

In this chapter we saw that wikis have several characteristics which make them very suitable as knowledge sharing environment in the architecting process. They could be employed as all-round environments, offering strong support for collaboration, online communication, and consensus decision making. In an attempt to better answer research question RQ-II.7 (using a second iteration of tool development and experimentation at RFA), we found that the applicability of wikis as environment for sharing architectural knowledge is quite broad. Nevertheless, to guarantee the wiki's success, a proper strategy must be chosen for setting up and adoption of the wiki in an organization. To assist both researchers and practitioners in this task, we have established a number of dos and don'ts for wiki usage in the architecting process.

The most apparent limitation of the wiki we found is that codification of architectural knowledge is less straightforward than with using state-of-the-art tools that are explicitly constructed for this task, and requires additional support by means of plugins or templates. This drawback, however, is outweighed by the wiki's versatility to manage all sorts of non-architectural knowledge related to the architectural knowledge produced, as well as its support for integration with other tools, intuitive interface and low learning curve.

Wikis are quite suitable in creating a community of architects in which everybody knows from each other where certain expertise, or experience resides. This helps architects in various activities such as conducting reviews,writing reports, or communication with stakeholders. The enterprise wiki we experimented with offers various plugins which add various interesting features, such as modeling support, generation of statistics and overviews, and even text mining features in combination with database integration. We argue that these aspects can further support architects in their daily work by offering smart or pro-active support when needed, in a non-intrusive way (e.g., automatically identifying patterns or rules from raw wiki content). Semantic wikis seem promising in this respect, as argued by Lago (2009). In these systems knowledge retrieval is improved by enforcing stricter ways of knowledge representation.

11 What Architects Do and What They Need

In this chapter we present the results of a survey conducted in four IT organizations in the Netherlands, to which almost 279 practicing architects participated. This study's main goal is to unravel what architects really do and what kind of support they need for sharing architectural knowledge. Our survey validates a theory that was formed based on state-of-the-art literature as well as on insights gained during the four case studies conducted at RFA.

11.1 Introduction

Over the past few years, an increasing emphasis is put on the role of decision making in the software architecting process. Consequently, the role of the architect in this process is a frequently recurring topic of discussion, and more and more researchers and practitioners deliberate on what a proper set of duties, skills and knowledge of architects would be (Clements et al., 2007). In addition, researchers have proposed various tools to support the architect, who is characterized as an all-round knowledge worker. Such a knowledge worker, as these researchers argue, would definitely benefit from specific support for managing design decisions, modeling architectural solutions, or related activities that can be automated. With such support relevant architectural knowledge is easier to share, which leads to more effective knowledge exchange and reuse, and which prevents knowledge vaporization.

There are various approaches to support architects in sharing architectural knowledge. In practice, however, the adoption of these approaches is limited, perhaps due to a lack of alignment to the architecting process. We found that these approaches are often developed from a technological perspective alone (cf. Chapter 8). To keep from falling into this so-called ICT trap (Huysman and de Wit, 2004) we wish to find out what support for architectural knowledge sharing best fits the architecting process.

The only way to understand what would really benefit architects, is to unravel what architects really do and what kind of support they need during their main activities, which corresponds to answering research questions RQ-II.2 and RQ-II.3. For this purpose, we have constructed a theory on architecting, based on a review of state-of-the-art literature and experiences gained during the four case studies conducted at RFA. Our theoretical framework consists of a number of identified architecting activities and a number of support methods that assist architects in sharing architectural knowledge during these activities. Further, we hypothesize the existence of a number of 'patterns': certain methods for sharing architectural knowledge are more supportive than others to architects involved in a specific activity. The identification of such patterns helps in the development and maturation of methods and tools for sharing architectural knowledge.

In this chapter we report on the fifth and final study of our action research cycle that aimed to validate aforementioned theory. To validate our theory we have conducted large-scale survey research in four IT organizations in the Netherlands. In total 279 practicing architects were asked about their daily activities, and how important they consider the various types of support for sharing architectural knowledge. Our study consisted of a pilot study to pretest the theoretical framework, followed by a main study to answer our research questions and hypotheses.

The remainder of this chapter is organized as follows. In §11.2 we outline our theory on architecting activities and support methods for architectural knowledge sharing. In §11.3 we elaborate upon the research methodology of our study. §11.4 presents our data analysis to validate the theory. In §11.5 we reflect on the results and discuss their implications.

11.2 Theoretical framework

Based on experiences in earlier industrial case studies, combined with established literature on software architecture and knowledge management, we developed a theory to characterize what architects do and what they need for sharing architectural knowledge. In the next three subsections we outline our theory on what kind of activities architects are involved in (§11.2.1), what kind of support they need (§11.2.2), and what patterns between support and activities exist (§11.2.3).

11.2.1 Architecting activities

Our theory of architecting activities is depicted in Table 11.1 and consists of five categories: communication, decision making, quality assessment, documentation, and knowledge acquisition. For each activity we defined several subactivities that are most indicative to its activity, which was tested by our online survey. The remainder of this section highlights some of the related work our theory was built on.

According to Kruchten (2008), software architects should "make design choices, validate them, and capture them in various architecture related artifacts". Doing all these things involves a lot of consensus *decision making* in which architects balance between quality attributes, stakeholder concerns, and requirements, and in which they

Table 11.1: Architecting activities

A1: Communication
- I inform colleagues about the results of my work. - My colleagues keep me up-to-date on the results of their work. - I explain existing architectural principles to colleagues. - During discussions I share my knowledge to colleagues.
A2: Decision Making
- Before I take a decision I weigh the pros and cons of possible solutions. - While taking decisions I meet the wishes of the stakeholders. - I think about what impact my decisions have on the current architecture. - I study the reasoning behind taken design decisions.
A3: Quality Assessment
- I convince stakeholders in order to share the architectural vision. - I convince stakeholders about the value of my architectural solution. - Stakeholders approach me to discuss about architectural issues. - I notify stakeholders about actions that deviate from the architecture. - I check whether proposals from stakeholders are in line with the architecture. - I judge whether architectural proposals could continue or be executed.
A4: Documentation
- I create documents that describe architectural solutions. - I use (parts of) existing documents while creating new deliverables. - I use templates to store architectural knowledge. - I write vision documents to inform stakeholders. - I write progress reports about the architecture to stakeholders.
A5: Knowledge Acquisition
- I learn from my colleagues about architectural principles. - Colleagues learn from me about architectural principles. - I read (scientific/professional) literature on architecture. - I keep my knowledge up-to-date by searching relevant information on Intranet or Internet. - I expand my knowledge by visiting conferences and other events about architecture.

try to apply architectural styles and patterns.

A study by Clements et al. (2007) on the duty, knowledge, and skills of architects confirms the above view of the primary tasks of architects. They found that architects frequently interact with stakeholders, are involved in organization and business related issues, but primarily guide the architecting process: "We realized, from our many interactions with practicing architects, that their job is far more complex than just making technical decisions, although clearly that remains their most essential single duty".

In the decision making process architects *communicate* and share ideas with various stakeholders and collaborate in teams to find optimal solutions (cf. Chapter 8). To keep track of all knowledge being created or shared in this process, architects maintain a backlog (Hofmeister et al., 2007). In this backlog, an overview of decisions, constraints, concerns, open issues etc. are maintained by the architect. As input to this process, architects need a vast body of architectural knowledge. According to Clements et al. (2007) this body of knowledge includes basic computer science knowledge, design knowledge, and knowledge about organizational context and management. Architects share this knowledge not only by talking to colleagues in their team, but also by communicating with stakeholders outside the team.

That architecting is not only about technology is also illustrated by an architect competency list proposed by Bredemeyer and Malan (2004). These competencies are classified in the categories technology, consulting, strategy, organizational politics, and leadership. Bredemeyer's vision is further underlined by Eeles (2006b), who acknowledges that architects do not only do technical stuff ('they are a technical leader', 'they understand the development process') but that they also need a lot of soft skills: Architects 'have knowledge of the business domain', 'are good communicators', 'make decisions', 'are aware of organizational politics', and 'need to be effective negotiators'. Soft skills are also emphasized by van der Raadt et al. (2008), who studied how different stakeholders in the Enterprise Architecture domain perceive the enterprise architecture function. Apart from technical knowledge or skills, enterprise architecture involves to a large extent governance, communication, vision, and collaboration.

Another main architecting activity is to *document* architectural knowledge. This includes not only storing application-generic knowledge such as architectural principles, patterns or styles, but also codifying application-specific knowledge such as design decisions made and their rationale. Architectural knowledge may be stored in architectural descriptions or databases, modeled in reference architectures, or grouped in specific repositories.

During the 2006 and 2007 workshops on sharing and reusing architectural knowledge (SHARK) an explicit distinction was made between architecting as a product and as a process (Lago and Avgeriou, 2006; Avgeriou et al., 2007b). During these sessions a classification scheme for architecting as a process was constructed. This scheme adds two important architecting activities to the decision making, documenting and communication activities: learning and assessing. With respect to learning, Clements et al. (2007) list university or industrial courses, and certification programs as possible sources of training and education for architects. Apart from these specific learning activities, in their everyday work architects are also constantly busy with *knowledge acquisition* activities to update their knowledge. By having discussions with colleagues or customers, reading fora on the internet, or by visiting seminars, workshops or conferences they stay up-to-date and become acquainted with new trends, developments or best practices.

The last main architecting activity we define is *assessing the quality* of architectures. Assessing and reviewing architecture descriptions or related deliverables allow architects to control quality. They are ultimately responsible for the quality of the software systems. The design and evolution of an architecture allows them to reach this goal. As part of the quality control, architects also prescribe certain rules and regulations to stakeholders in the projects they work on. Stakeholders need to adhere to these rules, and architects may reject deviations from these rules by overruling specific decisions made.

11.2.2 Support methods

We also wanted to know the kind of support architects require with respect to sharing architectural knowledge. Effective support for sharing architectural knowledge is crucial, not only to assist individual architects, but also to improve the quality of architectural designs by leveraging all knowledge assets available in the organization. Table 11.2 shows our theory of support methods for sharing architectural knowledge (i.e., decision management, search efficiency, community building, intelligent advice, and knowledge management). The indicative sub-methods listed in this table served as items on our survey.

In the previous section we have seen that architects communicate, make decisions, assess quality, document and acquire knowledge, all of which are highly knowledge-intensive activities. In this subsection we outline out thoughts on the main areas support methods for sharing architectural knowledge should focus on.

A large part of the architectural knowledge produced is related to the design decisions, so that explicit support for *management of these decisions* is warranted. Recently, various researchers have proposed tools that help managing concepts like design decisions, rationale, and related architectural knowledge (Ali Babar and Gorton, 2007; Capilla et al., 2007; Jansen et al., 2007). These tools include specialized templates, overviews and storage facilities for architectural knowledge concepts. Such support for management of design decisions could help architects involved in decision making.

Table 11.2: Support methods

S1: Decision Management - An overview of the most important architectural decisions. - An overview of the relations between taken decisions. - Templates for codification of architectural decisions. - Insight into conflicts between architectural decisions. - An overview of changes through time of certain decisions. - A repository to store architectural decisions.
S2: Search Efficiency - Search methods for existing architectural guidelines. - Retrieving all documentation related to a specific architectural subject. - Search facilities for decisions within a specific project. - Retrieving relevant information within a project. - Overview of important events related to architecture.
S3: Community Building - A central system to hold discussions with stakeholders. - Notify colleagues about relevant documentation. - A central environment to collaborate with colleagues on arch. issues. - Retrieve information about the expertise of colleagues. - Acquire insight into which architectural projects colleagues worked on.
S4: Intelligent Advice - Concrete feedback during the writing process of arch. documentation. - Specific advices on which architectural decisions need to be taken. - Suggestions on how to use architectural guidelines in a specific project. - Being notified about the availability of new architectural publications. - Being sent an overview of the status of an architectural project.
S5: Knowledge Management - Automatic retrieval of architectural guidelines within projects. - Simple methods to annotate architectural knowledge concepts in documents with meta-data. - A automatically generated overview of open design issues in documents. - A system that tracks overlap among codified architectural guidelines. - Central maintenance of architectural guidelines and best practices.

While working on the ‘backlog’ of open issues and challenges they can organize their thought processes and manage the architectural knowledge they produce (Hofmeister et al., 2007).

In addition to assisting architects in producing architectural knowledge, support during the consumption of such knowledge is equally important. The need for a more balanced view on architectural knowledge sharing, in which support for both producing and consuming architectural knowledge is included, has been discussed before (Lago and Avgeriou, 2006). During the third case study at RFA it became clear that one of the main requirements architects have is support in retrieving the right architectural knowledge at any time (cf. Chapter 9). Support for *efficient searching* of architectural knowledge will stimulate reuse (reusable assets are better accessible) and learning among architects (they can find what they need more easily).

Due to the size and complexity of most software systems, it is often infeasible for one architect to be responsible for everything alone. This focus on teamwork is especially true in global software engineering environments. Consequently, the architect-role is often fulfilled by multiple collaborating architects. Liang et al. (2009) provide an introduction to how a collaborative architecting process would look like. To foster sharing of architectural knowledge between all these architects calls for support methods that focus on *community building*. We found during the fourth case study at RFA that 'islands' of practitioners can exist that work relatively independent from each other (cf. Chapter 10).

Architects often do not know what experienced colleagues from other teams work on, what their experience or expertise is, or what their interests are. We argue that support for community building improves this situation and increases the amount of architectural knowledge shared. With the advent of 'Web 2.0', this is much easier to implement than in the past. The need for systems that enable such support was explicated in Chapter 8 where we presented EAGLE. Likewise, as discussed in Chapter 10, systems such as Wikis can be employed to improve networking by letting stakeholders share ideas, discuss progress, or collaboratively produce architectural knowledge in a variety of formats.

Another main type of support in the architecting process relates to *intelligent advice* and support. Architects could benefit from intelligent support during almost all their core activities: a) during production activities such as writing an architectural description, b) directly after producing architectural knowledge (in terms of feedback on what they did), and c) during reviewing and evaluation activities. Intelligent or pro-active support helps architects to leverage and share the architectural knowledge assets available by taking over certain tasks. This might result in a software assistant who thinks together with the practitioners and suggests ideas, challenges decisions, etc. Implementations of pro-active and intelligent assistants are not often seen in practice yet. In a research setting, examples start to emerge. One example is provided by Garlan and Schmerl (2007), who designed a personal cognitive assistant called RADAR, which could help architects to accomplish their high-level goals, coordinating the use of multiple applications, automatically handling routine tasks, and, most importantly, adapting to the individual needs of a user over time. Another example is the Knowledge Architect tool suite that, among other things, assists architects in writing documentation, and that can perform quantitative architectural analysis (Liang et al., 2009).

The last type of support we envision is advanced *management of architectural*

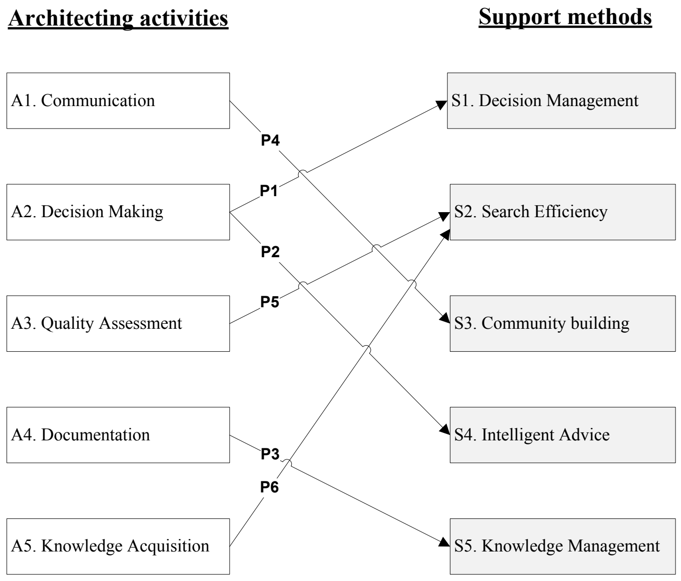


Figure 11.1: Hypothesized patterns of architectural knowledge sharing support

knowledge that includes refinement, enrichment and smart overviews of such knowledge. Architects could benefit from (semi)-automatic interpretation of stored knowledge to enrich it. Text mining services (e.g., (Fan et al., 2006)) could be employed to automatically sift through existing architectural knowledge stored (e.g., in a database) looking for new patterns, define best practices, or locate trends. Based on the findings, additional meta-data could be generated and presented to the architect. Consequently, searches executed could deliver more in-depth results that can then be shared, because part of the interpretation of the 'raw' architectural knowledge has already been done.

11.2.3 Architectural knowledge sharing patterns

Table 11.1 and Table 11.2 show the factors in our theory on architecting activities and support for architectural knowledge sharing, respectively. We hypothesized that specific support might be more appreciated during specific architecting activities. This means that certain 'patterns' exist that describe which type of support fits which type of architecting activity best.

We hypothesized that six of such patterns exist, as depicted in Fig. 11.1. For each pattern we formulated a hypothesis that was tested with our survey.

P1 Architects involved in taking architectural decisions (A2) benefit particularly from support for management of architectural decisions (S1). We argue that practitioners who are often involved in decision-making could benefit greatly from explicit support for working on a decision 'backlog', by having overviews of open issues, conflicting decisions, traces between requirements and solutions, etc.

P2 Architects involved in taking architectural decisions (A2) benefit particularly from support for intelligent advice (S4). We argue that during the thought processes and tradeoffs inherent to decision making, pro-active or Just-in-Time support might benefit architects greatly. This could include (automated) advice on which decisions to take in a specific situation, or tips about how to apply architectural patterns, styles or tactics.

P3 Architects involved in documenting architectural knowledge (A4) benefit particularly from support for maintenance and overview of architectural knowledge (S5). Architects who are often involved in codifying knowledge in artifacts such as architectural descriptions are directly helped by templates, models, frameworks etc. Also easy access to other codification methods such as central repositories for guidelines and repositories lower the threshold of producing something and eases the storing process.

P4 Architects involved in communication with colleagues or other stakeholders (A1) benefit particularly from support for community building (S3). We expect architects who are frequently involved in communication with colleagues or other stakeholders to be more enthusiastic about community building, sharing knowledge, and networking using for example social networks (e.g., wikis, blogs). After all, methods such as people directories, yellow pages, but also discussion facilities are more interesting for people that often use them and also lead to greater benefits than when there is less collaboration with others.

P5 Architects involved in quality assessments (A3) benefit particularly from support for efficient retrieval of (or searching for) architectural knowledge (S2). During reviews and assessments quickly retrieving information about the status of a project, the set of principles applied, or the key architectural decisions taken greatly benefit architects. This would also lead to higher-quality results during the evaluation, because it is based on more accurate and complete architectural knowledge.

P6 Architects involved in expanding their knowledge on architecture (A5) benefit particularly from support for efficient retrieval architectural knowledge (S2). Architects could learn more effectively (i.e., faster, more to-the-point) if they have the means to quickly retrieve the architectural knowledge they need at a specific point in time. Just-in-Time architectural knowledge allows them to quickly find new domain knowledge, get the status of a project, or read through the main deliverables of a project. To put it differently, by having good support for retrieving such knowledge, the practitioner can enrich and refine the knowledge himself by combining inputs from different sources, so that the knowledge internalization process is significantly improved.

11.3 Research Methodology

In total 279 Dutch architects from four IT organizations were included in this research. These organizations include:

1. RFA, the software development organization in which we conducted the four case studies presented in the previous chapters.
2. SCI, an international consultancy firm specialized in infrastructure and maintenance of software systems.
3. SOT, an international IT Service Organization for consulting, system integration and managed operations.
4. OBG, a central Dutch governmental organization for the governance of ICT architectures in the public domain.

Our study is conducted according to a structured design, which is elaborated upon in the remainder of this chapter. Our research methodology includes all important elements required to properly design, administer and analyze a survey (Kitchenham and Pfleeger, 2001-2002). Our survey consists of a pilot study, followed by the main study, both of which are further highlighted in the following sections. For a complete discussion we refer to the technical report written about this study (Farenhorst et al., 2009).

11.3.1 Pilot study

The theoretical framework presented in §11.2 is not something we built up overnight. An earlier theory consisting of six categories of architecting activities and six different support methods was presented in (Farenhorst and van Vliet, 2009). The main goal of the pilot study was to see whether this earlier theory was sufficiently strong for usage in the main study, and to improve – or refactor – it where necessary. To this end, we made constructs for all activities and support methods. Inspired by relevant literature and gained insight during the four case studies conducted at RFA, we formulated 10 to 15 questions (i.e., 'items') for each of these constructs. After iteratively scrutinizing these items on clarity and redundancy, some were removed, leaving each construct between 5 and 10 items.

Because 10 items per construct would make the questionnaire of the main study too long, we decided to let a small percentage of the total sample ($N = 8$) pretest all items in a pilot questionnaire. To test the internal consistency of the items we performed reliability analysis on the item scores of the pilot questionnaire, and selected the ones with the strongest internal consistency (i.e., highest Cronbach's alpha).

To make sure that items did not indicate other activities or support methods as well, we performed factor analysis. This way we could sharpen our concepts. To give an example, in our earlier theory we made a distinction between communication within a team (with colleagues) and communication outside the team (with other stakeholders). However, the factor analysis of the pilot questionnaire indicated that this distinction is not made by architects; the items of both constructs all mapped on one factor. Consequently, we merged these two constructs and labeled the new one 'Communication'. This is why our final theory of architecting activities (Table 11.1) contained five categories and not six. In a similar fashion, we refactored our theory of support methods. Coincidentally, again we had to change the theory by going from six to five constructs. For each of these constructs the five or six most explanatory items were selected, as depicted in Table 11.2.

11.3.2 Main study

In our main study we tested our theoretical framework as presented in §11.2. To properly answer our main research question, which involved unraveling what architects do and what support they need, the questionnaire of our main study consisted of three parts:

- **Part 1: Architecting activities.** Each of the five architecting activities was treated as a separate scale, consisting of the (four to six) items as depicted in Table 11.1. Each item described a sub-activity (e.g., *'I inform colleagues about the results of my work'*) and respondents replied to what extent that activity related to their daily work by scoring a 6-point Likert-type rating scale (1 = totally disagree, 6 = totally agree).

- **Part 2: Support methods.** Each of the five support methods for sharing architectural knowledge was treated as a separate scale, consisting of the (five to six) items representing sub-methods, as depicted in Table 11.2. Respondents indicated on 6-point Likert-type rating scales to what degree these support methods were helpful in their daily work. A sample item was *'In my daily work, I benefit from an overview of the most important architectural decisions'*
- **Part 3: Prioritization of support methods.** The respondents prioritized the five types of support methods in order of importance for *one particular* activity alone. They ranked the five methods between 1 and 5, where a score of '1' corresponded to the highest ranked method and '5' corresponded to the lowest ranked method for that architecting activity. We made sure that no score could be given twice and that no missing values were allowed.

In addition to the questions of the three parts, our (anonymous) questionnaire contained some additional questions including the respondent's role (which 'type' of architect), and his or her experience (number of work years). This allowed us to analyze the impact of these additional factors on respectively the architecting activities they are involved in (Part 1), the support methods they like (Part 2) and their prioritization of those support methods (Part 3).

To prevent bias and fatigue effects we also randomized the order of the question screens of Part 1 and Part 2 as much as possible. We have constructed the following experimental design to accommodate for these requirements:

- We split up the sample in 5 equally large homogeneous groups of architects. Each of these groups got a different questionnaire. Of these 5 questionnaires Part 1 and Part 2 were identical, except for the ordering of the screens. The difference was in Part 3: each questionnaire only contained *'one'* prioritization question, i.e., selected one architecting activity and asked the architects to prioritize the support methods for this activity. This prevented too much repetition (and possible contaminated results), which would have occurred if the same respondent had to rank the same methods five times, once for each activity (Hoorn et al., 2006).
- To verify whether the ordering of the screens in Part 1 and Part 2 did not lead to a bias in how the support methods in Part 3 were ranked, we made two versions for each of the 5 groups mentioned above. Each of these versions had a distinct ordering of the question screens, as is depicted in Table 11.3. So in the end the respondents were divided equally over 10 questionnaires (Q1-Q10), but as soon as the test for bias had turned negative,five bigger groups could be formed again (by combining Q1+Q6, Q2+Q7, etc.).

	Part 1: Architecting activities					Part 2: Support for AK sharing					Part 3: Prioritize support
Quest.	**Act1**	**Act2**	**Act3**	**Act4**	**Act5**	**Sub1**	**Sub2**	**Sub3**	**Sub4**	**Sub5**	**Ranking question**
Q1	1	2	3	4	5	1	2	3	4	5	Act1
Q2	1	2	3	4	5	1	2	3	4	5	Act2
Q3	1	2	3	4	5	1	2	3	4	5	Act3
Q4	1	2	3	4	5	1	2	3	4	5	Act4
Q5	1	2	3	4	5	1	2	3	4	5	Act5
Q6	5	4	3	2	1	5	4	3	2	1	Act1
Q7	5	4	3	2	1	5	4	3	2	1	Act2
Q8	5	4	3	2	1	5	4	3	2	1	Act3
Q9	5	4	3	2	1	5	4	3	2	1	Act4
Q10	5	4	3	2	1	5	4	3	2	1	Act5

Table 11.3: Experimental design: Different versions of the same questionnaire

- Finally, all questions in the screens of Part 1 and Part 2 were randomized, which means that they were posed to each respondent in a different order.

- All respondents start from the same introduction screen in the web survey tool Examine [1], but as soon as they start with the actual questionnaire, the 'experimental design' feature of Examine redirects the respondent to one of the ten versions (Q1-Q10). The advantage of this approach is that there is only one link to our (anonymous) questionnaire, which was thus easy to place in the invitation emails send out in the four participating organizations.

11.4 Survey Analysis

Our main study was conducted in February 2009 at the four IT organizations in parallel. In total 271 architects from these organizations were included in the main study, 156 of whom responded (see Table 11.4). From this group, 13 were discarded in the analysis because their records were completely empty. One respondent was removed from the analysis because s/he produced scores of '1' alone. This left us with 142 respondents for the scale and factor analysis, which accounts for a response rate of 52.4%. Remaining missings were treated as subject-missing.

From the 142 respondents, 48 labeled themselves as Enterprise architects, 63 as IT or software architect, and the remaining ones (31) were either information analysts, infrastructure architects, or application designers. Their average relevant working expe-

[1] http://examine.vu.nl/

Table 11.4: Response rate

Organization	**Sample**	**Responses**	**Response rate**
RFA	56	42	75,0%
SCI	39	20	51,3%
SOT	171	76	44,4%
OBG	5	4	80%
Total	**271**	**142**	**52,4%**

rience was 6 years; 26 architects indicated less than 3 years of architecting experience, 89 architects 5 years or more.

11.4.1 Scale analysis

To assess whether our data was internally consistent and that the items were really indicative for the constructs of our descriptive framework, we started with conducting scale analysis. To this end, we assessed the psychometric quality of the 5 Activity scales (Communication, Decision Making, Quality Assessment, Documentation, Knowledge acquisition) and the 5 Support scales (Decision Management, Search Efficiency, Community Building, Intelligent Advice, Knowledge Management). Scales had between 4 and 6 items each. We tested whether items correlated sufficiently with their own scale by means of Corrected Item-Total Correlations and regular Cronbachs alpha (indicating reliability).

Item selection was a trade-off among several criteria. We wanted to establish as many items on a scale as possible with a minimum of 2, provided that Cronbachs alpha for a scale was $\geq .60$ and Corrected Item-Total Correlations $\geq .20$. In addition, the degree to which items did not correlate with other scales was tested with factor analysis, one time for the Activities scales and one time for the Support scales (PCA, 25 iterations Varimax rotation with Kaiser Normalization, number of factors to be extracted: 5). The rotated component matrix showed that all five factors could be retrieved in the data, except for a limited number of items. We removed 8 items that correlated more strongly with another scale than with their own scale. Three more items were removed that correlated above .40 with other scales irrespective of high correlations with their own scale. In Table 11.5 the resulting set of items is depicted.

The thus revised scales were submitted to reliability analysis again. Scale length was reduced to 3, 4, or 5 items. Regular Cronbach's alphas were between .72 and .90, which is reasonable to good. The scale Knowledge Acquisition had an alpha of .61, which is suspect but still acceptable. In Table 11.6 the reliability analysis with

Table 11.5: Selected items after reliability analysis

A1: Communication
- I inform colleagues about the results of my work.
- My colleagues keep me up-to-date on the results of their work.
- I explain existing architectural principles to colleagues.
- During discussions I share my knowledge to colleagues.

A2: Decision Making
- Before I take a decision I weigh the pros and cons of possible solutions.
- I think about what impact my decisions have on the current architecture.
- I study the reasoning behind taken design decisions.

A3: Quality Assessment
- I convince stakeholders about the value of my architectural solution.
- Stakeholders approach me to discuss about architectural issues.
- I notify stakeholders about actions that deviate from the architecture.
- I check whether proposals from stakeholders are in line with the architecture.
- I judge whether architectural proposals could continue or be executed.

A4: Documentation
- I use (parts of) existing documents while creating new deliverables.
- I use templates to store architectural knowledge.
- I write vision documents to inform stakeholders.
- I write progress reports about the architecture to stakeholders.

A5: Knowledge Acquisition
- I read (scientific/professional) literature on architecture.
- I keep my knowledge up-to-date by searching relevant information on Intranet or Internet.
- I expand my knowledge by visiting conferences and other events about architecture.

S1: Decision Management
- An overview of the most important architectural decisions.
- An overview of the relations between taken decisions.
- Templates for codification of architectural decisions.
- Insight into conflicts between architectural decisions.
- An overview of changes through time of certain decisions.

S2: Search Efficiency
- Search methods for existing architectural guidelines.
- Retrieving all documentation related to a specific architectural subject.
- Search facilities for decisions within a specific project.
- Retrieving relevant information within a project.

S3: Community Building
- A central system to hold discussions with stakeholders.
- Notify colleagues about relevant documentation.
- A central environment to collaborate with colleagues on arch. issues.
- Retrieve information about the expertise of colleagues.

S4: Intelligent Advice
- Specific advices on which architectural decisions need to be taken.
- Suggestions on how to use architectural guidelines in a specific project.
- Being sent an overview of the status of an architectural project.

S5: Knowledge Management
- Automatic retrieval of architectural guidelines within projects.
- Simple methods to annotate architectural knowledge concepts in documents with meta-data.
- A automatically generated overview of open design issues in documents.
- A system that tracks overlap among codified architectural guidelines.

Table 11.6: Reliability analysis with shortened scales after factor analysis

Scale	*Cronbach's α*	*# items*	*n*
Communication	.80	4	132
Decision making	.85	3	131
Quality assessment	.80	5	128
Documentation	.72	4	131
Knowledge acquisition	.61	3	136
Decision management	.80	5	123
Search efficiency	.83	4	123
Community building	.78	4	123
Intelligent advice	.74	3	123
Knowledge management	.90	5	122

shortened scales after factor analysis is depicted.

11.4.2 What architects do

To find out what practicing architects really do we studied the data of Part 1 of our questionnaire. We ran a GLM-Repeated Measures for the 5-leveled within-subjects Activities factor (Communication, Decision Making, Quality Assessment, Documentation, Knowledge Acquisition) on the mean agreement scores with number of work years as covariate. According to the multivariate tests, the main effect of Activities was significant with a considerable effect size (Pillai's Trace = .51, $F_{(4,121)} = 31.4$, $p = .000$, $\eta_p^2 = .51$). See (Farenhorst et al., 2009) for more details.

As depicted in the upper part of Table 11.7, the activity Decision Making received the highest agreement scores ($\mu = 5.21, \sigma = .65$) and Documentation the lowest ($\mu = 3.98, \sigma = .89$). Analysis of variance showed that these results were not significantly different between the four participating organizations, nor did the type of architect impact this outcome. We did, however, find a significant correlation between the amount of work experience architects have and the activities they are involved in. Particularly, agreement to Quality Assessment and Documentation increased significantly with the number of Work Years (see (Farenhorst et al., 2009)).

Paired-samples t-tests showed that all differences among the activities were significant ($t > 2.7$, $p < .009$), except for the contrast between Communication and Knowledge Acquisition ($t_{125} = -.38$, $p >> .05$). However, these effects were mod-

Table 11.7: Descriptive statistics for activities and support methods

	Mean	Std. Deviation	Correlation with Work Years	N
Work years (experience in architecture projects)	*6.22*	*4.110*		*142*
Architecting activities				
Communication	4.6155	.67423	.20*	132
Decision Making	5.2087	.65285	.18*	131
Quality Assessment	4.4484	.82204	.37**	128
Documentation	3.9847	.89052	.23**	131
Knowledge Acquisition	4.6152	.74135	.15	136
Support methods				
Decision Management	4.8374	.58188	.15	123
Search Efficiency	4.8943	.60902	.09	123
Community Building	4.5325	.74586	.10	123
Intelligent Advice	4.5230	.64030	.10	123
Knowledge Management	4.3033	.90077	.15	122

ulated by the number of work years that respondents worked on architectural projects ($\mu = 6.22$ years, $\sigma = 4.11$). The interaction between Activities and Work Years was significant albeit with small effect size (Pillai's Trace = .09, $F_{(4,121)} = 2.9$,$p = .026$, $\eta_p^2 = .09$). Further inquiry with Pearson correlations into the relationship between this covariate and the different Activities showed that particularly agreement to Quality Assessment and Documentation increased significantly with the number of Work Years. This was also the case for Communication and Decision Making but not for Knowledge Acquisition.

Our survey results thus indicate that interestingly enough, from the five architecting activities, architects are mostly busy with taking architectural decisions and least with documenting the results. This runs the risk of leading to substantial architectural knowledge vaporization (Bosch, 2004) and substantially decreases the chances for effectively reusing architectural knowledge. The effect of work experience indicates that more experienced architects seem more often involved in auditing activities, and – maybe related to this – spend more energy on documenting their results. Maybe the advantages of retrieving or reusing codified knowledge has proven itself to these experienced architects over the years.

11.4.3 What architects need

To examine what kind of support methods architects desire, we studied the data of Part 2 of our questionnaire. We ran a GLM-Repeated Measures for the 5-leveled within-subjects Support factor (Decision Management, Search Efficiency, Community Building, Intelligent Advice, and Knowledge Management) on the mean agreement scores with Work Years as covariate. The main effect of Support was significant with medium to low effect size (Pillai's Trace = .21, $F_{(4,117)} = 7.7$, $p = .000$, $\eta_p^2 = .21$). Again, see (Farenhorst et al., 2009) for more details.

As depicted in the lower part of Table 11.7 Search Efficiency raised the highest agreement scores ($\mu = 4.89, \sigma = .61$) and Knowledge Management the lowest ($\mu = 4.30, \sigma = .90$). Again, analysis of variance showed no significant differences between types of architects or between architects from different organizations. The interaction between Support and Work Years also was not significant ($F < 1$).

To further explore the main effect of Support, paired-samples t-tests showed that all differences among the kinds of Support were significant ($t > 2.7, p < .008$), except for the contrast between Decision Management and Search Efficiency ($t_{122} = -.38$, $p >> .05$) and between Community Building and Intelligent Advice ($t_{122} = .15$, $p >> .05$). This means that Search Efficiency and Decision Management were equally agreed upon as the kind of support most needed, irrespective of Work Years.

We find it rather contradictory that although architects do not document much, these results indicate that they do wish for efficient support to retrieve useful – previously stored – knowledge. It seems that architects really wish to use codified knowledge, but refrain from codifying it in the first place. A proper balance between producing and consuming architectural knowledge, as deemed important in (Lago and Avgeriou, 2006), is lacking.

Another surprise related to architects' opinion about the various support methods is indicated by the low score for Knowledge Management. Considering that architects make a lot of decisions but are not that eager to properly document them, we expected that they would be more enthusiastic about automated support for these activities. One would say that easy annotation of architectural knowledge in documents would make the codification tasks a little less cumbersome. Yet, architects do not fancy such pro-active, or automated support. Perhaps they wish to remain in the driver's seat, and do not trust a system that manipulates and processes architectural knowledge independently. This is in line with an earlier study that indicated that one of the desired properties for architectural knowledge sharing support we identified in Chapter 8.

11.4.4 Preferences in support methods

We examined which support methods architects prefer by conducting two separate analyses: a) we tested the patterns hypothesized in Fig. 11.1 by trying to retrieve them in the data of Part 1 and Part 2 of our questionnaire, and b) we examined the ranking data of Part 3.

For our first analysis we formulated six regression models in which the level of agreement to activities should predict the level of agreement to the kind of support needed (see Fig. 11.1): $A1 \rightarrow S3$, $A2 \rightarrow S1$, $A2 \rightarrow S4$, $A3 \rightarrow S2$, $A4 \rightarrow S5$, $A3\&A5 \rightarrow S2$. Linear regression analysis confirmed the hypotheses formulated for patterns P1, P2, and P5. In each case the corresponding activity explained to a considerable extent the variance in agreement to the specific support methods:

P1 Decision Making explained 25% of the variance in Decision Management (P1). This means that architects who make decisions are particularly in favor of decision management. This is in line with our assumption that backlog management and effective management of design decisions makes it easier for architects to have a proper overview of the solution space, reusable assets, potential conflicts, stakeholder demands, etc.

P2 Decision Making explained 19% of the variance in Intelligent Advice (P2). This means that intelligent advice such as specific advice on which design decision to take, or suggestions about which architectural guidelines or styles to use, is particularly helpful for architects who are busy taking design decisions.

P5 Quality Assessment explained 34% of the variance in Search Efficiency (P5). Retrieving architectural knowledge is important for architects, particularly when they are conducting audits or other quality evaluations. Apparently, to decide about the quality of an architecture it is important to quickly retrieve the standards, rules, etc. that need to be adhered to.

The other three patterns were rejected, because in these cases the activities did not significantly explain the variance in agreement to the corresponding support methods. Apparently, advanced Knowledge Management support is not particularly important to architects busy with Documentation (P3), Community Building is not particularly interesting to architects active in Communication (P4), and Search Efficiency is not particularly appreciated by architects busy with Knowledge Acquisition (P6).

For our second analysis we looked at the data of Part 3 of the survey, where the respondents ranked the five support methods. They did so for one activity alone, so that a 5-leveled between-subjects factor of Activity Ranking Condition could be created (Communication Condition, Decision Making Condition, Quality Assessment Condition, Documentation Condition, Knowledge Acquisition Condition). We ran a GLM-Repeated Measures for the between-subjects factor Activities Ranking Condition and the 5-leveled within-subjects factor of Ranked Support (Ranked Decision Management, Ranked Search Efficiency, Ranked Community Building, Ranked Intelligent Advice, and Ranked Knowledge Management) on the mean rank numbers with Work Years as covariate (see (Farenhorst et al., 2009) for more details).

However, the main effect of Work Years was not significant ($F < 1$) and the interaction between Ranked Support and Work Years also was not significant ($F < 1$). Therefore, we reran the analysis but this time excluding Work Years as covariate.

The multivariate tests showed that the interaction of Ranked Support by Activity Ranking Condition was not significant ($F < 1$) and that the main effect of Activity Ranking Condition on the mean rank numbers also was not significant ($F < 1$). However, the main effect of Ranked Support did reach significance with a small effect size (Pillai's Trace = $.12, F_{(4,112)} = 3.8, p = .006, \eta_p^2 = .12$). Keeping in mind that the lower rank order number indicates the higher priority, Table 11.8 shows that according to the architects, Ranked Decision Management ($\mu = 2.63, \sigma = 1.37$) had higher priority than Ranked Intelligent Advice had lowest priority ($\mu = 3.33, \sigma = 1.42$).

Paired-samples t-tests showed that five out of 10 comparisons among the Ranked Support options were significant, which means that the need for support is a sliding scale of priorities (see Table 11.9).

In combining and interpreting Table 11.8 and Table 11.9, we get to a hierarchy of support methods that are more *or* less needed for architecting activities. The lower rank numbers indicate the higher priority. According to these architects, Decision Management had highest priority and Intelligent Advice lowest, irrespective of the activities they are involved in.

Priority of need for support throughout all architecture activities:

- **Most needed:** Decision Management ($\mu = 2.63$), Search Efficiency ($\mu = 2.8$)
- **Needed:** Knowledge Management ($\mu = 2.91$), Community Building ($\mu = 3.28$)
- **Least needed:** Intelligent Advice ($\mu = 3.33$)

Table 11.8: Descriptive statistics for the effects of activities on ranked support

	Activity for which support options were ranked	Mean	Std. Deviation	N
Ranked Decision Management	Communication	2,78	1,166	23
	Decision Making	2,54	1,474	24
	Quality Assessment	2,75	1,351	28
	Documentation	2,76	1,338	21
	Knowledge Acquisition	2,29	1,517	24
	Total	**2,63**	**1,366**	**120**
Ranked Search Efficiency	Communication	2,91	1,379	23
	Decision Making	2,88	1,329	24
	Quality Assessment	2,96	1,232	28
	Documentation	2,57	1,287	21
	Knowledge Acquisition	2,92	1,248	24
	Total	**2,86**	**1,279**	**120**
Ranked Community Building	Communication	3,39	1,559	23
	Decision Making	2,88	1,541	24
	Quality Assessment	3,04	1,347	28
	Documentation	3,90	1,480	21
	Knowledge Acquisition	3,33	1,659	24
	Total	**3,28**	**1,529**	**120**
Ranked Intelligent Advice	Communication	3,30	1,579	23
	Decision Making	3,38	1,173	24
	Quality Assessment	3,50	1,622	28
	Documentation	3,05	1,359	21
	Knowledge Acquisition	3,33	1,373	24
	Total	**3,33**	**1,421**	**120**
Ranked Knowledge Management	Communication	2,61	1,340	23
	Decision Making	3,33	1,494	24
	Quality Assessment	2,75	1,481	28
	Documentation	2,71	1,347	21
	Knowledge Acquisition	3,13	1,076	24
	Total	**2,91**	**1,366**	**120**

Table 11.9: Paired-samples t-tests for comparisons between priorities of support options

	t	*p*
Ranked Decision Management - Ranked Search Efficiency	-1.2	.221
Ranked Decision Management - Ranked Community Building	-3.0	.003**
Ranked Decision Management - Ranked Intelligent Advice	-3.6	.000**
Ranked Decision Management - Ranked Knowledge Management	-1.5	.146
Ranked Search Efficiency - Ranked Community Building	-2.1	.038*
Ranked Search Efficiency - Ranked Intelligent Advice	-2.3	.023*
Ranked Search Efficiency - Ranked Knowledge Management	-.28	.777
Ranked Community Building - Ranked Intelligent Advice	-.20	.841
Ranked Community Building - Ranked Knowledge Management	1.7	.085
Ranked Intelligent Advice - Ranked Knowledge Management	2.0	.048*

11.5 Discussion

The survey results offer good food for thoughts. Although we have found some plausible explanations for our findings, we were quite interested in the opinion of practicing architects. Therefore, we asked the respondents of our survey to provide feedback on the results. We sent them an email that summarized the main results that were presented in the previous section. This email triggered several responses within days, from which we could distill a number of interesting quotes. Some of these quotes are included in the discussion below.

With respect to what architects do, the most striking result is that architects seem to make lots of architectural decisions, but – especially the less experienced ones – neglect documenting those decisions and their rationale. Most obvious reasons for this is a lack of time or interest. Often the benefits of documenting design rationale are only visible in the longer term; in the short term the most important thing is to meet the deadline, and move on to the next project. The lack of interest in long-term knowledge reuse is in line with observations from Harrison et al. (2007), who found that architects lack a real motivation to document and maintain architectural knowledge.

The lack of a defined, visible process for architecting activities could also partly explain why architects omit to document much. If the role, responsibilities, and extent of the authority of architects would be better defined, these architects can be made more accountable by the organization for their actions. Experience in several software projects shows that a well-defined, visible process (e.g, by using a charter) stimulates architects to document more knowledge (Kruchten, 1999).

Another reason for not documenting architectural knowledge is that architects already possess this knowledge in their minds. There is no direct need for making this tacit knowledge explicit. This phenomenon is acknowledged by Kruchten et al. (2009), who also mention that as a consequence, architects cannot revisit or communicate these decisions. Bosch (2004) also acknowledges this problem, which he called knowledge vaporization. This process is to be prevented at all costs, because if the reasoning behind an architectural solution is lost or 'forgotten', it may cause design erosion.

Two more reasons for not documenting were provided by architects who responded to our feedback request:

> *"Whether or not to document is a tough decision. Doing so could make yourself redundant in case questions are asked later on, and that is what worries architects most."*

> *"Many architects have a technical background and were programmers, designers or system maintainers before they became an architect. For these people, documenting has always been a pain, and this is a mindset they still have."*

Our results are in line with a study on the architect's mindset by Clerc et al. (2007a). They conducted a survey among architects in the Netherlands and found that the prevalent mindset of architects is focused on 'to create and communicate' rather than 'to review and maintain' an architecture. In our study decision making and communication scored highest and quality assessment and documentation scored lowest, which further underlines the emphasis put on creating short-term solutions, rather than maintaining a body of knowledge that is useful for the longer term as well.

When looking at the prioritization of support methods we were not surprised to see that managing architectural design decisions was found most important. This is in line with an increasing focus on design decisions in the software architecture research community, as reflected in the formulation of a decision view on software architecture practice (Kruchten et al., 2009) and the adoption of central concepts as 'decision' and 'rationale' to the upcoming ISO/IEC 42010 standard for software architecture description (ISO/IEC WD3 42010).

The interesting thing, however, is that our results show that experienced architects differ slightly in their mindset; experienced architects do more on quality assessments and documentation. Perhaps they value these activities higher because they have encountered more situations in which sufficient documentation proved to be useful. Another explanation could be that writing architectural documentation is easier if you are more experienced (e.g., formulating rationale for decisions, or drawing effective architectural views). Likewise, conducting architectural audits is something that is only possible if you have sufficient experience in the field yourself. This makes it easier to spot inconsistencies, conflicts, or missing qualities of the system.

As for the other support methods, we were surprised that ‘knowledge management’ support received the lowest scores (see Table 11.7) and that ‘intelligent advice’ was ranked lowest by the architects (see Table 11.8). These two categories both assist architects in their routine knowledge processing tasks by automatically retrieving, and manipulating information, offering suggestions, etc. Particularly because architects appear to have little time or interest in documenting architectural knowledge, we had expected that all pro-active support would be more appreciated, and make certain basic tasks less cumbersome.

We suspect architects are a bit frightened by the idea that smart automated tooling would take over their role, although they acknowledge the positive effects (pattern P2 was significant). Architects want to sit in the driver’s seat and have a tight control over all processes. That also explains why support methods that do not endanger this role (e.g., search efficiency and decision management) are clearly ranked higher. Another reason for the relative low rank of intelligent support could be the lack of effective implementations of such support. We heard from several architects that most intelligent tools and methods are still rather immature and add little to their daily work, but that more versatile and mature implementations would be welcomed:

> *I find most specialized tools to have too little functionality that supports my work as a software architect. As a result, I often resort to more generic tools such as MS Visio, spreadsheets, etc.”*

The fact that architects want to have control over their actions is further underlined by the fact that in the prioritization of support methods, community building is ranked fourth (out of five). We had expected the architects to be a bit more ‘team-players’, or at least more in favor of creating a ‘community of architects’ in which experts effectively exchange knowledge, discuss experiences, and build up a collective memory. Our study gives us the impression that practicing architects are rather ‘lonesome’. They are not the community builders we had expected them to be. Instead they act in splendid isolation. Several architects confirmed this, two of whom explained it as follows:

> *“I recognize myself in the ‘lonesome architect’ characterization and find it actually quite logical. An architect is often busy working in the early stages of the software life cycle and of projects. Colleagues are busy helping to solve problems of other customers and are thus not around. Furthermore, there is often no guidance for the work you do as an architect because the systems you help design do not exist yet.”*

> *"An architect usually considers his own judgment as most precise and valuable. Referring to others (humans or tools) is often only a strategic move."*

Coming back to our main research question, our study provides valuable insight into what architects do and what support they need for sharing architectural knowledge. Architects are individual experts who consume substantial amounts of architectural knowledge but care less for actively sharing such knowledge. We conjecture that the best way to motivate these lonesome architects in sharing architectural knowledge involves non-intrusive support that offers various mechanisms to easily store, communicate or manipulate knowledge in the architecting process. Based on the results of our survey, we postulate that such support should have two fundamental characteristics:

1. **It respects the architect's autonomy.** We learned that architects like to control the processes they are involved in. Methods that support architectural knowledge sharing should respect this. Automated support that 'thinks' for the architect, or tools that prescribe architects what to do are not going to work. What does work, however, are more person-centric descriptive tools that assist knowledge workers 'on the fly' during daily routines (Cormican and Dooley, 2007). Although in this survey architects ranked intelligent support as least needed, feedback obtained suggests that such support could still be valuable if implementations would be sufficiently versatile and mature.

2. **It stimulates both production and consumption of architectural knowledge.** We saw that support for decision management and search efficiency were most needed according to practicing architects. Search efficiency support assists the autonomous, or lonesome, architect in retrieving relevant architectural knowledge when needed, and is thus oriented towards consumption of architectural knowledge. Decision management helps architects to manage their thought processes when taking (new) design decisions, solving conflicts, or codifying design rationale. This kind of support thus focuses mostly on the creation – or production – of architectural knowledge. Based on our survey results, we argue that mature support for sharing architectural knowledge should assist architects in both producing and consuming such knowledge.

   A promising development in software architecture is the emergence of integrated platforms for sharing architectural knowledge that implement a range of 'use cases' related to production and consumption of knowledge. A review of such platforms is provided by Liang and Avgeriou (2009). A main characteristic of some of these platforms is that they follow a hybrid strategy for architectural knowledge sharing. As Lago et al. (2008) argue, such a strategy “support[s] the knowledge producers in efficiently documenting the knowledge and the consumers in using it”, which would resolve the imbalance of producing and consuming knowledge we observed during our study.

11.5.1 Threats to validity

We list possible limitations to our study by discussing the internal validity, construct validity and external validity following Kitchenham et al. (2002) and Perry et al. (2000).

Our questionnaire design and pilot study ensured that our questions were unambiguous, to-the-point and our hypotheses well-focused on the research question we wanted to answer. This made analysis of the survey data relatively easy, and increased the chance for meaningful and significant results. With respect to *internal validity*, our questionnaire design leaves little room for selection bias or confounding variables.

To conform to *construct validity* we constructed our theory (cf. Fig. 11.1) as one of the first steps of our survey design, based on which the items of the questionnaire were phrased. During both the pilot study and main study we conducted scale analysis and factor analysis to assess whether our theorized concepts could be unambiguously retrieved in the data. In the pilot study this helped us to refactor our original framework (as presented in (Farenhorst and van Vliet, 2009)) into the framework presented in this paper. Scale and factor analysis during the main study indicated that indeed five factors of architecting activities and support methods could be identified, which indicates good construct validity.

With respect to *external validity* we deem our results fairly generalizable. Our sample was relatively large, and the fact that four different organizations participated decreased the chance for organizational bias significantly (even though three of the four organizations were rather similar in that their core business is IT consultancy). To obtain a fair reflection of a typical architecting process, many types of architects were included in our study, including software, IT, solution, enterprise and infrastructure architects. However, even though the participating companies were international, our study only focused on practicing architects working in the Netherlands. Our study thus does not deal with possible cultural or educational factors in which Dutch architects differ from their colleagues.

11.6 Conclusions

Architects make lots of architectural decisions, but neglect to document them. Producing and subsequently sharing of architectural knowledge is clearly not one of their most popular activities. For this reason, one would expect that supporting architects in this latter task is something they appreciate, but the opposite is true. Architects rather stay in control themselves and still need to be convinced of the value of current automated or intelligent support. When it comes to consumption of architectural knowledge, however, architects indicate that support for effective retrieval of (stored) architectural knowledge is on the top of their wish list. This imbalance in production and consumption shows that with respect to sharing architectural knowledge architects are not the 'community builders' many expect them to be. Instead, they are rather lonesome decision makers who act in splendid isolation.

Our survey research helped unraveling what architects really do and what they need with respect to sharing architectural knowledge, and thus provided answers to research questions RQ-II.2 and RQ-II.3. The answers to these questions act as call for awareness to both researchers and practitioners. We found that effective support for sharing architectural knowledge should acknowledge 'the nature of the beast'. We therefore plead for descriptive, non-intrusive support that respects the architect's autonomy. Moreover, to be effective, this support should stimulate architects in both producing and consuming architectural knowledge. Integrated, 'all-round', tools environments that implement a variety of architectural knowledge sharing use cases, seem promising because they could provide a solid balance between production and consumption of knowledge in the architecting process.

12 Conclusions

In this part we have studied how to support architects in sharing architectural knowledge. Using a typical action research cycle we have conducted four case studies at RFA followed by a fifth study that validated earlier findings. During these five studies we learned what organizational challenges exist to sharing knowledge in the architecting process and what desired properties of tool support are. Experimentation with our own tool environments, plus the identification of what architects do and what they need, further enhanced our understanding of how to effectively support architects in sharing architectural knowledge. In §12.1 we elaborate upon our contributions by revisiting and answering the research questions posed in Chapter 6. In §12.2 we discuss several topics that need further investigation.

12.1 Contributions

Software architecting is a highly knowledge-intensive process. Many different stakeholders are involved in this process. Due to the increase in size and complexity of software systems, architecting means collaborating. Hence, it is often the case that there is no one single all-knowing architect; instead the architect role is fulfilled by more than one person. In order to take well-founded decisions, all involved stakeholders need to obtain relevant architectural knowledge at the right place, at the right time.

Among researchers and practitioners there is a broad consensus on the importance of sharing architectural knowledge, in particular for reusing best practices, obtaining a more transparent decision making process, enabling efficient collaboration between stakeholders, and maintaining and increasing the know-how of architects.

It is standard practice nowadays for a software development organization to have databases and repositories at hand to store architectural documentation, standards, guidelines, patterns and other architectural knowledge; intranet sites harbor all kinds of related information; templates exist to model and collect specific knowledge entities such as design decisions in predefined formats. Still, storing, distributing and reusing knowledge happens on a rather *ad hoc* basis, which calls for more systematic approaches to effectively sharing architectural knowledge. Such approaches should support architects in their daily knowledge-intensive activities. This prevents from falling in an 'ICT-trap', which is caused by an underlying, but unrealistic, assumption that ICT can always support and improve knowledge sharing within organizations.

In this part of the thesis we have studied how to effectively support architects in sharing architectural knowledge. As depicted in Fig. 6.1 in Chapter 6, answering this main research question was only possible by understanding what architects really do and what kind of support for sharing architectural knowledge they need. Below, we first answer these latter two questions, and then revisit our main research question.

12.1.1 What do architects do that involves architectural knowledge sharing? (RQ-II.2)

Sharing architectural knowledge is crucial to prevent this knowledge from dissipating. However, we found in Chapter 7 that successful knowledge sharing can only be achieved if the necessary incentives are created. Architects will only share knowledge with each other if they are motivated to do so. We have identified three incentives for architectural knowledge sharing: the establishment of social ties, more efficient decision making, and knowledge internalization. We argue that the main organizational challenge to sharing architectural knowledge (RQ-II.4) is that these incentives need to be created.

Creating aforementioned incentives can be achieved by meeting a number of prerequisites for sharing architectural knowledge (RQ-II.5). By studying the issues encountered with existing tools and methods at RFA, and trying to address those issues, we learned that alignment between design artifacts, traceability between decisions and descriptions, a central role of architects, and a central architectural knowledge repository, all help in creating an environment in which architects are induced to share architectural knowledge.

Our initial diagnosis of the architecting process of RFA (cf. Chapter 7) also indicated that architects in fact do much more than creating architectural solutions. The interactions they have with other stakeholders (e.g., in the customer organization), extracting useful knowledge from repositories, and collaborating with colleagues are at least equally important activities. This initial understanding of the architecting process was extended with a literature review and insights gained during subsequent case studies.

In the fifth study (see Chapter 11) we culminated all insights by constructing a theory of architecting. This theory was then validated by means of a survey. This study confirmed that the five core activities of architects are: communication, decision making, quality assessment, documentation, and knowledge acquisition. All these activities involve sharing of architectural knowledge, either directly or indirectly, from person to person, or using repositories. Understanding what knowledge-intensive activities architects are involved in not only answered research question RQ-II.2, but also helped to create a focus when creating solutions to support them in these activities.

12.1.2 What do architects need with respect to architectural knowledge sharing? (RQ-II.3)

Our answer to research question RQ-II.3 started with studying knowledge management best practices and typical characteristics of architecting. In Chapter 8 this resulted in the identification of seven desired properties of tool support for sharing architectural knowledge, which answers research question RQ-II.6. These tools should 1) offer stakeholder-specific content, 2) allow easy manipulation of content, 3) be descriptive in nature, 4) support codification and 5) support personalization of architectural knowledge, 6) enable collaboration, and 7) be sticky in nature.

Based on these properties, plus subsequent investigations into the requirements architects at RFA have (research question RQ-II.7) with respect to tool support, we conducted two rounds of tool experimentation. In the first round we built EAGLE, a web portal for sharing architectural knowledge. As discussed in Chapter 8 this portal adheres to all requirements identified earlier. In Chapter 9 we showed that EAGLE also offers 'Just-in-Time' architectural knowledge to architects using its various knowledge modules.

Experimentations with EAGLE indicated that it fell a bit short on community building and communication, while this apparently were two central activities of architects. To even better align to the daily work of architects, in the second round of tool experimentation we tested an enterprise wiki environment at RFA, as discussed in Chapter 10. We found that the applicability of wikis is rather broad. Wikis can be used as encyclopedia of architectural knowledge, as community where architects can find each other, or both.

In the fifth study (see Chapter 11) we aggregated our experience with tool development and experimentation in a theory of support methods for sharing architectural knowledge. This theory was validated by a survey, which confirmed that the five typical categories of support methods are (in order of preference of practicing architects): decision management, search efficiency, community building, knowledge management and intelligent advice.

12.1.3 How to effectively support architects in sharing architectural knowledge? (RQ-II.1)

To answer our main research question we combine our knowledge about what architects do and what they need. In other words, we unify the answers found on research questions RQ-II.2 and RQ-II.3.

Our research over the past four years has significantly furthered our understanding with respect to architectural knowledge sharing in practice. We now understand better what architects do, what their responsibilities are, in which context they operate, and what role knowledge management plays or could play in their daily work. Taken together, this provides a good overview of how to effectively support architects in sharing architectural knowledge, and thus answers our main research question (RQ-II.1). This overview is culminated in the following lessons learned:

1. **Architects do more than just architecting.** Although creating architectural solutions is one of their primary responsibilities, architects spend a considerable amount of time on associated tasks and activities as well (see §12.1.1). A broader view of what architects do in practice is thus required. Consequently, mechanisms to support architects in sharing knowledge should not only include architecting support such as modeling architectural design decisions, or offering templates for architecture descriptions, but also include assistance for stakeholder communication, quality monitoring and searching relevant information.

2. **Architects do not share knowledge automatically.** Time is precious, especially for architects. By acting as 'bridge' between business and IT stakeholders, they spend much time on stakeholder communication, requirement negotiation, and on developing architectural solutions. Sharing architectural knowledge of and by themselves is, however, not an explicit part of their job description, especially not after the fact. Expecting architects to (extensively) share knowledge automatically is therefore not realistic. They need to be motivated to do so. To create these incentives, architectural knowledge sharing support should focus on assisting architects *during* architecting activities, instead of only offering repositories or templates to store their expertise and experiences after the fact. This reduces the overhead created by knowledge sharing and helps in convincing architects of the immediate and direct benefits related to this.

3. **Software architects benefit from lightweight, Just-in-Time, tool support.** Tools, methods or techniques to support architects in sharing knowledge should definitely be *lightweight* to better adapt to the various technical and non-technical

tasks architects work on. This means that support mechanisms should be sufficiently user-friendly, usable, intuitive, and responsive, in order to motivate architects to keep using them. In addition, to properly accommodate architects in their knowledge needs, architectural knowledge should either be delivered to them or become accessible to them *Just-in-Time*.

4. **Effective architectural knowledge sharing follows a hybrid knowledge strategy.** Codification of architectural knowledge using templates, repositories or meta-models, only supports architects to a certain extent. A substantial amount of the knowledge of architects will always remain tacit – i.e., impossible to articulate, let alone codify – and in these circumstances architects must be able to find the right person, instead of the right document. Software architecture knowledge management therefore should follow a hybrid strategy, incorporating both codification and personalization techniques.

5. **Support for sharing architectural knowledge should respect the architect's autonomy.** During our fifth study we found that architects rather stay in control themselves when making architectural design decisions (their core activity) and still need to be convinced of the value of current automated or intelligent support. Methods that support architectural knowledge sharing should respect this. Automated support that 'thinks' for the architect, or tools that prescribe architects what to do are probably not going to be successful. Descriptive support that aligns well to the activities architects spend most of their time on (decision making, communication, quality assessment, and searching information) is far more effective.

6. **Support for sharing architectural knowledge should stimulate both production and consumption of architectural knowledge.** Our fifth study highlighted that support for decision management and search efficiency were most needed according to practicing architects. Search efficiency support assists the autonomous – or lonesome – architect in retrieving (i.e., consumption of) relevant architectural knowledge when needed. Decision management helps architects in producing architectural knowledge, i.e., to manage their thought processes when taking design decisions, solving conflicts, or codifying design rationale. Mature support for sharing architectural knowledge should assist architects in both producing and consuming such knowledge.

Tools and methods to support architects in sharing architectural knowledge should follow these lessons in order to be effective. Experimentation with EAGLE, our web portal for sharing architectural knowledge (Chapter 9), and an enterprise wiki (Chapter 10) indicated that these allround, versatile, descriptive, platforms are appreciated by architects and successful to foster sharing of architectural knowledge in organizations.

12.2 Discussion

The methodology followed in this part of the thesis is action research. Our four case studies focused on

1. diagnosing the architecting process to identify organizational challenges to sharing architectural knowledge (Chapter 7),
2. action planning by identifying desired properties of tool support for sharing such knowledge (Chapter 8),
3. action taking and evaluating these actions by experimenting with EAGLE (Chapter 9), and
4. action taking and evaluating these actions by experimenting with an enterprise wiki platform (Chapter 10).

The fifth and last stage of action research, specifying learning, typically involves generation of a conceptual or theoretical model based on the outcomes of the earlier stages. In Chapter 11 we presented our theory, which we then validated using survey research.

Action research is among the more qualitative approaches. It is often used for achieving valuable goals for the research subjects, which makes it a popular technique for organizational development (Baskerville, 1999). In our case, conducting action research was particularly suitable for understanding the challenges at RFA related to sharing architectural knowledge, and for collaborating with the practicing architects in making improvements in this area. Action research, however, also has its limitations. Kock identifies three threats of action research (Kock, 2004). Below these threats are explained, and we discuss how we dealt with them during our research.

1. The essence of **the uncontrollability threat** is that the environment (i.e., organization) being studied will often change in ways that are completely unexpected, which may force the researchers to revisit their methods, theoretical assumptions, or even the research topic before the action research cycle is completed.

   Our research was susceptible to the uncontrollability threat during the fourth case study at RFA, when RFA's contact person of the GRIFFIN project left the company. Fortunately, we already established good contacts with several other architects and managers in this organization, but still some delay was caused because we needed to convince new managers of the trials we were conducting with our enterprise wiki.

2. The essence of **the contingency threat** is that the vast amounts of data obtained during action research can be rather ‘shallow’ – or difficult to generalize – because the rich context in which these results are collected makes it difficult to separate out different components that refer to particular effects or constructs.

   Our research was to a certain extent vulnerable to the contingency threat, especially during the four case studies conducted at RFA. Although occasionally architects from other teams were interviewed as well, the core of the experimentation was conducted using one team of architects. Because this architect team was positioned as being central to RFA, they offered services to all business units and domains (e.g., industry, finance, government, health care). Still, the main experience and expertise of the software and enterprise architects in this team targeted central and local government systems, because they primarily worked on software development projects in the public domain. We thus ran the risk that the insights gained by interviewing these architects were not completely generalizable over the whole architect population. This was one of the main reasons for including three additional organizations in our survey, that acted as validation study. The results of the survey indicated no significant changes between these four participating organizations, which led us believe that the impact of the contingency threat is not large in our research.

3. The crux of **the subjectivity threat** is that the deep involvement of researchers with client organizations in action research studies may hinder good research by introducing personal biases in the conclusions. After all, it is impossible for a researcher to both act in a detached position and at the same time exert positive intervention on the environment and subjects being studied.

   Although our research may be vulnerable to the subjectivity threat, just like any other action research study, we argue that the way the GRIFFIN consortium was organized minimized any negative effects. Periodic meetings between all researchers of the project ensured that the studies conducted by individual researchers were in line with the overall goals of the GRIFFIN project, and that sound research methods were used. Moreover, in yearly meetings with all industrial partners all research results were presented to each other, after which commonalities and differences between organizations were spotted and the research agenda of the next year was defined. The nature of the project consortium thus prevented the researchers working at the case studies at RFA from becoming biased, so that research data could not be interpreted wrongly or subjectively.

Although we made considerable progress in the domain of architectural knowledge sharing support over the past four years, much more can and needs to be done. Below, we highlight some topics that deserve particular attention in further research.

With respect to understanding what architects do and what they need for sharing architectural knowledge, we are interested in verifying whether our results can be generalized further. As mentioned in §11.5.1 our survey population consisted of Dutch architects alone. We find it interesting to explore whether any cultural or regional effects exist to what architects' typical activities are or what knowledge needs they have.

It is also worth to continue investigating preferences for architectural knowledge sharing support. As discussed in §11.4.4 we did find a sliding scale of priorities of support methods, but we could only make an initial ordering of methods that are more, or less, needed by architects. By forcing architects to explicitly choose between support methods, we can gain a better understanding of the tradeoffs architects make in choosing which support is most useful. Conjoint analysis (Orme, 2006), a technique often used in marketing research, would be suitable as a method in a follow-up survey for this purpose.

With respect to tool development, experiments with EAGLE and our wiki indicated that non-intrusive, integrated environments have much to offer to architects as long as they adhere to the lessons learned described in §12.1. Integration with existing (non-architectural) tools is crucial as well. For example, many large IT organizations already have software to manage documentation and workflow (e.g., MS Sharepoint). To be effective, architects are not helped by yet another tool, but by an environment that further glues existing solutions together. This increases the traceability of architectural knowledge in the organization, and enhances both the functionality and quality of knowledge sharing support for architects. Such an effective ICT infrastructure positively influences the level of structural social capital in an organization (van den Hooff and Huysman, 2009). This in turn stimulates the creation of a community of architects; a community in which sharing of architectural knowledge is not a punishment, but a reward; a community in which even the most lonesome architect feels inclined to share some of his precious expertise.

Part III

Supporting Auditors

By Remco C. de Boer

13 Discovering Architectural Knowledge

Software product audits are a typical example of 'knowledge work'. In this chapter, we present the main challenge taken on in Part III, namely how auditors can be supported in discovering architectural knowledge relevant to an audit. We briefly discuss the main types of architectural knowledge in the context of the audit process, and distill research questions that shall be answered throughout the remainder of this part.

13.1 Introduction

On occasion, organizations may experience the need to verify the quality of a software product. Such a need may arise, for example, prior to acquisition or in the case of contracted-out development. A way to assess the quality of a software product is to let an independent party perform an audit[1].

The architectural design of a software product and the architectural design decisions taken play a key role in software product audits. Architectural design decisions and their rationale provide, for instance, insight into the trade-offs that were considered, the forces that influenced the decisions, and the constraints that were in place. The architectural design that is the result of these decisions allows for comprehension of such

[1]The ISO/IEC 14598-1 international standard (ISO/IEC 14598-1) defines a software product as 'the set of computer programs, procedures, and possibly associated documentation and data'. Quality is defined as 'the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs', while quality evaluation is 'a systematic examination of the extent to which an entity is capable of fulfilling specified requirements'. Consequently, when we refer to a software product audit - i.e., an official examination in which the quality of a software product is evaluated - we refer to a 'systematic examination of the extent to which a set of computer programs, procedures, and possibly associated documentation and data are capable of fulfilling specified requirements'. For reasons of brevity, we occasionally uses the term 'audit' to denote 'software product audit'.

matters as the structure of the software product, its interactions with external systems, and the enterprise environment in which the software product is to be deployed.

Over the past four years, we have collaborated with DNV, especially with their Dutch (IT) advisory and education branch DNV-CIBIT. This unit acts - amongst others - as an independent expert in software product quality assessments and has been one of the industrial partners in the GRIFFIN project. The goal of this collaboration was to investigate the role architectural knowledge plays in software product audits, and to improve upon current practices to manage such knowledge.

In Part I we concluded that a typical challenge related to audits is architectural knowledge discovery. In this part, we will further examine this challenge and investigate related topics. The remainder of this chapter is organized as follows. In §13.2 we provide a high-level overview of the problems addressed in this part; we identify two types of architectural knowledge pertinent to software product audits and discuss their relation to two different stages in the audit process. In §13.3, we derive the research questions that shall be addressed in this part. In §13.4 we briefly outline the research methods applied in our research; these methods will be further detailed in the subsequent chapters. In §13.5 we provide an overview of the chapters in this part and their relation to the research questions from §13.3. In §13.6, finally, we list the prior publications upon which this part has been based.

13.2 Problem Statement

The conclusion in Part I that there is a challenge to enhance the discovery of relevant architectural knowledge was based on a characterization of the use of architectural knowledge within DNV (cf. §4.5). In this section, we examine the use of architectural knowledge in software product audits in more detail.

In a software product audit, two types of architectural knowledge can be distinguished. On the one hand there is architectural knowledge pertaining to the *current state* of the software product; this knowledge – which we may refer to as *IST* architectural knowledge, or *IST-AK*[2] for short – reflects the architectural decisions *taken*. On the other hand there is architectural knowledge pertaining to the *desired state* of the software product; this *SOLL-AK*[3] reflects the architectural decisions *demanded* (or expected). It is the auditor's job to compare the current state with the desired state. Hence, software product audits are knowledge-intensive tasks in which architectural knowledge plays a pivotal role. In other words, audits can be classified as 'knowledge work'.

[2]from the German word *ist*, meaning 'is' or 'as it is'.

[3]from the German word *soll*, meaning 'as it should be'.

Knowledge work, according to Mackenzie Owen (2001), incorporates “the gathering, processing, creating, sharing and disseminating of knowledge” and consists of three distinct stages: input, throughput, and output. In each of these stages, knowledge is employed:

- In the input stage, relevant existing knowledge and data are gathered;
- In the throughput stage, knowledge and data are analyzed and processed;
- In the output stage, the results of the previous stage are recorded and disseminated.

A typical software product audit uses architectural knowledge in these stages as follows:

- Input stage: gather relevant architectural knowledge regarding the the desired state (SOLL-AK) and current state (IST-AK) of the software product;
- Throughput stage: compare the SOLL-AK with the IST-AK;
- Output stage: lay down findings regarding deviations of the IST-AK from the SOLL-AK in a report.

We can distinguish three stakeholders in a software product audit: the customer (i.e., the party who requested the audit), the supplier (i.e., the party who developed the software product), and the auditor (i.e., the party who independently assesses whether the supplier’s software product conforms to the customer’s needs).

The input to a software product audit can be divided into two categories:

1. an evaluation frame (which reflects the SOLL-AK), and
2. the product deliverables (which reflect the IST-AK).

During a software product audit, the product deliverables are assessed against the evaluation frame. In the **input stage**, the product deliverables are obtained from the supplier and the evaluation frame is written by the auditor based on the high-level quality characteristics the customer desires. The auditor usually determines these characteristics, expressed in terms of quality attributes such as ‘maintainability’, ‘reliability’, or ‘efficiency’ (cf. (ISO/IEC 9126-1; van Zeist et al., 1996)), in a workshop with the customer. From these quality attributes, the auditor derives quality criteria that represent concrete measures that should or should not be present in the software product. This definition of an evaluation frame is an important prerequisite for a successful audit.

In the **throughput stage**, the auditors assess the satisfaction of the criteria in the evaluation frame. For each of the criteria from the evaluation frame, the expected effect on the product is determined. The affected artifacts are then examined to establish whether the criteria are satisfied. For instance, (solutions for) criteria regarding scalability are usually expected to be found in the software architecture document. As a consequence of this method, artifacts that are not expected to contain such solutions are considered not in scope of the audit. Nevertheless, these artifacts might be skimmed to gain an overall impression of the product.

Because of the amount of artifacts in a typical software product, the audit work is usually split up over multiple auditors. A typical division is documentation artifacts (i.e., the product documentation) vs. documented artifacts (e.g., the product's source code). Documentation artifacts are often further subdivided because this set may still be very large. Only the very essential documents are then read by all (documentation) auditors; the other documents are only read by a single person. Assessment of code artifacts is often performed with spot checks based on the documentation.

The result from the **output stage** is an audit report. In this report, the (factual) findings are presented followed by a conclusion. The conclusion comprises an interpretation of the findings by the auditors, and identifies for instance risks related to the observed facts.

From our discussions with auditors and our analysis of the use of architectural knowledge within DNV, we identified two problems with the use of architectural knowledge, related to the input stage and throughput stage of an audit, respectively. Both are related to the challenge of architectural knowledge discovery:

1. In the input stage of an audit, the discovery of applicable quality criteria is difficult. Although many criteria are applicable to multiple projects or products, there is no real means of easily sharing these criteria. Audit tools mainly focus on the calculation of code metrics in the throughput stage and not on knowledge reuse from earlier audit projects. In order to find applicable criteria that were used in earlier projects, one has to delve into the earlier evaluation frame documents. This is not only a tedious and time-consuming task, but can also lead to oversights.

2. In the throughput stage, a consequence of the strategy chosen to divide the work load is that none of the auditors has a complete picture of the software product. Although within DNV each of the auditors informs the other auditors and asks them for more information when necessary, the auditors acknowledge there is a potential risk that some issues remain unseen. Unfortunately, the sheer amount of artifacts makes it impossible to have two or more auditors assess all prod-

uct deliverables. This is especially troublesome for the product documentation artifacts, which – unlike code – cannot be automatically analyzed whatsoever.

13.3 Research Questions

In the remainder of this part, we address the challenge that auditors experience in discovering relevant architectural knowledge in the context of a software product audit. Consequently, the main research question is:

RQ- III.1 *How can auditors be supported in discovering relevant architectural knowledge?*

In the problem statement in §13.2, we have seen that auditors experience problems with discovering relevant SOLL-AK and IST-AK in the input and throughput stage of an audit, respectively. Hence, the main research question can be divided into two subquestions, as depicted by the arrows in Fig. 13.1):

RQ- III.2 *How can the discoverability of relevant SOLL-AK be enhanced?*

and

RQ- III.3 *How can the discoverability of relevant IST-AK be enhanced?*

We have seen that the SOLL-AK in an audit is very much related to the quality requirements of the customer. Ultimately, the manifestation of SOLL-AK is in the form of quality criteria, which are preferably to be reused from earlier audits. We could view the quality criteria as an elaborate version of the customer's high-level quality requirements. However, in the throughput stage the quality criteria need to be compared not against requirements, but against the actual architectural design decisions that form the product's IST-AK. Hence, quality criteria also seem to have a relation to the architectural solution: they appear to lie somewhere on the border between requirements and architecture. A better understanding of the role and nature of quality criteria in an audit is necessary if we want to support their reuse. Therefore, we first need to address the question:

RQ- III.4 *What is the relation between requirements and architecture?*

When we have an better understanding of the relation between requirements and architecture, we can build upon this understanding to answer the question:

RQ- III.5 *How can the reuse of quality criteria be supported?*

Regarding the IST-AK, the amount of product artifacts to assess in the throughput stage is a major point of concern for auditors. Especially the amount of documentation artifacts can be overwhelming. We therefore ask:

RQ- III.6 *How can the discovery of AK in software product documentation be supported?*

The fact that there is not a single document that contains (an overview of) all architectural knowledge is an indication that sharing architectural knowledge through documentation is hard. Many software development projects end up with huge stacks of documents in which one can easily lose track. This means that the author's message may not come across. If the use of documentation for sharing architectural knowledge is indeed inherently difficult, we would like to know why. Our final question therefore is:

RQ- III.7 *Why is sharing AK using documentation hard?*

13.4 Research Methods and Methodology

The research reported on in this part has been part of the GRIFFIN project. In this project, we collaborated with several industrial partners. Most of the research for Part III has been conducted in collaboration with DNV.

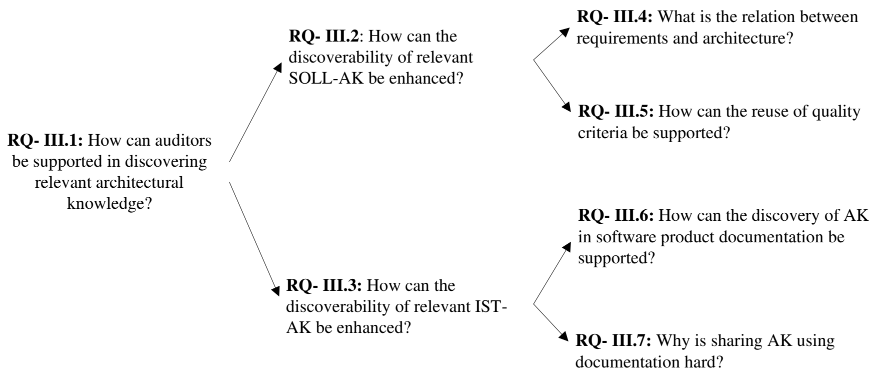

RQ- III.1: How can auditors be supported in discovering relevant architectural knowledge?
RQ- III.2: How can the discoverability of relevant SOLL-AK be enhanced?
RQ- III.3: How can the discoverability of relevant IST-AK be enhanced?
RQ- III.4: What is the relation between requirements and architecture?
RQ- III.5: How can the reuse of quality criteria be supported?
RQ- III.6: How can the discovery of AK in software product documentation be supported?
RQ- III.7: Why is sharing AK using documentation hard?

Figure 13.1: Research questions of Part III

The research questions from §13.3 can be broadly categorized in questions that target ‘world problems’ and questions that target ‘knowledge problems’. Knowledge problems, according to Wieringa and Heerkens (2006), consist of “a lack of knowledge about the world”. Their solution involves a change in the state of our knowledge, and as such they are ‘real’ research problems. World problems, on the other hand, consist of “a difference between the way the world is and the way we think it should be”. Their solution involves a change in the state of the world, and as such they are a type of engineering problem. The research questions RQ-III.5 and RQ-III.6 both address world problems and call for the design of new tools or techniques, whereas research questions RQ-III.4 and RQ-III.7 address knowledge problems and call for a more exploratory type of research.

In order to answer questions RQ-III.5 and RQ-III.6, we followed a constructive research approach. This is “a research procedure for producing innovative constructions, intended to solve problems faced in the real world and, by that means, to make a contribution to the theory of the discipline in which it is applied.” Constructive research can be viewed as “a form of conducting case research parallel to ethnographic, grounded theory, theory illustration, theory testing and action research” ((Lukka, 2003), cited in (Caplinskas and Vasilecas, 2004)).

Our research involved interviews, critical analysis of the literature, and proof of concept development. Interviews were primarily held to gain a better understanding of the types of problems auditors experienced surrounding the (re)use of architectural knowledge in audits. Proofs of concept were used to demonstrate in which way these problems can be overcome. In one case (RQ-III.6), additional interviews were used to validate the proof of concept.

The particular interview technique we used for proof of concept validation is known as the ‘repertory grid technique’. This technique was also the main method used in our exploratory research to answer research question RQ-III.7. A critical analysis of the literature, finally, was the primary method to answer question RQ-III.4.

13.5 Outline of Part III

In the remainder of this part, the distinction between SOLL-AK and IST-AK, and the correspondence to the input and throughput stages of an audit, can be found back in the order of the chapters. Chapter 14 and Chapter 15 together answer question RQ-III.2. Chapter 16 and Chapter 17 together answer question RQ-III.3. All four chapters together answer our main research question RQ-III.1.

The observing reader will notice that the four chapters are not presented in the chronological order of their publication (cf. §13.6). While performing our research, we initially focused on the discovery of IST-AK from software product documentation, since the auditors indicated at that time that this topic had the highest urgency. Later in the project, we also dedicated our efforts to the research problems surrounding (reuse of) SOLL-AK. From the perspective of the audit process, however, presenting the chapters on SOLL-AK before those on IST-AK, rather than in chronological order, provides a more coherent picture of our research results.

This part consists of the following chapters:

- Chapter 13 (this chapter): provides a high-level overview of the audit process, the types of architectural knowledge that play a role in this process, and the research problems addressed in the remainder of Part III.
- Chapter 14: addresses research question RQ-III.4; it describes the relation between requirements and architecture and argues that architecturally significant requirements and architectural design decisions can be treated as equal.
- Chapter 15: addresses research question RQ-III.5; based on the premise from Chapter 14 this chapter presents an ontology for codification of quality criteria and a prototype tool based on that ontology that supports auditors in reusing applicable quality criteria in new audit projects.
- Chapter 16: addresses research question RQ-III.6; it proposes the use of text mining (in particular latent semantic analysis) to discover and guide the auditor to relevant architectural knowledge.
- Chapter 17: addresses research question RQ-III.7; it presents a study from which we hypothesize that the understanding of software documentation is tightly linked to one's role in the development process. For independent auditors, who are not directly involved in the development process, this would imply that they may experience difficulties understanding the software documentation. But even for members of the same development team this link to the process may result in problems fully understanding the documentation.

13.6 Publications

Most of the research presented in Part III has either been published previously or is currently in press. The chapters in this part are based on the following publications.

Parts of Chapter 14 have been published as:

- de Boer, R.C. and H. van Vliet. On the Similarity between Requirements and Architecture. *Journal of Systems and Software*, 82(3):544–550, 2009.

Parts of Chapter 15 have been published as:

- de Boer, R.C. and H. van Vliet. QuOnt: An Ontology for the Reuse of Quality Criteria. In *4th Workshop on SHAring and Reusing architectural Knowledge* (SHARK'09). IEEE Computer Society, 2009.
- de Boer, R.C., P. Lago, A. Telea and H. van Vliet. Ontology-Driven Visualization of Architectural Design Decisions. In *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture* (WICSA/ECSA 2009). In press.

Parts of Chapter 16 have been published as:

- de Boer, R.C. Architectural Knowledge Discovery: Why and How? *ACM SIGSOFT Software Engineering Notes*, 31(5), 2006.
- de Boer, R.C. and H. van Vliet. Constructing a Reading Guide for Software Product Audits. In *6th Working IEEE/IFIP Conference on Software Architecture* (WICSA 2007), IEEE Computer Society, 2007.
- de Boer, R.C. and H. van Vliet. Architectural Knowledge Discovery with Latent Semantic Analysis: Constructing a Reading Guide for Software Product Audits. *Journal of Systems and Software*, 81(9):1456–1469, 2008.

Parts of Chapter 17 have been published as:

- de Boer, R.C. and H. van Vliet. Writing and Reading Software Documentation: How the Development Process may Affect Understanding. In *Cooperative and Human Aspects of Software Engineering* (CHASE 2009), IEEE Computer Society, 2009.

14 On the Similarity Between Requirements and Architecture

Many would agree that there is a relationship between requirements engineering and software architecture. However, there have always been different opinions about the exact nature of this relationship. Nevertheless, all arguments have been based on one overarching notion: that of requirements as problem description and software architecture as the structure of a software system that solves that problem, with components and connectors as the main elements. Recent developments in the software architecture field show a change in how software architecture is perceived. There is a shift from viewing architecture as only structure to a broader view of architectural knowledge that emphasizes the treatment of architectural design decisions as first-class entities. From this emerging perspective we argue that there is no fundamental distinction between architectural decisions and architecturally significant requirements.

14.1 Introduction

The relation between requirements and software architecture has long been subject to debate. As early as 1994, at the first international conference on requirements engineering, a discussion panel shed its light on the role of software architecture in requirements engineering. In their position papers (Shekaran et al., 1994), all panel members in one way or another hinted at a fundamental distinction between requirements engineering and software architecture. Garlan discussed the difference between problem space and solution space; Jackson contrasted application domain with machine domain; Mead stated that the architect represents the software developer's viewpoint and implies that the requirements engineer represents the customer's viewpoint; Potts distinguished between ‘what’ and ‘how’; Reubenstein defined requirements as an index into solutions, and opposes an idealized architecture to the as-built architecture; Shekaran – like Garlan – distinguished also between problem and solution, but further argued that requirements engineering is somewhere in between the two.

Ever since this panel discussion, many have tried to find approaches that bridge the gap between requirements and architecture and integrate the two. Over the years there have been numerous attempts to find such approaches, attempts that have even briefly sparked the dedicated STRAW (software requirements to architectures) workshop series. Those attempts have resulted in many approaches – including the Twin Peaks model (Nuseibeh, 2001b), problem frames (Jackson, 2001), and the CBSP approach (Medvidovic et al., 2003) – that have increased our collective awareness of the relation between requirements and architecture. However, this relation is invariably characterized based on the *distinction* between requirements and architecture; a gap that needs to be bridged, often in terms of requirements representing the problem and architecture being the solution.

A problem with the focus on distinction is that the fuzzy line between what is called ‘requirements’ and what is called ‘architecture’ can be arbitrarily drawn. Commonly used criteria to tell requirements and architecture apart include ‘what’ versus ‘how’, ‘problem’ versus ‘solution’, and (a more pragmatic distinction we found to be used in industry) ‘determined before’ versus ‘determined after the contract with the customer has been signed’. Those criteria all carry a notion of ‘already fixed’ versus ‘what remains to be done’. It is nothing new that in software development a clear line between ‘fixed’ and ‘to be done’ is impossible to define. The discussion by Swartout and Balzer (1982) on this ‘inevitable intertwining of specification and implementation’ holds equally well for the inevitable intertwining of requirements and architecture.

Based on an emerging view on architecture, we hypothesize that there is no fundamental difference between what we call requirements and what we call architecture. Or, to put it more precisely, that architecturally significant requirements are in fact architectural design decisions, and vice versa. We understand that this is a provocative hypothesis that needs further corroboration. In the remainder of this chapter we show how we arrive at this claim, as well as some of its implications.

14.2 Where do Problems End and Solutions Begin?

The ongoing discussion about the relation between requirements and architecture has undoubtedly been influenced by the prevailing view on both fields. In requirements engineering, the prevailing view tends to be one of requirements engineering as problem analysis (cf. Jackson's problem frames, but also goal-oriented approaches such as KAOS). The traditional view on architecture, on the other hand, is one of architecture as solution structure, usually described in boxes-and-arrows type of diagrams. This traditional structure-oriented view is currently shifting to a more knowledge-oriented view.

14.2.1 Requirements engineering as problem analysis

Requirements engineering is usually seen as focusing on the problem domain, as opposed to the solution domain that architecture belongs to. A prototypical example of requirements engineering as problem analysis, Jackson's problem frames are an increasingly popular requirements engineering instrument for problem analysis and problem decomposition. The problem frames framework emphasizes a thorough understanding of the problem before any solutions are considered.

Jackson (2001) discusses the difference between statements on fixed and chosen (or desired) properties in relation to problem analysis by referring to the mood in which statements are expressed. To this end, in his problem frames framework he uses the term 'indicative' for given, objective truths and 'optative' for chosen options. Problem analysis involves specifying the problem domain, the requirements, and the machine specification. The problem domain is the part that is fixed; it is the part of the world where the problem is located which has indicative properties that one is relying on, such as the behaviour of of a car on a road. The chosen parts are the requirements and the machine specification. The requirements are "optative descriptions of what the customer would like to be true in the problem domain", for example[1] the correspondence of a speedometer's display with the current speed of the car; the machine specification is an "optative description [of] the machine's desired behaviour at its interfaces with the problem domain", for instance the way in which a computer receives pulses from a rotating wheel and sets the speedometer's display accordingly.

The limitation of a machine specification to the behaviour at the machine's interfaces suggests a crisp distinction between problem and solution, between requirements and architecture; since the machine domain is considered to provide the solution to the problem at hand, the only options that need to be described are the customer's requirements and the machine's desired behaviour with respect to the problem world. The various solution options, i.e., ways to structure and build the machine, are not yet considered.

However, it has been recognized that problem analysis cannot be completely sep-

[1]see (Jackson, 2001) for details on this and more examples.

arated from the considered solutions, since those solutions themselves may affect the problem world. Rapanotti et al. (2004) propose an extension of problem frames, called architectural frames, to represent 'those architectural elements that impact the problem description'. They demonstrate their approach by defining an architectural frame for the pipe-and-filter architectural style. This style mandates an architecture in which components (the 'filters') have input and output connectors that can be linked to form connections (the 'pipes') through which data flows from component to component. Rapanotti et al. argue that using the pipe-and-filter style for a transformation problem (they use the KWIC problem[2] as an example) introduces new sub-problems regarding input, output, transformation, and scheduling. This modification of the problem world leads to new requirements as well: a number of filters needs to be designed, or reused, to address the original transformation problem, input and output data must be converted to a suitable format, and scheduling must be fair (Rapanotti et al., 2004).

While Rapanotti et al. take the use of a pipe-and-filter architectural style for granted, there are more architectural options applicable to the KWIC problem. Chung et al. (1999) show how a customer's needs determine the relative criticality of requirements which can be satisfied through different architectural alternatives. They discuss a scenario where a KWIC component is to be used in a web-based shopping catalog. The e-shopping vendor's user requirements, such as ease of use, good search time performance, and effective space resource utilization, are translated to system requirements regarding for example interactivity, extensibility, and space performance. The high relative criticality of space performance – the number of items in the catalog is expected to increase rapidly – rules out the use of a pipe-and-filter style for the KWIC component. Another option is to use a shared data architecture, which would meet the required space performance but is ruled out because it fails to meet the extensibility requirements. Instead, Chung et al. conclude that, taking all requirements and criticality values into account, for this particular scenario an implicit invocation style would suit the KWIC component best – even though it does not score as good on space performance as a shared data architecture. Hence there is not a one-on-one mapping from a particular problem and/or requirement to a particular architectural solution.

In summary, although the idea of requirements engineering as problem analysis – separated from solution considerations – seems conceptually clean, in reality this separation doesn't hold true. There is a rather intricate interplay between problem and solution. Choices for particular solution directions involve trade-offs that favour

[2]The 'keyword in context' or KWIC problem is a famous problem in software engineering, particularly due to Parnas' discussion of this problem in the context of modular design (Parnas, 1972). A KWIC index system 'circularly shifts' a given line of text, e.g., a publication title, by repeatedly removing the first word and adding it to the end of the line. The resulting KWIC index is a lexicographically ordered list of all shifted lines, in which one can easily find a title when only part of the title is known.

certain requirements over others. The choice for a certain solution impacts not only which requirements can be satisfied, but – perhaps even more important – also which ones cannot be satisfied. At the same time, the choice for a particular solution may introduce new (sub)problems and hence new requirements.

14.2.2 Software architecture: Beyond solution structure

Until recently, the view on architecture has been relatively stable. Architecture was considered to be the (high-level) structure of a software-intensive system, usually in terms of components and connectors. As of the early 2000s, part of the architecture field is moving from this structure-oriented view on architecture to a more knowledge-oriented view, in which design decisions in particular are treated as first-class entities (cf. Chapter 3). As a result, 'the architecture' is no longer solely regarded as the solution structure, but also – occasionally even *instead* – more and more as the set of design decisions that led to that structure. The actual structure, or architectural design, is merely a reflection of those design decisions.

Architectural design decisions are not only directly related to requirements, but they can also be related to other architectural design decisions. The ontology for architectural design decisions constructed by Kruchten (2004) aims among others to formally describe such relations. Possible relationships between design decisions include constraints, conflicts, and alternatives. For example, the decision to use J2EE constrains the decision to use JBoss (a J2EE application server), and conflicts with the alternative decision to use dotNet. Similarly, the decision to use a pipe-and-filter style constrains the decision to select or design individual filters. Although the impact of the decision on the problem world is not treated explicitly, this is the same kind of mechanism that we saw in §14.2.1; architectural design decisions may introduce new problems to be solved by subsequent decisions.

By breaking down architecture design into individual design decisions, the emerging knowledge-oriented perspective embodies a new way of how we perceive architecture design. It shifts the focus from the result in terms of components and connectors to the process of getting to the architecture design instead. It consequently puts more emphasis on the rationale of an architectural design. It also exposes, again, the fuzziness of the 'problem vs. solution' distinction, this time from a solution perspective.

14.2.3 Problem vs. solution: A false dichotomy?

Many requirements can be posed that do not play a role at all at the architectural level, for instance the requirement to use the metric system for a car's speedometer, i.e., display a car's speed in km/h and not mph. We will not consider such requirements, but we limit our discussion to requirements that are 'architecturally significant', in other words requirements that influence the architecture. This is a natural limitation, since we intend to explore the relation between requirements and architecture. Therefore, in the remainder of this chapter we focus on architecturally significant requirements (ASRs) and architectural design decisions (ADDs) as first-class entities from requirements engineering and architecture design, respectively.

We have already seen that the distinction between problem and solution is at best fuzzy. In Chung's example paraphrased in §14.2.1, for instance, the indicative problem domain properties of a rapid increase of catalog items and limited storage space (implicitly assumed by the original authors) lead to the optative statement that the available space should be effectively used. This statement, clearly an architecturally significant requirement, in turn impacts the eventual choice of architectural style. Note that for the same indicative properties a different requirement could have been phrased, for example the use of scalable storage space. In that case, yet another architectural style might have been more appropriate, or a pipe-and-filter style might have been the best choice after all.

The fuzziness between problem and solution does not merely play a role at a theoretical level, as can be seen from various reports from industry. For example, Poort and de With (2004) argue that requirements conflicts they encounter in practice necessarily "arise from limitations in the solution domain". Along the same line, Kozaczynski (2002) observes that "in practice key requirements are not fully understood until the architecture is baselined". And after analyzing five industrial software architecture design methods, Hofmeister et al. (2007) conclude that all five methods proceed non-sequentially because "the inputs (goals, constraints, etc.) are usually ill-defined, and only get discovered or better understood as the architecture starts to emerge", and that while "most architectural concerns are expressed as requirements on the system, [. . .] they can also include mandated design decisions". Finally, Savolainen and Kuusela (2002) present a system design framework in which they suggest to mix design and requirements specifications when appropriate since "all design details that are not optional can be treated as highly constrained requirements".

Apparently the choice of required behavior given a particular problem can have tremendous impact on the architecture of the machine that deals with that problem. Moreover, the decision to use a particular architectural style again impacts the problem world, where it may introduce new requirements, as well as subsequent ADDs. Hence both ASRs and ADDs appear to live in a grey area between the desired behavior in the problem world and the desired properties of the machine to be constructed. Their origin and their effect are not confined to either problem or solution.

14.3 Why Architectural Requirements and Design Decisions Are Really The Same

The difficulty to distinguish between ASRs and ADDs is perhaps not all that strange if we consider that both are optative statements, or decisions, that embody what one wants to achieve. Those decisions may constrain, or may themselves be constrained by, other decisions. This means that every optative decision based on an indicative problem setting has itself indicative properties from the perspective of subsequent decisions, as depicted in Fig. 14.1. In other words, the decision is reflected in, and may even become part of, the problem world. Some of those decisions we call 'requirements', other 'architectural design decisions', but there is no fundamental difference between the two.

To illustrate this point, let us consider a system that consists of a dedicated machine, say a package router (cf. (Jackson, 2001)), that we need to build. If the hardware configuration has already been determined, it is part of the indicative problem world and its physical layout of pipes and switches would co-determine and constrain the requirements and subsequent architectural design of the software. If, on the other hand, the hardware configuration has yet to be decided upon, architectural design decisions would have to be made to determine the architecture of the software as well as the architecture (or physical layout) of the hardware. Obviously, decisions on the layout of the physical pipes and switches would still constrain the feasibility of requirements and the architecture for the system's software. The only difference with a predetermined

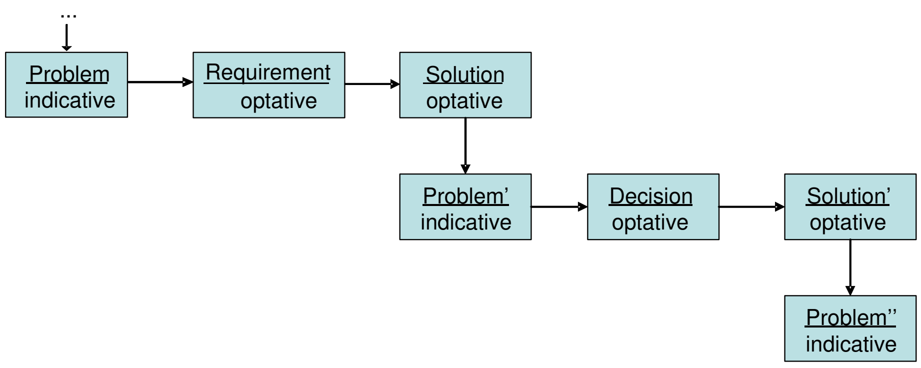


Figure 14.1: Indicative properties of optative decisions

hardware configuration is that in this new situation the architecture of the hardware is also chosen (optative), rather than given (indicative). This means that in the former situation the hardware is part of the problem, and its use a requirement, whereas in the latter situation the hardware is part of the solution and has effects in the problem domain. Similar shifts in indicative and optative properties occur for example in the selection of COTS components or the use of a reference architecture or framework, where new requirements and architectural design decisions both need to cope with the predefined architectural properties of the selected software.

Both ASRs and ADDs have similar effects on the development of software: they indicate preferences for the direction in which the software should develop and prune the design space by ruling out undesired alternatives. This similarity may only be apparent if one does not regard architecture merely as structure, but embraces the view that architectural design decisions are an important element of architectural knowledge.

Relations between architectural design decisions can be modeled as a 'decision loop', in which ADDs introduce new design issues for which new ADDs need to be taken. The theoretical underpinning of such a decision loop has been discussed in Chapter 4. With this decision loop depicted in Fig. 14.2 in mind, we can explain the similarity of ASRs and ADDs from an architectural knowledge point of view.

Suppose that we have a situation in which we need to build a KWIC component that is highly extensible and has effective space performance, basically the same situation that we discussed in §14.2.1. Those requirements ask for several decisions to be taken. In other words, the requirements introduce one or more *decision topics*, most notably 'how to limit the amount of data stored'. This decision topic can be addressed through various alternatives, such as 'shared data', 'abstract data types', and 'implicit invocation'. When one of these alternatives, for instance 'implicit invocation', is chosen (note that the way in which the alternatives are ranked depends on various concerns, including 'extensibility') that alternative becomes an *architectural design decision* which immediately leads to new decision topics (e.g., 'how to distribute events to components').

Furthermore, the requirement that the KWIC component has effective space performance may stem from a higher-level (possibly implicit) *concern*, in this case the concern that the component can handle an increasing amount of data. Indeed, the requirement that the KWIC component must have effective space performance could itself be regarded as an architectural decision that addresses the requirement 'must cope with an increasing amount of data'; it tells *how* the system should cope with that increase, although another alternative (e.g., 'scalable data storage') could have been selected as well.

In short, an architectural design decision (or 'solution') for a requirement (or 'problem') inevitably introduces new 'problems'. If the problem is 'the system must be able

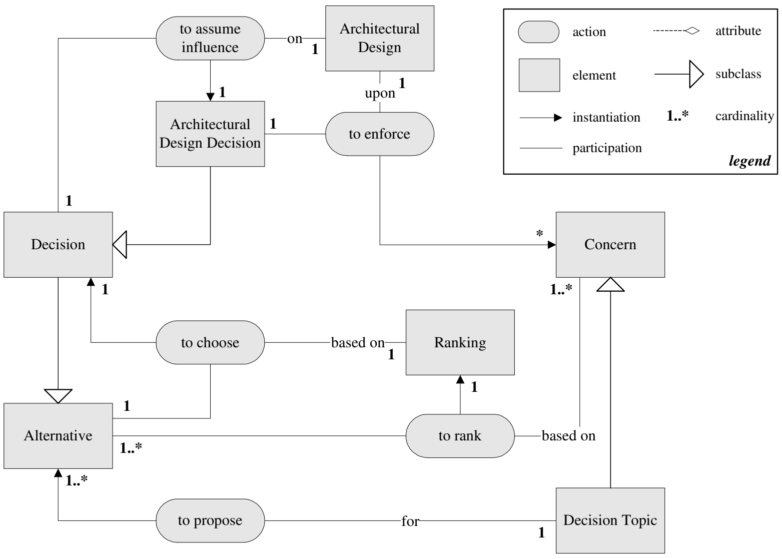


Figure 14.2: Architectural design decision loop

to cope with an increasing amount of data' and the solution is 'by means of effective space performance' then one of the new problems is 'the amount of data stored must be limited'. In this decision loop, concerns (part of the problem) lead to decisions (part of the solution) which lead to new concerns (again, part of the problem). In the example above, effective space performance could just as well be regarded a requirement as an architectural design decision, depending on who you ask – or rather: depending on who has coined it, the requirements engineer or the architect. In any case, the need to cope with an increasing amount of data is the reason (or rationale) behind the choice for effective space performance. Note that the decision loop described here may extend beyond the architecture design phase; some architectural decisions may be taken earlier, some later.

As in the architectural design decision loop, the idea of 'rationale' plays a role in requirements engineering as well, for instance in 'goal-oriented requirements engineering' or GORE (van Lamsweerde, 2001). Given any goal, by asking *why* that goal is needed higher level goals can be found. The other way around, by asking *how* a system may help satisfy a goal, that goal may be refined into new – lower level – goals. Eventually, goals are refined to the level where every goal is realizable, either in the environment or in the software. Those latter goals are the requirements, hence by means of goal refinement a hierarchical structure of goals and requirements develops. In GORE, the decision loop described above appears in a requirements engineering context. The same ‘how’ and ‘why’ questions used in goal refinement can be asked for concerns and decisions in the decision loop. Indeed, even though architectural design decisions are not an explicit part of the GORE refinement process, van Lamsweerde (2003) maintains that “high-level architectural choices are already made during that process” since “for each [refinement] decisions have to be made which in the end will produce different architectures”.

We have now discussed various examples of problem and solution influencing each other, and how this leads to ambiguity regarding whether to call something a requirement or a design decision. But haven’t we unnecessarily and unrealistically complicated things by questioning the distinction between problem and solution in the first place?

As it turns out, there seems to be evidence from the field of psychology that shifting between problem and solution really occurs in human reasoning. Holyoak and Simon (1999) discuss this bidirectional decision making, in which ‘the distinction between premises and conclusions is blurred’, much like our observation of the blurred distinction between ASR and ADD. This type of decision making is thought to play a role in the presence of complexity, conflict, and ambiguity; characteristics that accompany most if not all software engineering situations in which architecture design and requirements engineering would play a pivotal role.

14.4 Architectural Statements and the Magic Well

In a way, architecturally significant requirements and architectural design decisions seem to accumulate in some kind of a ‘magic well’. Observers peering into the well see what they wish for. People wishing to find architecturally significant requirements will see architecturally significant requirements; people looking for architectural design decisions will find architectural design decisions. Hence, this magic well warps the way architectural statements (i.e., ASRs and ADDs) are perceived depending on the observer’s perspective, as shown in Fig. 14.3.

The well does not only determine how we perceive statements, but also affects methods and techniques. Table 14.1 shows requirements engineering and software architecture counterparts of actions that involve the well.

Both fields have their own particular methods and techniques to work with the magic well. For instance, requirements engineering and architecture both have their own standardized approaches to write down what is in the well. IEEE Std. 830 and IEEE Std. 1471 are well known examples of recommended practices for requirements

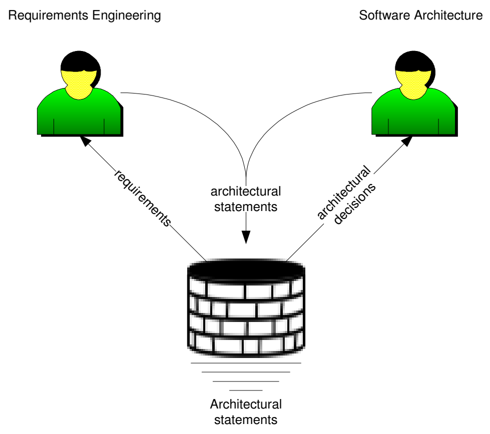


Figure 14.3: Magic well: Different perceptions from different angles

Table 14.1: Methods and techniques: Requirements engineering vs. architecture

Requirements Engineering	**Magic Well**	**Architecture**
Elicitation Negotiation	Creation of statements	Decision making Trade-off analyses
Specification	Drop statements in well	Design
Validation	Compare well contents with reality	Assessment
Documentation	Write down well contents	Description
Requirements management	Structure well contents	Architectural knowledge management

documentation and architecture description respectively.

Dropping freshly created statements in the well is akin to what requirements engineers call specification, and what architects call design. Any elicited requirement or architectural design decision has to be made explicit in one way or another. Not doing so will undoubtedly result in the architectural statement being ignored, or even forgotten, somewhere down the line. Both fields have their own preferred (not necessarily disjunct) sets of techniques for expression, such as natural language text, ER diagrams, UML diagrams, and other box-and-line diagrams.

Elicitation in requirements engineering puts a great deal of emphasis on elicitation techniques, such as interviews, focus groups, use cases, and prototyping. Through negotiation, the priority or relative weight of each requirement is determined. Architecture, on the other hand, focuses more on choosing the right alternative and making trade-off analyses. Requirements are not necessarily left implicit, but are processed less formally in architecture. The Architecture Business Cycle (Bass et al., 2003), for instance, clearly states the importance of addressing stakeholders and their requirements and concerns throughout the architecting process, but does not provide any methodological pointers as to how to elicit those requirements. Note that in both requirements elicitation and architectural decision making a certain amount of creativity is involved; like requirements, architectural solutions may need to be discovered or invented first.

In both requirements engineering and architecture, validation is an integral part of the process. Architecture validation is usually called assessment or evaluation. The architecture community has devised various approaches to architecture assessment that either focus on a single architectural quality such as modifiability (e.g., ALMA (Bengtsson et al., 2004)) or may consider various qualities (e.g., SAAM (Kazman et al., 1994)) and the tradeoffs between them (e.g., ATAM (Kazman et al., 1999)). Requirements validation typically follows a somewhat less formal approach with techniques like inspections and reviews. Although the details of the methods differ, they also have much in common. Both have well-defined process descriptions of how to go about in a validation. But in practice, organizations often use a cafetaria-like approach, in which they borrow method snippets that best fit the situation at hand. Methods in both architecture assessment and requirements validation are often scenario-based. And the results of both surpass the mere identification of errors.

Roeller et al. (2006) compared architectural design decisions with material floating in a pond. “When not touched for a while, they sink and disappear from sight”. If one just drops statements in the well and leaves them there, this is exactly what will happen. Management of the architectural statements in the well has therefore received considerable interest from the requirements engineering community and, more recently, the architecture community. Both requirements management and architectural knowledge management have to do with regarding the well not as an accumulation of statements, but as an information repository. Both aim to impose a structure on the well’s contents that captures the connections between individual requirements or architectural design decisions.

14.5 Cross-fertilization

From Table 14.1 we can try to identify areas in which the architecture community could benefit from a closer inspection of the state of the art in requirements engineering and vice versa. Apart from high-level models such as the Architecture Business

Cycle (ABC), for instance, the architecture community seems to have hardly any standardization or best practices on elicitation. This may not be a big surprise, given that the magic well probably leads us to believe that elicitation (of requirements) is not the architect's job. But on the other hand, the ABC puts quite a bit of emphasis on interaction with stakeholders and understanding their requirements. Stakeholder concerns and business goals play a major role for software architects as well, and using approaches proven and tested in requirements engineering might pay off.

The most interesting area for cross-fertilization, however, is probably the area of management. The emphasis on architectural design decisions and the subsequent focus on architectural knowledge and architectural knowledge management are a relatively recent development in the architecture community. While this has led to many promising initiatives over the short time span of only a few years, the requirements engineering community has also been working on similar issues that the architecture community now encounters. Both fields encounter challenges in areas such as traceability, consistency, evolution, and rationale management.

Current initiatives in architectural knowledge management mainly focus on the development of frameworks and models to capture architectural knowledge (cf. Chapter 3, §3.4). Many architectural knowledge management initiatives seem to have some overlap with approaches in requirements management.

In requirements engineering, goals play a role in many areas related to architectural knowledge management, including traceability, conflict detection and resolution, and exploration of design choices (Rolland and Salinesi, 2005). The requirements engineering community has devised several methods to model goals, such as i* (Yu, 1997) and KAOS (van Lamsweerde, 2001).

An example of a still relatively unexplored area in requirements engineering is that of requirements interdependencies. Dahlstedt and Persson (2005) sketch a research agenda with four major research areas: What is the nature of requirements interdependencies?; How can we identify requirements interdependencies?; How can we describe requirements interdependencies?; and How do we address requirements interdependencies in the software development process? In architectural knowledge management, interdependencies (between architectural design decisions) play an important role as well. The ontology of ADDs that Kruchten (2004) proposes, for instance, tries to address some of the questions above from an architecture perspective. Especially in those management areas where both communities are just beginning to find answers, close collaboration and regular exchange of research results between the two communities should be advantageous to all.

15 Quality Criteria: Structure and Reuse

In this chapter, we propose an ontology for codification of quality criteria. With the similarities between architectural design decisions and (quality) requirements – discussed in Chapter 14 – in mind, we derive this ontology from a combination of three sources: our own work in Chapter 4, the ontology for architectural design decisions proposed by Kruchten (2004), and a collection of informally defined quality criteria used within an industrial organization. The resulting quality ontology can be used to codify quality criteria and their interrelations. This ontology aids the reuse of predefined quality criteria in the input phase of software product audits. We present a prototype implementation of a decision support system that aids auditors in their decision to include or exclude certain quality criteria in a particular audit.

15.1 Introduction

In a software product audit, a customer asks an independent third party – the audit organization – to perform an assessment of the quality of a supplier's product. Auditors of the audit organization need to elicit the customer's idea of 'quality' and compare it with actual characteristics of the software product delivered by the supplier. Hence, one of the first stages of a software product audit, is for the auditor to translate the customer's idea of quality – often expressed in terms of quality attributes such as "the product should be scalable" or "security is important" – to certain quality criteria: concrete measures that should, or should not, be present in the product. For example, in a system where security is important, proper user authentication is a likely quality criterion; without such authentication the necessary level of security is unlikely to be reached.

Whether or not a measure should be present in a software product is often a matter of trade-offs. For example, when user-friendliness is essential, there may arguably be no user authentication measures present; on the other hand, when security is essential user authentication measures are mandatory. If both security and user-friendliness are important, the appropriateness of authentication measures depends on which of the two has precedence. Such trade-offs imply that deciding on a particular quality criterion – i.e., whether or not a particular measure should be present – can be hard.

Software product audits should not be regarded as isolated projects. Rather, individual audits affect each other even if they target unrelated software products. For instance, lessons learned in one project might be applicable to another. Moreover, the applicability of certain quality criteria is not limited to a single project alone. Similar projects might use similar quality criteria, and some general quality criteria might even be applicable to virtually all software products. For example, in high-security systems some form of user authentication will always be needed.

In short, many quality criteria are reusable assets. However, even though there is a gradual increase of interest to use ontologies to capture architectural knowledge, in particular architectural design decisions, there are currently no models known from literature that support the structured codification of quality criteria for reuse. Moreover, while ontologies seem a viable approach to codification, the application of such codified knowledge to everyday practice may be non-trivial. In particular, browsing and searching an architectural knowledge repository for effective reuse can be cumbersome.

Consequently, reuse of quality criteria tends to occur on an ad-hoc basis, and may involve rereading past audit reports to identify previously used criteria applicable to the situation at hand. Some audit organizations may take this one step further and collect reusable criteria in a central location. As part of our research into reuse of quality criteria, we collaborated with DNV, who experienced difficulties in reusing applicable quality criteria from past projects, despite attempts to create a central repository. Since this repository essentially consisted of a list of quality criteria linked to quality attributes, important additional knowledge including dependencies between criteria could not be captured.

In this chapter, we draw on architectural knowledge management theory to propose QuOnt, an ontology that supports codification of quality criteria with reuse as primary goal. The use of ontologies to codify architectural knowledge fits a general trend, as we shall see in §15.3. To support reuse, we must pay particular attention to the relationships between quality criteria and quality attributes, as well as the interrelationships between quality criteria themselves. To this end, in §15.2 we first briefly describe how quality criteria can be viewed as decisions that determine the ‘soll’ architecture of a software product. In §15.4 we then draw analogies between Kruchten’s ontology of architectural design decisions and the repository structure that originated in DNV from their experience with quality criteria. Based on overlap and distinctions between the two structures, we derive our proposed ontology for quality criteria in §15.5. We then demonstrate in §15.6 how criteria codified according to QuOnt can be visualized using ontology-driven visualization – ODV, a novel type of visualization of architectural design decisions – and how ODV provides decision support for auditors to determine which quality criteria should be used in an audit. This decision support consists of three main elements: 1. support for trade-off analysis, 2. support for impact analysis, and 3. support for if-then scenarios. This demonstration shows how QuOnt supports reuse of quality criteria and therewith solves the shortcomings of ad hoc reuse of quality criteria. We finish our discussion in §15.7 with an explanation of the design rationale behind the decision support system, and especially the ontology-driven visualization that it uses.

15.2 Quality Criteria: Deciding the SOLL-Architecture

Architectural knowledge in software product audits exists on at least two levels. First, there is architectural knowledge that relates to the actual state of the software product. This knowledge originates from the product's supplier, and in its codified form can be found in product artifacts such as code and documentation. Secondly, there is architectural knowledge that relates to the desired state of the software product. This knowledge originates from the customer, is enhanced by the auditor, and takes the form of what we call 'quality criteria'.

Quality criteria are in a way similar to what Bass et al. (2003) call 'tactics'. Both relate architectural choices to their effect on quality attributes. A major distinction, however, is that tactics are usually employed from an optimistic perspective: as potential improvements in a forward engineering sense. Therefore, the focus is on tactics as *contributors* to the system's quality (e.g., improve or attain security through user authentication). Quality criteria, on the other hand, have an inherently more pessimistic nature; they need not always be a representation of measures that *contribute* to the desired quality level. Instead, when a certain measure is known to *inhibit* the desired quality level, a quality criterion could be that that measure should *not* be present in the software product. Quality criteria may thus be expressed as an explicit rejection of a particular design option.

For example, 'authenticate users' could indeed be a quality criterion for a system in which security is an important quality attribute. In that case, the quality criterion resembles (or maybe even equals) the idea of a tactic, in that it represents a contribution to attainment of the desired level of security. On the other hand, when security is not important but, let's say, user-friendliness is the prime target, the applicability of 'authenticate users' as a tactic disappears. As a quality criterion, however, it is still an important consideration, since a system in which users are required to authenticate is less user-friendly than a system without authentication mechanisms. Hence, an auditor might still want to look for authentication mechanisms in the system, motivated by the fact that authentication actually lowers the system's quality (i.e., user-friendliness) and therefore should not occur. Obviously, in a realistic situation the desired level of quality will never be expressed in terms of *either* security *or* user-friendliness, but one quality attribute may be preferred over the other. In any case, in a software product audit it is important to know of any positive *and* negative effects of a design choice in order to determine whether that choice is allowed or even required to attain the desired level of quality.

The resemblance between quality criteria and tactics is indicative of the dual nature of quality criteria: they are a reflection of quality requirements pertaining to the software product, but at the same time show characteristics of design decisions. This directly relates the concept of quality criteria to other studies that discuss the characteristics of architectural knowledge, many of which employ a decision-oriented view on architecture and design. Indeed, 'decisions' can be seen as an umbrella concept that unifies different views on architectural knowledge (cf. Chapter 3).

In Chapter 4, we discussed how architectural design decisions are related through what essentially is a decision loop. This loop reflects how architectural design decisions introduce new design issues for which new decisions need to be taken. One of the implications of this loop is that there is no clear-cut distinction between architecturally significant requirements and architectural design decisions. In fact, we have argued in Chapter 14 that requirements and decisions are similar statements, only viewed from different angles. This similarity shows again in the role of quality criteria in a software product audit. Thus viewed, the quality criteria gathered in the input stage of the audit process can be seen as a set of architectural design decisions that outline the desired state, or *Soll*-architecture, of the software product.

15.3 Ontologies and Reasoning Frameworks for Architectural Knowledge

The software architecture community is showing a gradual increase of interest to use ontologies to capture architectural knowledge, in particular architectural design decisions. Several researchers have proposed ontologies or reasoning frameworks for software architecture and software quality. Some of the proposed ontologies are very generic and aim to cover a broad range of software architecture concepts (e.g., (Babu T et al., 2007). Other ontologies and reasoning frameworks are focused more on quality aspects. Unfortunately, none is directly applicable to the selection of quality criteria in the input stage of a software product audit.

SEI's ArchE tool (Diaz-Pace et al., 2008), for instance, serves a goal that is similar to ours. It targets, albeit from a forward engineering point of view, guidance of the architect in selecting the right tactics given certain quality requirements. However, it relies on quantitative reasoning models, which are available for some quality attributes (e.g., performance or modifiability) but not for others (Bachmann et al., 2003). This limits its applicability to only a subset of the quality attributes that one may need to consider in a software product audit.

Erfanian and Shams Aliee (2008) propose an ontological approach to architecture evaluation. Their approach differs from ours in that it targets the throughput stage of a quality assessment. In their method, the actual architecture of the software system is formally codified and compared with a set of tactics and their effects on quality attributes. Our work, on the other hand, targets selection of quality criteria in the input stage of a quality assessment, and therefore does not rely on the formal codification of the actual architecture.

Another quality related ontology originated in the context of service engineering. SECSE's Quality of Service ontology (QoSont) bridges a potential semantic gap between service suppliers and service consumers by providing thorough definitions of Quality of Service properties and their metrics (Sawyer and Maiden, 2009). Although this ontology provides a formal approach to quality attribute specification, it does not address the way in which the desired level of quality can or cannot be attained. This black box perspective on quality contrasts with the white box perspective of our work.

Kruchten (2004), finally, has proposed an ontology for architectural design decisions. This ontology has been driven mainly by the need to document architectural design decisions, either on-the-fly or after-the-fact, to support evolution and maintenance of complex software-intensive systems. Unlike the ontology we propose, it has not been specifically designed to support reuse of preexisting knowledge. Nevertheless, given our perspective on quality criteria as architectural design decisions for a *Soll*-architectureas it exhibits many features that can be translated to the world of quality criteria, as we shall see in §15.4.

15.4 Crises and criteria

In this section, we further analyze the commonalities between quality criteria and architectural design decisions, and how this affects a prospective quality criteria ontology.

Kruchten (2004) argues that there are three major classes of decisions (or ‘crises’): ontocrises (existence decisions, with their counterpart anticrises or non-existence decisions), diacrises (property decisions), and pericrises (executive decisions). Likewise, we can distinguish three analogous major classes of criteria, each of which has to be assessed using a different approach:

- *Ontocriteria* (existence criteria) represent concrete elements or artifacts that must appear in the software product. Anticriteria, their counterpart, represent elements that must not appear in the product. Both ontocriteria and anticriteria can be traced to a single product artifact. An example of an ontocriterion could be ‘there must be a custom API for communication with back-end systems’. An example anticriterion could be ‘there must be no code duplication’.

  This class of criteria are the easiest to assess, since finding a single instance of some element in the software product suffices to prove (for ontocriteria) or disprove (for anticriteria) that the product satisfies the criterion. However, most ontocriteria need to be complemented with diacriteria, another class of criteria, since the mere presence of a certain element in the product is not enough to warrant the desired level of quality. In the case of the above example, the existence of the API must be followed by its use. In actual software product audits, those complementary criteria may be implied by the ontocriterion instead of being fully specified.

- *Diacriteria* (property criteria) represent properties (‘overarching traits’) that hold for the whole system and cannot be traced to a single product artifact. An example diacriterion could be ‘The rationale of design decisions must be documented’, or – as a complement to the example ontocriterion above – ‘All communication with back-end systems must use the custom-made API’. Diacriteria can be quite difficult to assess, since they cannot be traced to a single artifact and therefore require a full and thorough examination of the complete software product. A feasible alternative is to assess diacriteria using spot checks.

- *Pericriteria* (executive criteria) are criteria that are not directly related to the software product’s properties or quality attributes, but to criteria surrounding the audit process itself. An example pericriterion could be ‘All stakeholders agree on the evaluation plan’. Whenever pericriteria are not satisfied, the quality of the audit process itself is in jeopardy.

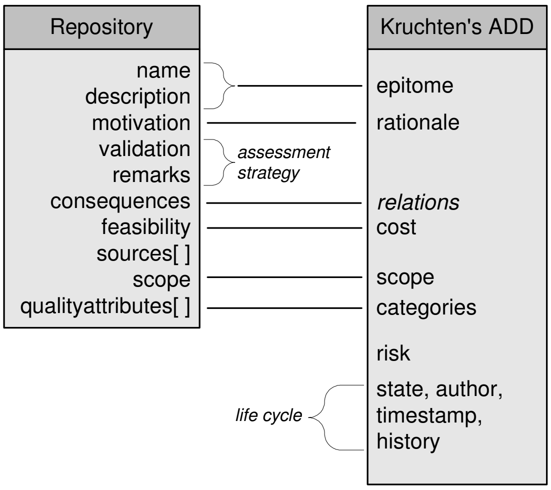


Figure 15.1: A comparison of quality criteria and design decision attributes

In addition to different classes of decisions, Kruchten specifies several attributes that have an almost one-to-one correspondence with quality criteria. Fig. 15.1 shows a comparison between attributes from DNV's quality criteria repository and Kruchten's architectural design decisions.

From Fig. 15.1 we see that the decision attributes epitome, rationale, cost, and scope have a direct counterpart in the quality criteria repository, with one notably different interpretation: for quality criteria, the 'cost' involved in selecting a criterion relates to the feasibility of assessing the product's conformance with that criterion. There are also some differences between the two models, which are a direct result of the different use goals of the repository (reuse) and ontology (documentation):

- Attributes related to a decision's life cycle (state, author, timestamp, history) are not present in the repository;
- Attributes related to a quality criterion's assessment strategy are not present in the decision ontology;
- Since the decision ontology inherently models product-specific decisions, there are no 'sources' to keep track of;
- The risk related to (non-compliance with) a quality criterion would be the *outcome* of an audit process and would not be part of the input, hence there is no

'risk' attribute in the repository.

There are also several comparable attributes that have slightly different representations in the two models:

- *Categories* in Kruchten's ontology are much like tags. They form an open-ended list that may include concerns and quality attributes. In the repository, quality criteria are explicitly related to one or more quality attributes. Likewise, in our ontology we shall draw on the notion of concerns (which include quality attributes) as first class entities. This notion is central to the decision loop that is part of our core model of architectural knowledge (Chapter 4), and enables us a) to further define relations between quality attributes to form a quality model such as the ISO 9126 quality model (ISO/IEC 9126-1), and b) to make inferences (including trade-off analyses) from the relations between criteria and quality attributes (and between quality attributes themselves).
- *Relations* are much more extensively defined by Kruchten than in the repository, where they are just a textual description of 'consequences'. Kruchten recognizes 10 types of decision-to-decision relationships that can be transferred directly to criterion-to-criterion relationships. Two additional relationships ('traces from/to' and 'does not comply with') are the type of relationships auditors would use during the throughput stage of an audit to relate quality criteria to architectural design decisions and other artifacts in the software product (e.g., method X in class Y *does not comply with* criterion 'no code duplication').

Finally, from discussions with DNV's auditors we know that, apart from direct relationships between criteria, one other important piece of knowledge should be captured that is neither present in the existing quality criteria repository, nor in Kruchten's design decision ontology. In essence, the effects from quality criteria on quality attributes must be reified, so that comparisons between the effects can be made. This makes it possible to capture such statements as '1024 bit encryption has a higher positive effect on security than 512 bit encryption'.

15.5 QuOnt: An Ontology for the Reuse of Quality Criteria

Fig. 15.2 depicts the essential concepts and relationships of QuOnt, our proposed ontology of quality criteria, based on our analysis from the previous section. It consists of the following elements:

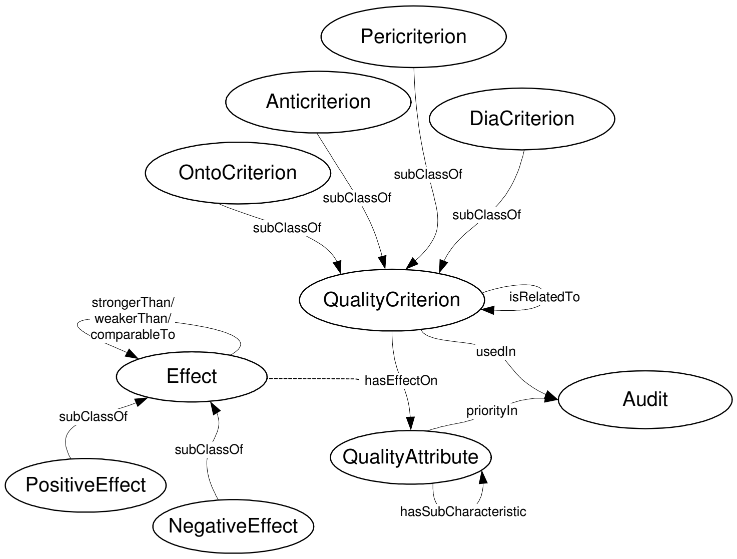


Figure 15.2: The QuOnt ontology

QualityCriterion is the ontology's main element. As discussed in §15.4, it has the attributes name, description, motivation, validation, remarks, feasibility, and scope. The four types of criteria – ontocriteria, anticriteria, diacriteria, and pericriteria – are modeled as subclasses of this element. The *isRelatedTo* relationships capture how a quality criterion can be related to other quality criteria (discussed in more detail below).

QualityAttribute represents a quality attribute that can be further specialized in subattributes. For example, in ISO 9126 'efficiency' is further divided into 'time behaviour', 'resource utilisation', and 'efficiency compliance' (ISO/IEC 9126-1).

Effect is a reified relation from criterion to quality attribute, having two attributes: effect type (positive or negative) and $[0 \ldots n]$ reciprocal relations to other 'effect' relationships, which indicate the relative strength (stronger than, weaker than, or comparable) of the 'effect' relations.

Audit models a software product audit in which particular quality criteria have been used to assess a prioritized set of quality attributes. The *usedIn* relation captures the relation between criteria and audits. The *priorityIn* relation captures the relation between quality attributes and audits.

We recognize ten ways in which criteria can be related. Following the expression of Kruchten's decision relations in terms of the core model's decision loop elements (cf. Chapter 4, §4.6.3), these different types of relationships can be expressed as constraints on a QuOnt ontology instance. Let x and y be quality criteria, let relation$_{x,y}$ denote a relation from criterion x to criterion y, and let usedIn$_x$ denote the use of criterion x in an audit (i.e., the decision that the measure prescribed or ruled out by x must be present in respectively absent from the audited software product). Then the ontological constraints from Table 15.1 hold on the relationships between x and y. These constraints on the one hand confine the facts that can be specified in a QuOnt-based knowledge base, and on the other hand may act as production rules that support the selection of quality criteria.

In the remainder of this chapter, we mainly focus on five types of relationships: *constrains*, *subsumes*, *conflicts/forbids*, and *alternative*. First, when a quality criterion x *constrains* another criterion y, y cannot be used in the audit unless x is also used (C2). Second, when a quality criterion x *subsumes* another criterion y, the use of x implies the use of y (C6). Third, when criterion x *conflicts* with criterion y, x *forbids* y and y *forbids* x (C8); this means that when x is used, y may not be used and vice versa, unless the decision is made that y overrides x (C5). Finally, when criterion x is an *alternative* to criterion y, x and y cannot both be used at the same time (C9). This is a transitive relation (C10). Of the remaining five relationships, *enables* and *overrides* are not supported by the particular reasoning engine our decision support system uses (see also §15.7); *isBoundTo* and *depends* are shorthands for combinations of other relations; and *comprises* can be seen as a trivial extension of *constrains*.

15.6 QuOnt in Action: Selecting Quality Criteria for Reuse with Ontology-Driven Visualization

While ontologies seem a viable approach to capture architectural knowledge, the application of such codified knowledge to everyday practice may be non-trivial. In particular, it can be difficult to explore and search an architectural knowledge repository so that previously captured knowledge can be reused. A demonstration to the auditors of DNV of an early prototype of a quality criteria knowledge base (Loza Torres, 2008) revealed the need for proper visualization of the knowledge base's contents.

In their early work on the visualization of codified architectural design decisions, Lee and Kruchten (2008) distinguish four types of visualization: a simple decision *table* showing design decisions with their attributes and a separate table of relations between decisions; a decision *structure* visualization showing decisions and their rela-

Table 15.1: Relations as constraints on a QuOnt instance

Relation	**Constraints**	
enables	C1	$\forall x, y$: enables$_{x,y} \Rightarrow \neg$constrains$_{x,y}$
constrains	C2	$\forall x, y$: constrains$_{x,y} \Rightarrow (\neg$usedIn$_x \Rightarrow \neg$usedIn$_y)$
isBoundTo	C3	$\forall x, y$: constrains$_{x,y} \wedge$ constrains$_{y,x} \Rightarrow$ isBoundTo$_{x,y}$
	C4	$\forall x, y$: isBoundTo$_{x,y} \Rightarrow$ isBoundTo$_{y,x}$
forbids	C5	$\forall x, y$: forbids$_{x,y} \Rightarrow ($usedIn$_x \Rightarrow \neg$usedIn$_y) \vee$ overrides$_{y,x}$
subsumes	C6	$\forall x, y$: subsumes$_{x,y} \Rightarrow ($usedIn$_x \Rightarrow$ usedIn$_y)$
conflicts	C7	$\forall x, y$: conflicts$_{x,y} \Rightarrow$ forbids$_{x,y} \wedge$ forbids$_{y,x}$
overrides	C8	$\forall x, y$: overrides$_{x,y} \Rightarrow$ forbids$_{y,x} \wedge$ usedIn$_x$
alternative	C9	$\forall x, y$: alternative$_{x,y} \Rightarrow \neg($usedIn$_x \wedge$ usedIn$_y)$
	C10	$\forall x, y, z$: alternative$_{x,y} \wedge$ alternative$_{y,z} \Rightarrow$ alternative$_{x,z}$
comprises	C11	$\forall x, y$: comprises$_{x,y_{1,2,\ldots,n}} \Rightarrow (\neg$usedIn$_x \Rightarrow \neg$usedIn$_{y_1} \wedge \neg$usedIn$_{y_2} \wedge \ldots \wedge \neg$usedIn$_{y_n})$
depends	C12	$\forall x, y$: depends$_{x,y} \Rightarrow$ constrains$_{y,x} \vee$ comprises$_{y,x} \vee$ overrides$_{x,y}$

tions as nodes and edges in a graph; a decision *chronology* visualization showing how decisions evolve; and a decision *impact* visualization that shows which decisions may be impacted by a change. They conclude that more case studies, and also additional visualization techniques, may be required.

According to Lee and Kruchten, decision tables are the most often used type of visualization for browsing. Yet, such a view has several drawbacks. Most notably, a basic list or table is not very effective in showing relationships. As such, it ignores much of the added value of using an ontology. On the other hand, a decision-structure visualization, which seems to be the most natural visualization for a decision ontology, has drawbacks too. While it accurately represents decisions and their relationships, the resulting graph can become cluttered and thus incomprehensible for all but the smallest data sets. Moreover, a graph-like visualization entails quite a radical deviation from the tabular view on information that many practitioners are most accustomed to.

In this section, we shall present a decision support system based on QuOnt and an ontology-driven visualization (ODV) that combines the strengths of the decision-table and decision-structure visualizations, and overcomes their drawbacks. This visualization revolves around two tabular views that show design decisions and their mutual relations, respectively. While not unlike Lee and Kruchten's decision/relationship tables, our visualization adds to the standard tabular view the capability to infer on-the-fly several decision attributes (i.e., columns) from the structural information captured by the decision ontology. Moreover, we provide several easy-to-use interaction and visual highlighting mechanisms that simplify the process of decision-making in the potentially large and complex state space implied by the underlying ontology. Overall, our aim is to enable users to employ the added value of a decision ontology without losing the simplicity of tabular information visualization.

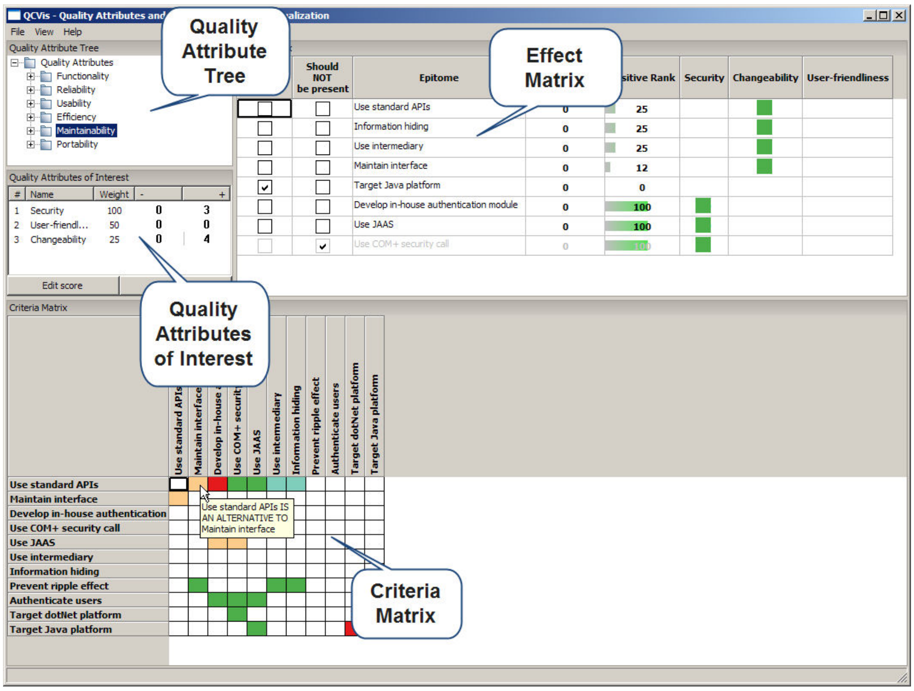
We present the decision support system by means of three usage scenarios derived from an auditor's actual practice and described step-by-step. Although different decision support elements are present in all scenarios, each scenario focuses on a particular aspect of the decision support provided by the ontology-driven visualization; the first scenario focuses on trade-off analysis, the second on impact analysis, and the third on 'if-then' scenarios. For reasons of confidentiality, we are not able to disclose quality criteria used by DNV. We therefore instantiate the ontology with several tactics described by Bass et al. (2003, Ch.5), since – as we have seen in §15.2 – tactics may serve as quality criteria, provided their positive *and* negative effects on quality attributes are taken into account.

Throughout the scenarios, we shall refer to different widgets, or areas, of the user interface as follows (see also Fig. 15.3)[1]. The 'Quality attribute tree' shows the hierarchy of quality attributes according to a particular quality model, in this case the extended ISO-9126 or 'Quint' model (van Zeist et al., 1996). The 'Quality attributes of interest' area shows the quality attributes of interest, which capture the customer's idea of 'quality' and are an input to the remainder of the audit. The 'Effect matrix' shows the quality criteria relevant to the current audit. Relevant criteria are those criteria that have a positive or negative effect on one or more attributes of interest, as well as criteria for which it is determined that they should or should not be present in the product. The 'Criteria matrix' shows the relations between quality criteria. All areas are correlated by means of interactive selection and drag-and-drop operations, thereby allowing the auditor to both construct and query data to support different audit scenarios.

15.6.1 Scenario 1: Trade-off analysis

In the first scenario, an auditor uses ontology-driven visualization of a QuOnt instance to perform a trade-off analysis for determining which quality criteria to include in an audit. The audit is done on behalf of BSO, a fictional enterprise that wants to assess the quality of a new human resource management (HRM) system being developed by a third-party, which will allow employees to view salary statements and request holiday leave. Together with BSO, the auditor establishes that this HRM system should be firstly secure, secondly user-friendly, and finally also easily changeable, since BSO's

[1]Full-color versions of the screenshots in this section have been included in the digital version of this thesis (available through VU-DARE) and can be obtained separately from `http://www.cs.vu.nl/~remco/Ch15-figures.pdf`



Quality Attributes of Interest

#	Name	Weight	-	+
1	Security	100	0	3
2	User-friendl...	50	0	0
3	Changeability	25	0	4

	Should NOT be present	Epitome		sitive Rank	Security	Changeability	User-friendliness
☐	☐	Use standard APIs	0	25		🟩	
☐	☐	Information hiding	0	25		🟩	
☐	☐	Use intermediary	0	25		🟩	
☐	☐	Maintain interface	0	12		🟩	
✓	☐	Target Java platform	0	0			
☐	☐	Develop in-house authentication module	0	100	🟩		
☐	☐	Use JAAS	0	100	🟩		
☐	✓	Use COM+ security call	0	100	🟩		

Figure 15.3: Decision support system - visualization overview

internal IT department will eventually do the maintenance. Based on this prioritized list of quality attributes (1. security, 2. user-friendliness, 3. changeability), the auditor now needs to determine which quality criteria should be used in this audit.

We assume the audit organization has used QuOnt to codify quality criteria from previous audits (not described here). Fig. 15.4 shows a part of the knowledge base available for this audit, comprised of a QuOnt instance. We deliberately consider only part of the knowledge base, so we can focus on just three interrelated quality criteria relevant in our audit. This instance is valid under the constraints outlined in Table 15.1, and consists of the following elements:

OntoCriterion OC1 USE PASSWORDS

OntoCriterion OC2 SINGLE SIGN-ON

DiaCriterion DC1 AUTHENTICATE USERS

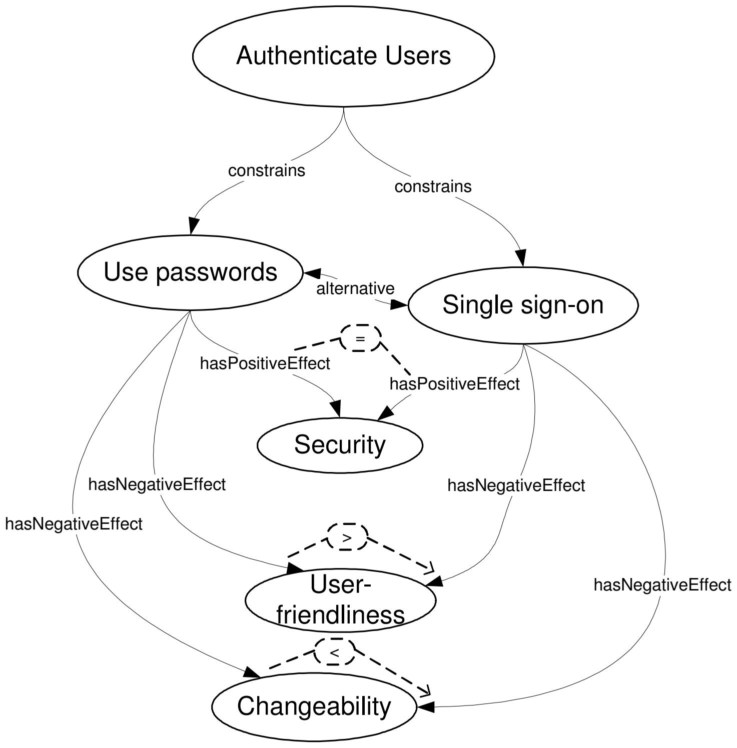


Figure 15.4: Ontology instance for scenario 1

Effect E1 OC1 hasPositiveEffectOn Security

Effect E2 OC2 hasPositiveEffectOn Security

Effect E3 OC1 hasNegativeEffectOn User-friendliness

Effect E4 OC2 hasNegativeEffectOn User-friendliness

Effect E5 OC1 hasNegativeEffectOn Changeability

Effect E6 OC2 hasNegativeEffectOn Changeability

Relation R1 E1 comparableTo E2

Relation R2 E3 strongerThan E4 (consequently E4 weakerThan E3)

Relation R3 E5 weakerThan E6 (consequently E6 strongerThan E5)

Relation R4 OC1 alternativeTo OC2

Relation R5 DC1 constrains OC1

Relation R6 DC1 constrains OC2

In short, the criteria USE PASSWORDS and SINGLE SIGN-ON are alternatives that are both constrained by AUTHENTICATE USERS. Both of them have a comparable positive effect on security. While the negative effect of the use of passwords on user-friendliness is stronger than that of single-sign on, its negative effect on the product's changeability is weaker.

Quality attribute selection

Given a list of prioritized quality attributes, the knowledge base can be used to support the auditor's decisions which quality criteria could be included. In short, criteria with a known effect on one of the high-priority quality attributes (including the attribute's sub-characteristics), should serve as an assessment baseline in the throughput stage of the audit. The auditor's first task, therefore, is to select and prioritize the quality attributes to be used in his audit.

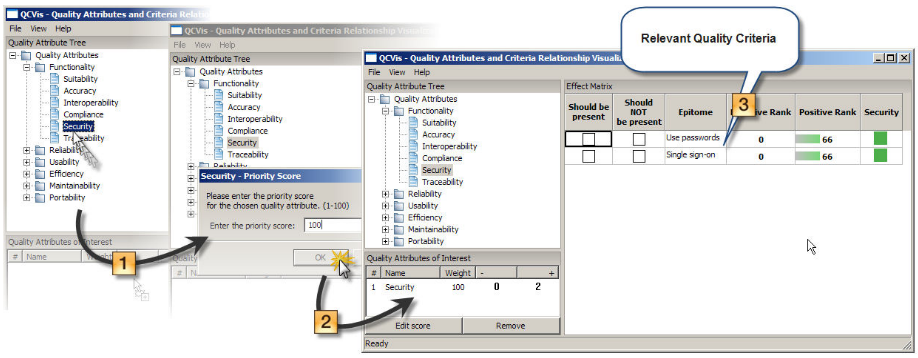
Priorities are quantified on a scale from 1 (lowest) to 100 (highest). The auditor usually elicits these values directly from the customer, e.g., using the '100-point method' (or 'hundred-dollar test' (Leffingwell and Widrig, 2003)) or similar workshop techniques. When such a workshop is infeasible, the auditor may assign a score to the (ordinal) prioritization expressed by the customer, e.g., 100 to the highest-priority quality attribute, 90 to the second highest, and so on.

The ODV supports interactive prioritizing of attributes: quality attributes can be dragged from the attribute tree and dropped in the list of quality attributes of interest below. Fig. 15.5 (step 1) shows this for the 'security' attribute. As part of the drop action, the user is asked to enter the priority score of the selected attribute (Fig. 15.5, step 2).

As soon as a quality attribute has been selected and prioritized, the tool inspects the underlying ontology and updates the effect matrix to display all quality criteria that have an effect on any of the quality attributes of interest. Fig. 15.5 (step 3) shows the criteria USE PASSWORDS and SINGLE SIGN-ON, in the effect matrix, which both have an effect on the 'security' attribute that was just selected. Just as for 'security', the auditor selects and prioritizes the two other attributes of interest, 'user-friendliness' and 'changeability', with a score of 50 and 25, respectively. Fig. 15.6 shows the result.

Effect matrix

The effect matrix has a row for each quality criterion relevant to the current audit, and several columns as follows. The rightmost n columns describe each of the n quality



Security - Priority Score

Please enter the priority score for the chosen quality attribute. (1-100)

Enter the priority score: 100

#	Name	Weight	-	+
1	Security	100	0	2

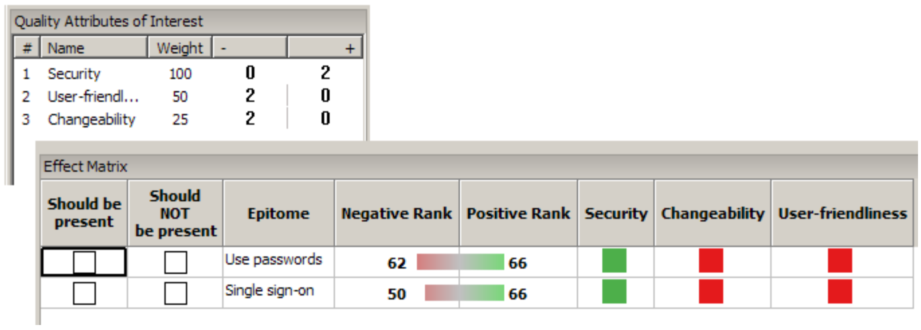
Should be present	Should NOT be present	Epitome	ive Rank	Positive Rank	Security
☐	☐	Use passwords	0	66	🟩
☐	☐	Single sign-on	0	66	🟩

Figure 15.5: Selection and prioritization of relevant quality attributes for an analysis scenario

attributes of interest. A cell at (row=i,column=j) in these columns is colored red, green, or white to show that criterion i has a negative, positive, and respectively no effect on quality attribute j. In Fig. 15.6, there are $n = 3$ such columns (the rightmost ones) for our three attributes of interest: security, changeability, and user-friendliness. We see that both criteria have a positive effect on security, and a negative effect on changeability and user-friendliness.

The effect matrix has two additional columns: 'negative rank' and 'positive rank' (columns 4 and 5 in Fig. 15.6). These show the overall negative, respectively positive effects of a quality criterion on all quality attributes, using scaled color bars. Longer bars represent higher values (also shown numerically[2]). Negative-rank bars are shaded from transparent gray (low values) to saturated red (high values). Positive-rank bars are shaded from transparent gray (low values) to saturated green (high values). In this way, the user's attention is strongly drawn to high positive or negative values, whereas low values are less prominent (Spence, 2007). In our example in Fig. 15.6 we see that, although the overall positive effect of both criteria USE PASSWORDS and SINGLE SIGN-ON is comparable, the overall negative effect of USE PASSWORDS exceeds that of SINGLE SIGN-ON (longer bar in column 4, row 2 than in column 4, row 1). This shows that, while both quality criteria have a comparable positive effect on security, USE PASSWORDS has a larger negative effect on user-friendliness (the second-highest

[2]We calculate these values using partial ordering (Carlsen, 2008). The score of a quality attribute is divided by the average rank of a criterion in the set of criteria partially ordered on effect strength. In Fig. 15.6, the negative value 62 for USE PASSWORDS, for example, is given by $\frac{50}{1} + \frac{25}{2}$, since the average rank of USE PASSWORDS based on the strength of its effect on user-friendliness is 1, and for its effect on changeability 2.



Quality Attributes of Interest

#	Name	Weight	-	+
1	Security	100	0	2
2	User-friendl...	50	2	0
3	Changeability	25	2	0

Effect Matrix

Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank	Security	Changeability	User-friendliness
☐	☐	Use passwords	62	66	🟩	🟥	🟥
☐	☐	Single sign-on	50	66	🟩	🟥	🟥

Figure 15.6: Effect matrix, all attributes of interest

priority quality attribute) than SINGLE SIGN-ON.

Having seen this, the auditor decides that SINGLE SIGN-ON is a criterion that should be present in the audited software. This is done by clicking the 'should be present' column checkbox for SINGLE SIGN-ON (row 2, column 1, see Fig. 15.7). A similar column with checkboxes for 'should not be present' quality criteria is provided in the effect matrix. Using these inputs, auditors can indicate which mix of quality criteria best matches the client's requirements.

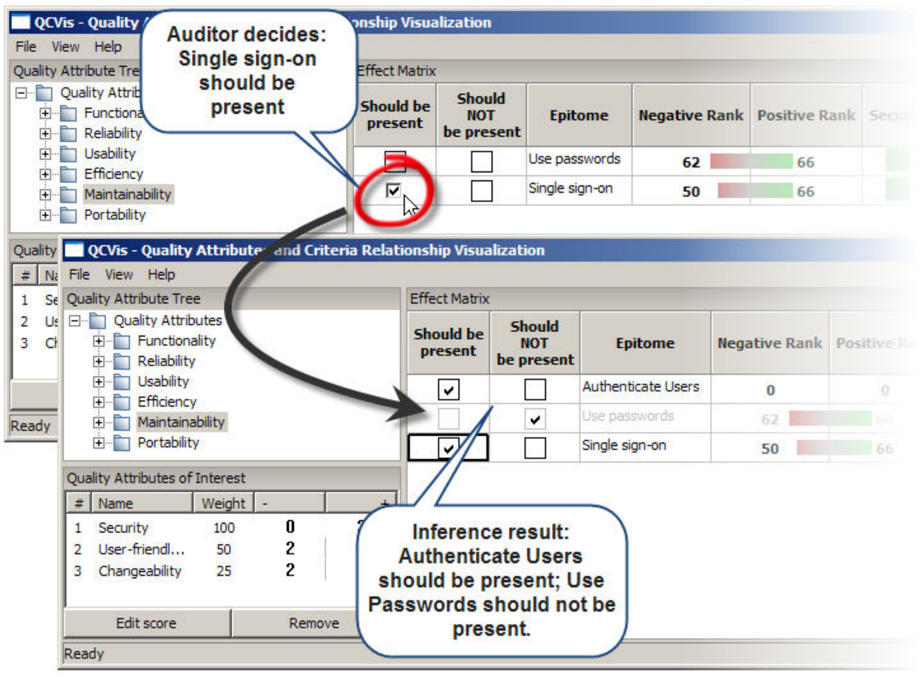
As soon as the auditor makes his decision by (un)checking a quality criterion, a reasoning engine dynamically inspects the ontology to determine which new facts can be inferred. In this case, since USE PASSWORDS and SINGLE SIGN-ON are alternatives, they cannot be both present in the software product (cf. C9 in Table 15.1 and Fig. 15.4). Hence, from the auditor's decision that SINGLE SIGN-ON should be present in the product, the reasoning engine infers that USE PASSWORDS should not be present; the checkbox 'should not be present' of criterion USE PASSWORDS is automatically checked and its name grayed out to reflect this. Moreover, since AUTHENTICATE USERS constrains SINGLE SIGN-ON, the presence of SINGLE SIGN-ON implies that AUTHENTICATE USERS should also be present; so the checkbox 'should be present' of criterion AUTHENTICATE USERS is checked accordingly, and the inference engine adds AUTHENTICATE USERS to the effect matrix. This inference is done automatically, and enables the ontological relations to be reflected directly in the effect matrix. The updated matrix showing which measures should be present is shown in Fig. 15.7. In this image, the auditor can see all quality criteria for this audit: the HRM system 1. should provide user authentication, 2. should provide a single sign-on facility, and 3. should not use passwords.

This simple scenario shows how the system supports the auditor in deciding which quality criteria to use. The decision input uses the visualized attribute trade-offs, shown as high or low positive or negative ranks. Typical decision-making will have measures with a high positive rank present in the software product, and avoid measures with a high negative rank. When the user records his decision, the reasoning engine determines and visualizes the impact of that decision on other quality criteria. This impact can be analyzed for further decision refinement, as described in the next section, for a more complicated scenario.

15.6.2 Scenario 2: Impact analysis

In this scenario, the auditor has the same goal as in scenario 1: assess the quality of an HRM system. This time, however, we will examine a larger part of the knowledge base, and consider more quality criteria than in the first scenario.

Fig. 15.8 shows a total of eleven criteria. Three criteria (INFORMATION HIDING,


Auditor decides: Single sign-on should be present
Inference result: Authenticate Users should be present; Use Passwords should not be present.
QCVis - Quality Attributes and Criteria Relationship Visualization
File View Help
Quality Attribute Tree
Quality Attributes
Functionality
Reliability
Usability
Efficiency
Maintainability
Portability
Effect Matrix
Quality Attributes of Interest
Edit score
Remove
Ready

Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank
☐	☐	Use passwords	62	66
✓	☐	Single sign-on	50	66

Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank
✓	☐	Authenticate Users	0	0
☐	✓	Use passwords	62	66
✓	☐	Single sign-on	50	66

#	Name	Weight	-
1	Security	100	0
2	User-friendl...	50	2
3	Changeability	25	2

Figure 15.7: Selection and inference of quality criteria

USE INTERMEDIARY, and MAINTAIN INTERFACE) have a positive effect[3] on changeability. Three other criteria (USE JAAS, DEVELOP IN-HOUSE AUTHENTICATION MODULE (or DEVELOP IN-HOUSE for short), and USE COM+ SECURITY CALL) have a positive effect on security. Additionally, the criterion USE INTERMEDIARY has a negative effect on performance due to the overhead it introduces. The other criteria have no direct effect on any quality attribute.

Again, the customer indicates that for the HRM system security is the most important quality attribute, followed by user-friendliness and changeability. Additionally, the customer stresses that the product should be built in Java, since his internal IT department only has experience with Java maintenance. Because of the upfront requirement that Java should be used, the auditor starts with a preselected criterion TARGET JAVA PLATFORM, even though no quality attributes have been selected yet (Fig. 15.9).

Just as in scenario 1, the auditor selects and assigns priorities to the quality attributes of interest, and views all quality criteria inferred by the tool. With 'security' being the most important quality attribute, the auditor wants to analyze which quality criteria have an effect on security. Hence, he sorts the effect matrix on the 'security' column by clicking the column's label. Now the quality criteria with the highest effect on security are placed at the top of the effect matrix, as shown in Fig. 15.10(a) (top image). These are the topmost three rows of the effect matrix, all having green cells in the 'Security' column; all three effects are positive. Out of these, USE COM+ SECURITY CALL, has been marked 'should not be used' by the reasoning engine, since it is constrained by TARGET DOTNET PLATFORM which in turn conflicts with the initially preselected criterion TARGET JAVA PLATFORM (cf. Fig. 15.8). This leaves the auditor with two security-related criteria to choose from: USE JAAS and DEVELOP IN-HOUSE.

Criteria matrix

Since USE JAAS and DEVELOP IN-HOUSE both have a comparable positive effect on security, both are eligible to be selected as practices that should be present in the audited product. To further decide, the auditor wants to inspect the relations between criteria. Our tools supports this task with its third and last area: the Criteria matrix (Fig. 15.10(a) bottom). This area shows all relations between quality criteria as an adjacency matrix. Each relation, i.e., matrix cell, is colored using the color scheme shown in Fig. 15.10(b). Red cells show relations that, when the auditor decides a particular

[3]For presentation conciseness, we assume that the strengths of all effects on a particular quality attribute are comparable. Hence, Fig. 15.8 only indicates that a criterion has a positive ('+') or negative ('-') effect on an attribute, followed by the attribute's name. Unlike in Fig. 15.4, we do not draw relations between the effects that indicate their relative strength so the diagram remains readable.

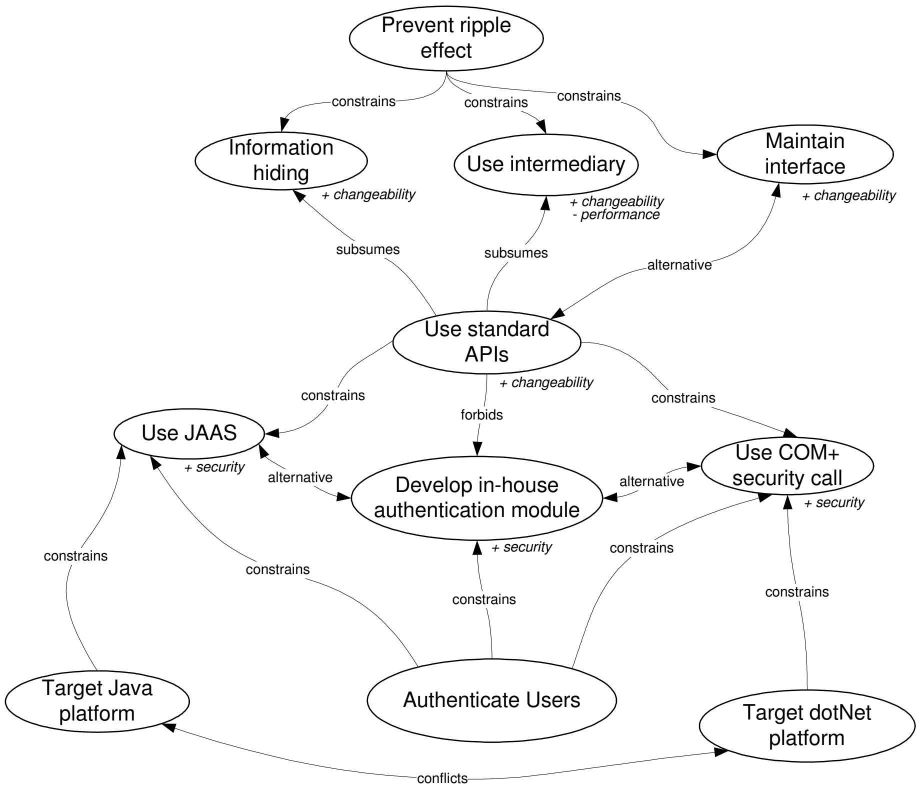
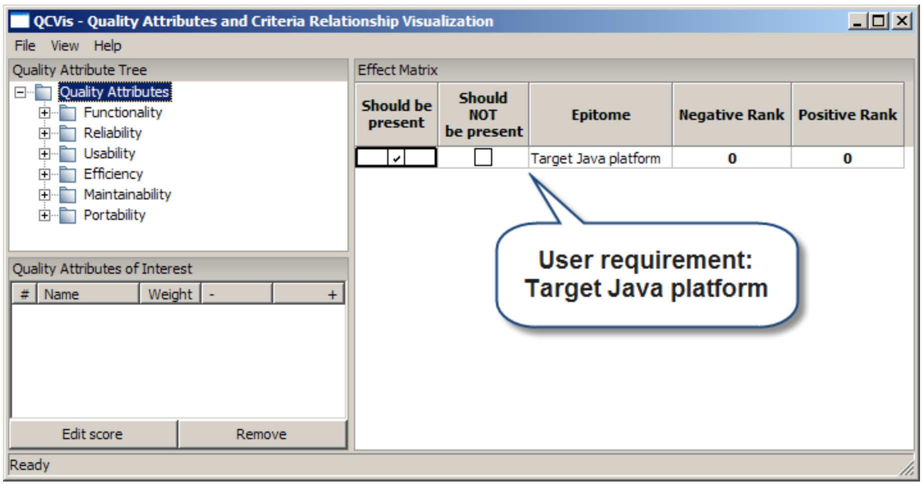


Figure 15.8: Ontology instance for scenario 2



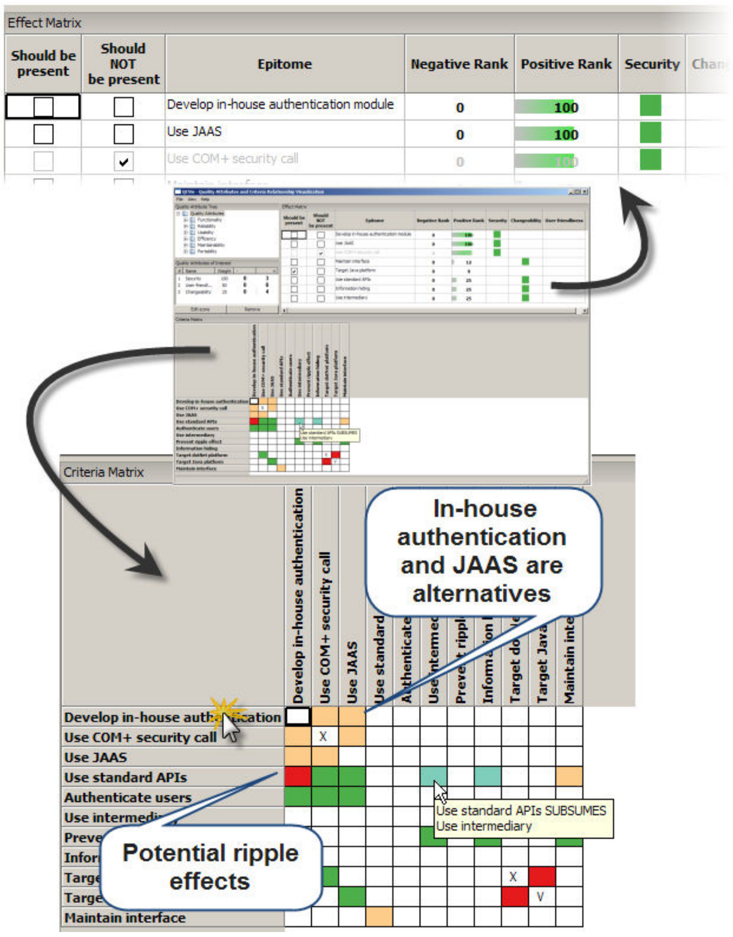
Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank
✓	☐	Target Java platform	0	0

Figure 15.9: Predefined user requirement

criterion should be used, inhibit the use of some criteria (i.e., ‘forbids’, ‘conflicts’, ‘alternative’); green cells show relations that imply the use of some criteria (‘constrains’, ‘subsumes’); blue cells show aggregation (‘subsumes’, ‘comprises’); purple denotes generic relations whose nature is not further specified (‘relatedTo’). Brushing with the mouse over the matrix cells shows tooltips with details on the relations. For example, the mouse over the cell (USE STANDARD APIS, USE INTERMEDIARY) in Fig. 15.10(a) shows that “USE STANDARD APIS subsumes USE INTERMEDIARY”.

Often, users want to focus on the quality criteria that are most relevant from the perspective of a chosen criterion of interest. We support this easily as follows. Clicking on a row or column label in the criteria matrix sorts the rows of this matrix so that the criterion of interest, corresponding to the clicked row or column, is placed first (at top) and the criteria that have a direct relation with that criterion are placed immediately thereafter. For example, Fig. 15.10(a) shows the criteria matrix sorted on the criterion DEVELOP IN-HOUSE.

Using the criteria matrix, the auditor observes that USE JAAS and DEVELOP IN-HOUSE are alternatives, hence only one of them can be present. Moreover, there are some conflicting (red) and enabling (green) relations from USE STANDARD APIS to both criteria. Also, USE STANDARD APIS has relations with other criteria such as INFORMATION HIDING and MAINTAIN INTERFACE, some of which have yet more relations with other criteria, as shown by the corresponding colored cells in the criteria



Effect Matrix

Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank	Security
☐	☐	Develop in-house authentication module	0	100	🟩
☐	☐	Use JAAS	0	100	🟩
☐	✓	Use COM+ security call	0	100	🟩

Criteria Matrix

Develop in-house authentication
Use COM+ security call
Use JAAS
Use standard APIs
Authenticate users
Use intermediary
Maintain interface
In-house authentication and JAAS are alternatives
Use standard APIs SUBSUMES Use intermediary
Potential ripple effects

(a) Effect matrix and criteria matrix

forbids / conflicts	🟥	alternative	🟧
constrains / bound to	🟩	enables	🟩
subsumes	🟦	comprises	🟦
relatedTo	🟪		

(b) Color scheme for criteria relations

Figure 15.10: Security-related analysis

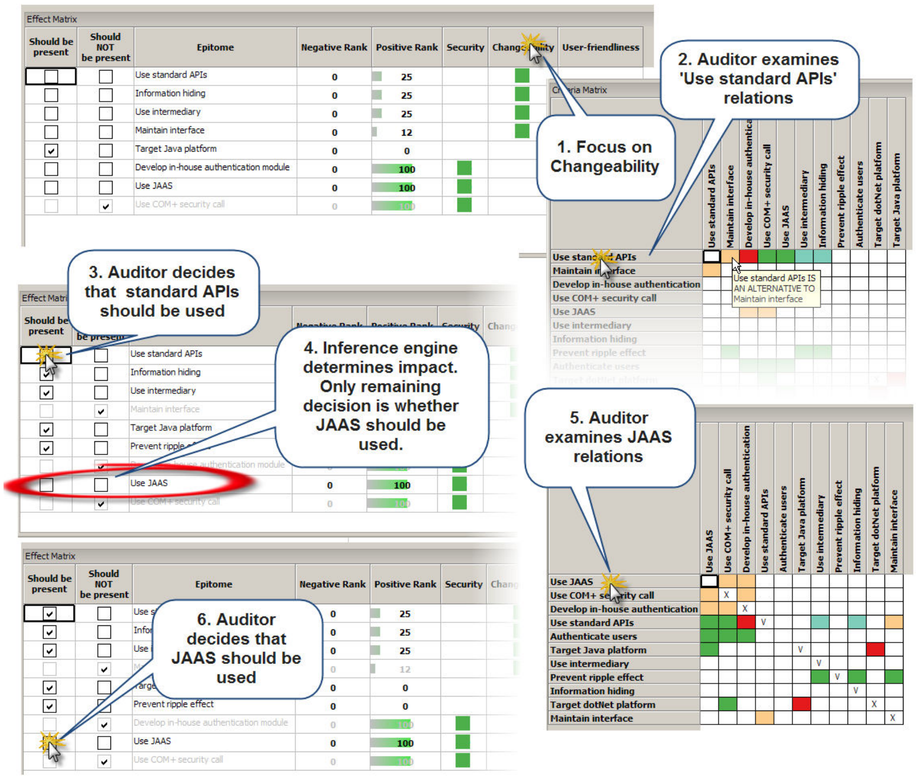
matrix in Fig. 15.10(a). This means that a decision on either USE JAAS or DEVELOP IN-HOUSE can have ripple effects on other criteria. Even though the auditor could examine the criteria matrix to trace those effects, it is difficult to see at once which of the two criteria he should expect to be present.

With the security-related alternatives at a tie, the auditor shifts his focus to the next highest-priority quality attribute. Since user-friendliness has neither positive nor negative effects associated with it, the auditor directs his attention to changeability.

Fig. 15.11 (step 1) shows the effect matrix ordered on the criteria's effect on changeability, i.e., with the four changeability-related criteria USE STANDARD APIS, INFORMATION HIDING, USE INTERMEDIARY and MAINTAIN INTERFACE at the top. From the criteria matrix in this image, we see (step 2) that USE STANDARD APIS and MAINTAIN INTERFACE (top-left corner of the matrix) are alternatives. Since the former has a higher overall positive rank than the latter (25 vs 12, as shown by the 'positive rank' column in the effect matrix in Fig. 15.11), USE STANDARD APIS is a good candidate to select as a practice required in the software product. Moreover, there are no relations between that criterion and the other changeability-related criteria INFORMATION HIDING and USE INTERMEDIARY that inhibit its selection (i.e., no red cells on the intersection of the top matrix row and the 6th and 7th columns). Hence, the auditor decides that USE STANDARD APIS should be present and checks its 'should be used' column accordingly (Fig. 15.11 step 3).

After the selection of USE STANDARD APIS, the reasoning engine automatically infers which of the other criteria should or should not be present. The result is shown in Fig. 15.11 step 4. We see that, in addition to USE COM+ SECURITY CALL, criteria MAINTAIN INTERFACE and DEVELOP IN-HOUSE are now also inferred not to be present. From the criteria matrix, the auditor can trace why: USE STANDARD APIS forbids DEVELOP IN-HOUSE, and is an alternative to MAINTAIN INTERFACE. The only choice left for the auditor is whether JAAS should be used or not. Since USE JAAS is an alternative to criteria that we now know should not be present (Fig. 15.11 step 5), the auditor determines that JAAS should indeed be used (Fig. 15.11 step 6). This concludes the selection of criteria for this audit.

Summarizing, we have seen how the criteria matrix and the inference engine aid the auditor in determining the impact of his decisions. The criteria matrix provides an overview of the relations between criteria. After the auditor takes a decision that a certain measure should be present or absent, the tool eliminates the need to take decisions that logically follow from that decision, using its inference engine. This saves the auditor time, and can also prevent taking conflicting decisions.


1. Focus on Changeability
2. Auditor examines 'Use standard APIs' relations
3. Auditor decides that standard APIs should be used
4. Inference engine determines impact. Only remaining decision is whether JAAS should be used.
5. Auditor examines JAAS relations
6. Auditor decides that JAAS should be used
Use standard APIs IS AN ALTERNATIVE TO Maintain interface

Effect Matrix

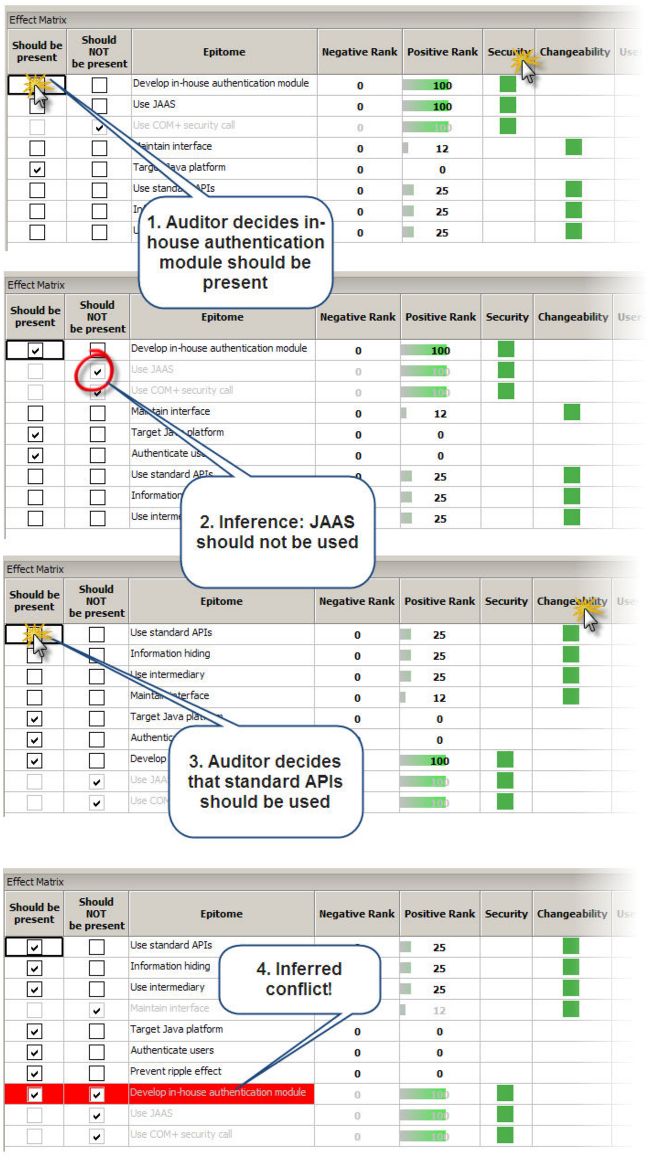
Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank	Security	Changeability	User-friendliness
☐	☐	Use standard APIs	0	25		🟩	
☐	☐	Information hiding	0	25		🟩	
☐	☐	Use intermediary	0	25		🟩	
☐	☐	Maintain interface	0	12		🟩	
✓	☐	Target Java platform	0	0			
☐	☐	Develop in-house authentication module	0	100	🟩		
☐	☐	Use JAAS	0	100	🟩		
☐	✓	Use COM+ security call	0	100	🟩		

Figure 15.11: Six-step changeability analysis

15.6.3 Scenario 3: If-then scenario

What would have happened in the previous scenario if the auditor had selected a security-related quality criterion instead of considering changeability first? With USE JAAS and DEVELOP IN-HOUSE at a tie, the auditor could choose DEVELOP IN-HOUSE without realizing that this would eventually conflict with USE STANDARD APIS.

In Fig. 15.12 (step 1), the auditor takes the (as we now know, wrong) decision to expect the presence of an in-house developed authentication module in the software product. Since there are no conflicts yet, the tool accepts the auditor's decision and infers that JAAS should not be used in the product (Fig. 15.12 step 2). Then, like in the scenario described in §15.6.2, the auditor decides that in this product standard APIs should be used (Fig. 15.12 step 3).


1. Auditor decides in-house authentication module should be present
2. Inference: JAAS should not be used
3. Auditor decides that standard APIs should be used
4. Inferred conflict!

Effect Matrix

Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank	Security	Changeability	Use
		Develop in-house authentication module	0	100	🟩		
		Use JAAS	0	100	🟩		
	✓	Use COM+ security call	0	100	🟩		
		Maintain interface	0	12		🟩	
✓		Target Java platform	0	0			
		Use standard APIs	0	25		🟩	
		In	0	25		🟩	
		U	0	25		🟩	

Effect Matrix

Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank	Security	Changeability	User
✓		Develop in-house authentication module	0	100	🟩		
	✓	Use JAAS	0	100	🟩		
	✓	Use COM+ security call	0	100	🟩		
		Maintain interface	0	12		🟩	
✓		Target Java platform	0	0			
✓		Authenticate users	0	0			
		Use standard APIs	0	25		🟩	
		Information		25		🟩	
		Use interm		25		🟩	

Effect Matrix

Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank	Security	Changeability	Use
		Use standard APIs	0	25		🟩	
		Information hiding	0	25		🟩	
		Use intermediary	0	25		🟩	
		Maintain interface	0	12		🟩	
✓		Target Java platform	0	0			
✓		Authentic		0			
✓		Develop		100	🟩		
	✓	Use JAA		100	🟩		
	✓	Use COM		100	🟩		

Effect Matrix

Should be present	Should NOT be present	Epitome	Negative Rank	Positive Rank	Security	Changeability	Use
✓		Use standard APIs		25		🟩	
✓		Information hiding		25		🟩	
✓		Use intermediary		25		🟩	
	✓	Maintain interface		12		🟩	
✓		Target Java platform	0	0			
✓		Authenticate users	0	0			
✓		Prevent ripple effect	0	0			
✓	✓	Develop in-house authentication module	0	100	🟩		
	✓	Use JAAS	0	100	🟩		
	✓	Use COM+ security call	0	100	🟩		

Figure 15.12: Conflict during security analysis

When the inference engine processes this last decision of the auditor, it finds a conflict: the use of standard APIs means that an in-house authentication module cannot be used; but the auditor explicitly specified that an in-house authentication module should be used. The tool supports the auditor in detecting and solving such conflicts by marking inconsistent criteria with a red background in the effect matrix (e.g.,DEVELOP IN-HOUSE in Fig. 15.12 step 4). The auditor can resolve this conflict manually, by deselecting that DEVELOP IN-HOUSE ‘should be used’ and setting it to ‘should not be used’, which creates the opportunity to do the opposite with USE JAAS. Alternatively, the auditor can undo the last steps up to the point where he selected the criterion that is now in an inconsistent state, and continue from there. In that case, the auditor has two remaining options: decide that JAAS should be used or consider another quality attribute first. In both cases, the end result would eventually be the same as the result in Fig. 15.11 (step 4).

This scenario shows how the auditor can perform if-then scenarios without the need to investigate and trace the intricate set of all relations. From discussions with actual auditors, we know that such scenarios are a valuable decision support mechanism, since they provide immediate feedback on the consequences of (tentative) decisions and thus save time by culling paths in the decision space.

15.7 Design Rationale of the Ontology-Driven Visualization

15.7.1 Visual design

In the design of the user interface, we followed several well-known design principles in information- and software visualization (Card et al., 1999; Chen, 2004). First and foremost, our visual design is simple. Each of our four linked views supports a user task: the quality attribute hierarchy for browsing all available attributes and selecting those of interest; the attributes-of-interest view for selecting attributes for a given analysis; the effect matrix for showing relations between criteria and attributes and criteria properties; and the criteria matrix for showing relations between criteria (Fig. 15.3). We use 2D matrix *layouts* to show relations, rather than graphs or 3D layouts. 2D matrix layouts are highly scalable and simple to use, as shown by many software visualization examples (Abello and van Ham, 2004; Diehl, 2007). Third, we use a small set of contrasting *colors*, which is effective in attracting the user’s attention to salient events, *e.g.* large positive or negative ranks (effect matrix) or conflicting relations (criteria matrix). Finally, *interaction* is simple and directly doable on all views: just a sequence of sorts and selects. Overall, the tool’s minimal design and classical GUI made it easily usable and accepted by its target group, and effectively lets users perform ‘what if’ scenarios in just a few mouse clicks. As such, our tool and its application fits in the newly emerging Visual Analytics discipline (Keim et al., 2008): instead of being a static data presentation, our tool guides and supports the user’s *decision and reasoning process*. Interaction, linked views, and continuously changing the displayed data based on the decision path are essential elements to this visual analytics design.

The auditors of DNV who assessed our tool reacted very positively. Especially the easy selection of quality criteria and the way the tool invites the user to ‘play around’ and consider ‘what if’ scenarios were cited as the tool’s main benefits (Szabó, 2008).

15.7.2 Technical design

The tool’s technical design relies on the use of semantic technologies. The ontology is implemented using the ‘Web Ontology Language’ (OWL), which is endorsed by the World Wide Web consortium and supported by various ontology editors and reasoning engines. The QuOnt ontology presented in §15.5 can be expressed in OWL quite straightforwardly.

The constraints from Table 15.1 are expressed using the Semantic Web Rule Language (SWRL), an OWL-based rule language. Like OWL, SWRL makes an ‘open world’ assumption. Briefly, this means that the absence of a statement does not necessarily mean the statement is false. Hence, in the OWL implementation of QuOnt, the absence of the statement that a criterion ‘should be used’ in an audit does not automatically imply that the criterion ‘should not be used’; it only means that it is not known yet whether the criterion should be used. This mimics the way in which auditors reason about quality criteria.

Another implication of the open world assumption is that OWL and SWRL only support monotonic reasoning; only new facts can be introduced and existing facts cannot be changed. Consequently, SWRL does not support negation ($\neg$) nor disjunction ($\vee$). Since many of the constraints in Table 15.1 use negation and/or disjunction, this poses some problems when modeling QuOnt constraints as SWRL rules.

Fortunately, many of the problems of constraint implementation can be solved. The lack of negation can be largely overcome by introducing a ‘notUsedIn’ relation for quality criteria that should not be used in an audit. With this new relation, some constraints can be rewritten to eliminate the open world assumption where appropriate. For instance, in SWRL the relation $\text{constrains}_{x,y}$ can be implemented as two rules: $\text{notUsedIn}_x \Rightarrow \text{notUsedIn}_y$ and $\text{usedIn}_y \Rightarrow \text{usedIn}_x$.

Still, two constraints cannot be implemented in SWRL: the ‘enables’ relation that implies the negation of the ‘constrains’ relation, which violates monotonic reasoning; and the ‘overrides’ relation that is a disjunction of the ‘forbids’ relation. Although this is unfortunate, we found that the 'enables' relation (which is simply defined as 'a weak form of constrains' (Kruchten, 2004)) has limited practical value. Also, given the open world assumption, the tool still allows to define criteria that at the same time 'should be' and 'should not be' used in an audit, which is in essence the goal of the 'overrides' relation in a closed world assumption.

15.8 Conclusion

In this chapter we have presented QuOnt, an ontology that supports the reuse of quality criteria in software product audits. We derived this ontology from the perspective that quality criteria can be viewed as decisions upon a *Soll*-architecture of a software product. We have demonstrated analogies between a theory of architectural design decisions and quality criteria. Based on those analogies, we have proposed our ontology, the value of which we demonstrated by walking through three scenarios of quality criteria selection using a QuOnt-based audit project memory.

For the visualization of quality criteria, we have introduced ontology-driven visualization (ODV), a type of visualization that combines the strengths of tabular and structural visualization of architectural design decisions and overcomes their drawbacks. We showed how ODV can be employed in a decision support system that assists in the reuse of quality criteria, a particular type of design decision in the early stage of a software product audit. This decision support consists of three main elements: 1. support for trade-off analysis, 2. support for impact analysis, and 3. support for if-then scenarios.

We have presented our work from the perspective of product audits and quality criteria. Given the decision characteristics of quality criteria, however, we have no reason to believe that our results are limited to this perspective alone. On the contrary, we conjecture that scenarios similar to the ones described in this chapter can also be used in a forward engineering setting. In that sense, our work shows the added value of maintaining explicit and reified relations to quality attributes, for quality criteria and design decisions alike.

We have taken the existence of ontology instances such as the ones used in the three scenarios as given. However, one of the largest challenges we see for widespread acceptance of ODV (and for the use of decision ontologies in general) is to overcome the need for complete up-front codification, especially codification of relations of which the potential amount rapidly increases when the number of decisions in the ontology rises.

A particularly interesting codification approach could be an incremental approach, in which ODV plays a role from the very beginning. Since ODV uses whatever information is codified in the ontology, it can be used even on small and/or incomplete knowledge bases. Whenever the user finds some information missing, this information can be added to the ontology and is henceforth available for use and reasoning. In this way, the codified knowledge can be incrementally extended and refined. Moreover, each refinement provides immediate benefit to the user, which provides a strong incentive to improve the knowledge base.

We see a role for data mining techniques that examine the ontology for details on past projects and derive knowledge from this historical data. For example, data mining techniques may label criteria that are often used together as 'relatedTo' each other. When these mined relations are presented to the auditor (e.g., in ODV's criteria matrix), the auditor can refine the type of relation in the ontology to one that the inference engine can reason with.

While assessment of our visualization by DNV's auditors was in general positive, more data is needed on the use of ODV in real-life situations. These situations could be the reuse of quality criteria in audits, but also reuse of design decisions in a forward engineering sense.

16
Constructing a Reading Guide for Software Product Audits

Architectural knowledge is reflected in various artifacts of a software product. In a software product audit this architectural knowledge needs to be uncovered and its effects assessed in order to evaluate the quality of the software product. A particular problem is to find and comprehend the architectural knowledge that resides in the software product documentation. In this chapter, we identify the main characteristics an architectural knowledge discovery method should exhibit. We discuss how the use of a technique called Latent Semantic Analysis can guide auditors through the documentation to the architectural knowledge they need. We validate the use of Latent Semantic Analysis for discovering architectural knowledge by comparing the resulting vector-space model with the mental model of documentation that auditors possess.

16.1 Introduction

For a given software product there is no single source that contains or provides all relevant architectural knowledge. Instead, architectural knowledge is reflected in various artifacts such as source code, data models, and documentation. In the case of a software product audit, auditors will typically first use the architectural knowledge contained in the documents to familiarize themselves with the system. Subsequently, the documentation is used as a source of evidence for findings about the software product quality. However, for large sets of documents it can be very hard to locate the knowledge needed for a specific purpose.

A complicating factor in distilling relevant architectural knowledge from software product documentation is the fact that there are often many different documents. Each of these documents is tailored to specific stakeholders and different documents can therefore reflect architectural knowledge at different levels of abstraction. For instance, a project manager might be particularly interested in the overall picture contained in a high-level document, whereas a development team needs to learn the effect of architectural decisions on the implementation from the technical specification. This means that each document has a different focus, and between the documents the levels of abstraction used may vary wildly. Furthermore, various explicit and implicit relations exist between the topics and concepts described in the documents, as well as between the documents themselves.

We have conducted a study at an organization that has broad experience in performing independent software product audits. Customers who request such audits range from large private companies to governmental institutions.

In our study we have investigated the use of architectural knowledge in software product audits. We were particularly interested in how the auditors gain the architectural knowledge needed to answer the audit question. To answer this question we observed an audit that was being conducted for one of the company's customers. We attended and observed the audit team meetings and had discussions with the audit team members on their use of architectural knowledge in the audit. In addition, we held more general interviews on this topic with five employees who had been involved in various audits, two of whom were also directly involved in the observed audit. The interviewed employees possess different levels of experience and have different focal points when conducting an audit.

Auditors have to deal with large quantities of documents that contain architectural knowledge. The auditors indicate they are in need of a reading guide that helps them discover the relevant architectural knowledge in the documentation. However, such a reading guide is often hard to obtain. We therefore need a mechanism to support (semi-)automated discovery of the important architectural knowledge in a set of documents.

In this chapter we outline the problem of discovering architectural knowledge in software product documentation and present a technique that can be used to alleviate this problem. This technique, Latent Semantic Analysis, uses a mathematical technique called Singular Value Decomposition to discover the semantic structure underlying a set of documents. We employ this latent semantic structure to guide the auditors through the documentation to the architectural knowledge needed. A comparison of the discovered semantic structure with the ideas auditors have of software product documentation shows that Latent Semantic Analysis produces a good approximation of the auditors' mental models.

The remainder of this chapter is organized as follows. The next section discusses the use of architectural knowledge in software product audits based on our observations in the study we conducted. §16.4 presents Latent Semantic Analysis (LSA) and its mathematical background, and presents a proof of concept of the use of LSA to discover architectural knowledge from written text. §16.5 discusses the application of LSA to a set of documents that contain software product documentation and shows how we can employ the semantic structure uncovered by LSA to guide the auditor to relevant architectural knowledge. In §16.6 we validate the LSA results through a comparison with auditors' mental models of software product documentation. §16.7 outlines research areas that are still open for further study, and §16.8 contains concluding remarks on this chapter.

16.2 Architectural Knowledge Use in Software Product Audits

In a software product audit, two types of architectural knowledge can be distinguished. On the one hand there is architectural knowledge pertaining to the *current state* of the software product; this knowledge reflects the architectural decisions *made*. On the other hand there is architectural knowledge pertaining to the *desired state* of the software product; this knowledge reflects the architectural decisions *demanded* (or expected). It is the auditor's job to compare the current state with the desired state.

In order to perform a comparison of current state and desired state, the auditor has to have a firm grasp on both types of architectural knowledge. A common method to structure the architectural knowledge of the desired state is to define a number of quality criteria (cf. Chapter 15). These criteria can be phrased as (architectural) decisions, and are a combination of the wishes of the customer and the expertise of the auditor. An example of such a criterion might be 'All errors in the software are written to a log. Each log entry contains enough information to determine the cause of the error.'. A software product audit consists of a comparison of these quality criteria against the current state of the software product.

The 'current state' architectural knowledge of the software product is reflected in different artifacts, in particular in source code and the accompanying documentation. Some architectural knowledge, for instance alternative solutions that were considered but have been rejected, might not be explicitly captured in these artifacts at all. This architectural knowledge is left implicit and lives only in the heads of its originators. Particular methods that are used to distill the architectural knowledge needed from these three sources - source code, documentation, and people - are scenario analysis, interviews, source code analysis, and document inspection.

Both interviews and scenario analysis are techniques to elicit (implicit) architectural knowledge from people's minds, and consequently require extensive interaction with the software product supplier. Source code analysis and document inspection, however, are performed using only the artifacts that have been delivered as part of the software product. In terms of availability of resources, the latter two are hence to be preferred. In the remainder of this chapter we will focus on document inspection in particular. A typical first use of the architectural knowledge reflected in the documentation is for auditors to familiarize themselves with a software product. Once a certain level of comprehension has been attained, the documents are used as a source of evidence for findings regarding the software product quality.

While document inspection is an important method in a software product audit, it can also be a difficult method to use. The difficulty of performing document inspection lies in the sheer amount of documentation that accompanies most software products. Auditors are swamped with documentation, and there is no single document that contains all architectural knowledge needed. Moreover, it is often not obvious which information can be found where. Auditors need to fall back on interviews, a resource-intensive technique, to gain an initial impression of the organization of architectural knowledge in the documentation.

The auditors indicate that, to locate and gain insight into architectural knowledge in documents, they need a 'reading guide'. Such a reading guide tells the auditors where to start reading, and which documents to consult when in need for an answer about a particular topic. Unfortunately, the auditors practically never encounter a reading guide among the documentation of a software product. This means that they have to fall back on interviews with for instance architects to introduce the architecture and documentation. These interviews are not always possible, for example due to time constraints.

From the above it should be clear that the auditors who perform a software product audit would greatly benefit from tools and techniques that can direct them to relevant architectural knowledge and therewith can provide a 'reading guide' for the software product documentation. We refer to the goal of such tools and techniques as 'Architectural Knowledge Discovery'.

16.3 Architectural Knowledge Discovery Requirements

Based on the observations above, this section describes what characteristics an architectural knowledge discovery method should exhibit. The requirements of such a method are aptly summarized by one of the auditors in the following quote:

"Tell me where I should start reading, which documents I can consult for

more detail about, let's say, functionality or about the architecture, or [for traces] from architecture to design. Provide me with a route through the documentation to bring me up to a certain level of knowledge."

This was the answer of an auditor when asked what the ideal reading guide should provide. An architectural knowledge discovery method that solves the problems the auditors encounter should bear such characteristics that the requirements posed in this quote are met. In other words, it should 1) tell where to start reading, 2) tell which documents to consult for more detail on a topic, and 3) provide a route through the documentation.

Since software product documentation consists for the largest part of natural language text, a technique that aims to discover knowledge contained in the documentation should be able to 'understand' natural language. A product document set typically contains information on many different topics, including high-level system architecture, functional design, logical design, and infrastructure architecture. These topics are not confined to a single document, but have relations with other topics in other documents as well. To gain a global understanding of the software product, these relationships need to be grasped by the auditor. In other words, the semantic structure of the information in the documents needs to be identified.

Discovering the semantic structure of the document set forms the core of architectural knowledge discovery. Grasping this structure transforms a set of individual texts into a collection that contains architectural knowledge elements and the intrinsic relations between them. This transformation is the foundation for further analysis outlined below.

16.3.1 Where to start reading

The semantic structure of a document set can be seen as a measure for similarity; documents that contain semantically related concepts are more similar to each other than documents that describe unrelated concepts. An architectural knowledge discovery method should exploit this intra-document similarity to group related documents together.

Documents grouped together in a cluster have certain semantic correspondence. For the auditor, this can be interpreted as an overview of which documents 'belong together'. If the topics that define the clusters are presented to the auditors, they can select documents that center around high-level concepts and read those before documents that describe concepts of a lower level of abstraction. This aids the auditor in deciding with which documents to start reading.

16.3.2 Which documents to consult

The question which documents to consult for more detail on a topic is an information retrieval question. Successful knowledge discovery should improve the success rate of information retrieval, not only by providing the auditor with insight that can be used to formulate better queries, but also by allowing the discovered semantic structure to be used in the retrieval. By taking the semantic structure into account, documents that contain text semantically similar to the query posed could be retrieved in addition to documents that contain the query terms themselves. In this sense, architectural knowledge discovery is complementary to information retrieval.

16.3.3 A route through the documentation

Architectural knowledge discovery should also employ the semantic structure to identify relationships between the different concepts themselves. Together, concepts and relationships provide a 'map' of the architectural knowledge contained in the document set. This map crosses the boundaries of individual documents, linking related topics that are not described in a single document. For example, one document may describe how certain decisions affect some of the product's quality attributes, while another document contains information on the effects of these decisions on the architectural structure of the system.

An architectural knowledge map can be used by the auditor to gain a further understanding of the architectural knowledge in the documents. The map should combine product specific concepts - such as subsystems, use cases, functional requirements, and the like - with more generic concepts, including quality attributes, architectural patterns, etc. Such a combination simplifies understanding the product, since unknown product specific terminology is linked to an idiom well-known to the auditor. A prerequisite for the success of such a combination is that the generic concepts, such as 'service oriented architecture' or 'reusability', are also contained in the documentation and are linked there to product specific concepts, such as 'subsystem X', for instance with the sentence 'To improve reusability, subsystem X implements a service oriented architecture'. The architectural knowledge map provides a route through the software product documentation by conveying which concepts from the documentation are related.

16.4 Latent Semantic Analysis

16.4.1 An overview of LSA

Latent Semantic Analysis (LSA) is a technique that infers the meaning of words and passages from natural text. The origin of LSA lies in information retrieval. LSA was presented by Deerwester et al. (1990) as ‘a new method for automatic indexing and retrieval’ of documents. Later research also focused on the psycholinguistic significance of LSA. Landauer and Dumais (1997), for instance, use LSA to simulate the acquisition of vocabulary from text, and present LSA as a theory of acquisition, induction, and representation of knowledge.

LSA starts with building up a matrix of term-document frequencies that represents a vector-space model. Vector-space models are based on the assumption that the meaning of a document can be derived from the terms that are used in that document. In a vector-space model, a document d is represented as a vector of terms $d = (t_1, t_2, ..., t_n)$, with t_i $(i = 1, 2, ..., n)$ being the number of occurrences of term i in document d (Letsche and Berry, 1997). A ‘document’ can be defined as a single document, or an arbitrary smaller unit such as a section or paragraph. The matrix elements consist of a weighted frequency count, indicating the number of times a word (the ‘term’) occurs in a document. This results in a high-dimensional sparse matrix.

With LSA, the dimensionality of the represented vector space is reduced by application of singular value decomposition. The resulting matrix is an n-dimensional reconstruction of the original matrix, with n being the reduced number of dimensions. The n-dimensional reconstruction is an approximation of the original matrix. Because of the dimensionality reduction, the new term-document frequency estimations represented by the cells in the matrix differ (sometimes significantly) from the original values. The interesting characteristic of this new matrix is that it no longer only links terms to documents in which they are present, but also to documents that are somehow related to that term. It has been argued that in this way LSA has the ability to model human conceptual knowledge (Landauer et al., 1998).

The (latent) semantic structure that is discovered with LSA can be further analyzed. Document clustering can for instance be performed by calculating the vector similarity between document vectors in the reduced LSA vector space; documents that describe similar concepts will have similar document vectors. These vectors correspond to the columns in the reconstructed matrix. Analogous to document similarities, term similarities can be calculated using the matrix rows as term vectors.

Vector similarity can also be used for information retrieval purposes. The query is then translated to a vector and compared to the document vectors in the vector space.

Because the document vectors result from LSA, a query can also retrieve documents that are related to the query, but do not contain the query terms itself. This evades the problem of synonyms in the document set, where searching for ‘reliability’ would not retrieve documents that contain information on ‘fault tolerance’.

Over the years, LSA has seen various application domains, including software engineering. For instance, Maletic et al. applied LSA to source code of software components in order to support program comprehension (Maletic and Marcus, 2000; Maletic and Valluri, 1999). Another approach is taken by Hayes et al. (2005), who use LSA to support the construction of requirements traceability matrices.

Using LSA for architectural knowledge discovery adds a new item to the list of LSA applications in software engineering. Although architectural knowledge discovery also contributes to better program understanding, our approach differs from the use of LSA by Maletic et al., who apply LSA to source code artifacts. Furthermore, as we shall see the effect of an LSA-based reading guide is not limited to better product comprehension, but further supports auditors in locating evidence for their findings. Since architectural knowledge can be reflected differently in source code and documentation, some of the evidence and knowledge that can be found in the documentation might not be available from the software product’s source code.

16.4.2 Proof of concept

Fig. 16.1 depicts a matrix based on the vector-space model constructed for three texts that were taken from the documentation of a software product. This software product, based on a service oriented architecture, was subject to an audit we observed.

The three texts used are representative selections from a use case definition (UC), a service specification (SVC), and an architecture description (ARCH). From the use case definition we took the introduction of the use case; from the service specification we took the description of the part of the service that relates to the selected use case; and from the architecture description we took the section that describes the conformance to SOA. Together, the three document vectors corresponding to these three texts contain approximately 90 distinct terms, excluding stopwords. This so-called term-document frequency matrix represents the number of occurrences of each of these terms in each of the three documents. The original document vectors are hence extended with terms that did not occur in the document itself, but do occur in one of the other texts. In these extended document vectors t_i is set to 0 if term i does not occur in the document. The cutout shows the exact number of occurrences of six terms in the respective texts. For reasons of non-disclosure, the terms ‘domain entity’, ‘use case’, and ‘business object’ have been substituted for the product-specific terminology.

Although the vector-space model in Fig. 16.1 captures some of the semantics of

	UC	***SVC***	***ARCH***
...	...	...	...
...	...	...	...
domain entity	0	2	0
service	0	3	1
SOA	0	0	3
system	0	0	4
use case	1	0	0
business object	8	3	0
...	...	...	...

Figure 16.1: Term-document frequency matrix based on the vector-space model for three software product documentation excerpts

the three texts, parts of the underlying semantic relationships are not represented very well. Based on Fig. 16.1 we can, for instance, only conclude that in theory neither the use case definition nor the service specification has anything to do with the term 'SOA' (an abbreviation for 'Service Oriented Architecture'). In practice, however, we would expect at least some relevance of the term 'SOA' in the context of a *service* specification. Latent Semantic Analysis allows us to find and exploit such underlying, or latent, semantic relationships.

As discussed in §16.4.1, LSA relies on a mathematical technique called Singular Value Decomposition (SVD). SVD decomposes a rectangular m-by-n matrix A into the product of three other matrices: $A = U\Sigma V^T$. The matrix Σ is a r-by-r diagonal matrix, in which the diagonal entries $(\sigma_1, \sigma_2, ..., \sigma_r)$ are singular values and r is the rank of A. As explained in (Deerwester et al., 1990), SVD is closely related to standard eigenvalue-eigenvector decomposition of a square symmetric matrix. In fact, U is the matrix of eigenvectors of the square symmetric matrix AA^T, while V is the matrix of eigenvectors of A^TA. Σ^2 is the matrix of eigenvalues for both AA^T and A^TA. The interested reader can find more technical details on SVD in advanced linear algebra literature such as (Golub and Loan, 1996).

Since SVD can be applied to any rectangular matrix, it can also be used to decompose a term-document frequency matrix such as the one depicted in Fig. 16.1. After such a decomposition, depicted in Fig. 16.2, the matrices U and V contain vectors that specify the locations of the terms and documents in a term-document space, respectively. The r orthogonal dimensions in this space can be interpreted as representations of r abstract concepts (cf. (Landauer et al., 1998)). The left-singular and right-singular vectors u_i and v_j indicate how much of each of these abstract concepts is present in

$$\mathbf{A} = \begin{array}{l|ccc} & \textit{UC} & \textit{SVC} & \textit{ARCH} \\ \hline \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ \text{domain entity} & 0 & 2 & 0 \\ \text{service} & 0 & 3 & 1 \\ \text{SOA} & 0 & 0 & 3 \\ \text{system} & 0 & 0 & 4 \\ \text{use case} & 1 & 0 & 0 \\ \text{business object} & 8 & 3 & 0 \\ \ldots & \ldots & \ldots & \ldots \end{array}$$

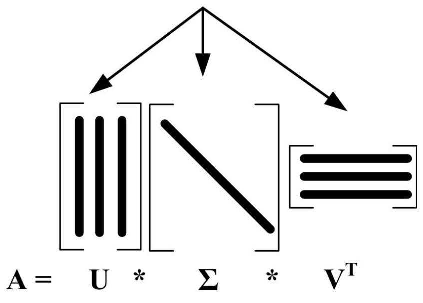
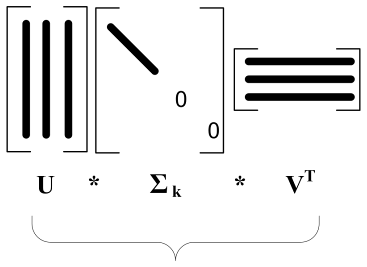


Figure 16.2: Singular value decomposition of a term-document frequency matrix

term i and document j.

As outlined above, the original matrix A can be reconstructed by calculating the product of $U\Sigma V^T$. Instead of a reconstruction, a rank-k *approximation* of A can be calculated by setting all but the highest k singular values in Σ to 0. This approximation, A_k, is the closest rank-k approximation to A (Berry et al., 1995). Calculating A_k for a term-document space, such as the one depicted in Fig. 16.1, results in the closest k-dimensional approximation to the original term-document space (Letsche and Berry, 1997). In other words, by using SVD it is possible to reduce the number of dimensions in a term-document space. It is exactly this capability of SVD, depicted in Fig. 16.3, that is employed by LSA.

By using only k dimensions to reconstruct a term-document space, LSA no longer recalculates the exact number of occurrences of terms in documents. Instead, LSA estimates the number of occurrences based on the dimensions that have been retained. The result is that terms that originally did not appear in a document might now be estimated to appear, and that other words that did appear in a document might now have a lower estimated frequency (Landauer et al., 1998). This is the way in which LSA infers the latent semantic structure underlying the term-document space, and the way in which the deficiencies in the semantics captured in a vector-space model are overcome.



$$A_k = U * \Sigma_k * V^T$$

$A_k =$

	UC	*SVC*	*ARCH*
…	…	…	…
…	…	…	…
domain entity	0.905	0.827	0.39
service	1.207	1.436	1.52
SOA	-0.451	0.585	2.806
system	-0.601	0.779	3.741
use case	0.651	0.452	-0.15
business object	6.566	4.859	-0.618
…	…	…	…

Figure 16.3: Calculation of the closest rank-k approximation to the original term-document space

In the reduced dimensional reconstruction of the term-document space, the meaning of individual words is inferred from the context in which they occur. This means that LSA largely avoids problems of synonymy, for instance introduced because two different authors of documentation for the same software product use two different terms to denote the same concept. One of the authors might for instance use the full product name in the documentation, while the other author prefers to use an acronym. Since the contexts in which these different terms are used will often be similar, LSA will expect the product acronym to occur with relatively high frequency in texts where the full product name is used and vice versa. However, it should probably be stressed here that we cannot expect LSA to improve the documentation other than making it more accessible. LSA will happily accept wrong, superfluous, or obsolesced documentation and guide anyone interested to ‘relevant’ parts of that documentation. Nonetheless, for reasonably well-written documentation the latent semantic structure LSA infers can be very well exploited to guide the reader.

Fig. 16.4 shows the result of the application of LSA to the term-document frequency matrix from Fig. 16.1. The cutout shows the same six terms that are shown in the cutout in Fig. 16.1, but this time the numbers correspond to the *estimated* term frequencies based on retaining only 2 dimensions. Upon inspection of this result, in-

	UC	***SVC***	***ARCH***
...	...	...	...
...	...	...	...
domain entity	0.905	0.827	0.39
service	1.207	1.436	1.52
SOA	-0.451	0.585	2.806
system	-0.601	0.779	3.741
use case	0.651	0.452	-0.15
business object	6.566	4.859	-0.618
...	...	...	...

Figure 16.4: Estimated term-document frequencies after the application of LSA to the matrix in Fig. 16.1

teresting patterns appear. For starters, the term SOA is now expected to be present in the service specification as well, albeit at a lower frequency than in the architecture description. This corresponds to our intuitive notion that we would expect at least some relevance of SOA to a service specification. The negative expected frequency of SOA in the use case specification is somewhat awkward to interpret mathematically, but might perhaps best be regarded as a kind of 'surprise factor'. In a sense, LSA tells us not only that it does not expect the term SOA to crop up in the use case specification (estimated number of occurrences = 0), but that indeed it would be quite surprised to encounter this term there.

In general, a pattern seems to emerge in Fig. 16.4. If we regard the use case specification as the lowest level of abstraction text, the architecture description as the highest level, and the service definition somewhere in between, we see that low-level concepts (such as 'business object' and 'use case') have a diminishing level of association as the level of abstraction of the text increases and vice versa. LSA also seems to indicate that the term 'service' is a central concept in the documentation: its estimated frequency is almost equal for all three documents. Those patterns stem from the semantic structure in the documents. We can employ this uncovered semantic structure to guide an auditor to the information needed.

16.5 Constructing a Reading Guide: A Case Study

The LSA technique introduced in §16.4 forms the basis of a detailed case study in which we examine how the semantic structure discovered by LSA can be employed to

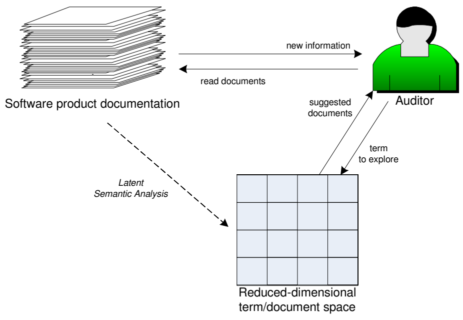


Figure 16.5: Schematic overview of the construction of a reading guide using the reduced-dimensional term-document space calculated by LSA

guide the auditors through the documentation. This section presents the results of this case study.

Fig. 16.5 depicts the interactive process by which an auditor is guided through the documentation. Initially, auditors start with a set of unread documents. Although the content of these documents is still unknown, the auditors have a goal that needs to be satisfied by reading (part of) the documentation. Examples of such goals are obtaining a global understanding of the software product, investigating certain quality attributes, or locating (further) evidence for certain findings. The reduced-dimensional term-document space, which results from LSA, can be inspected to locate documents that are highly associated with a term that corresponds to the auditor's goal. For instance - and this example will be worked out in more detail below - the term 'architecture' could be used to find documents that provide high-level information about the software product. From reading the suggested documents, an auditor learns new information including new - potentially product-specific - terms that can be used to locate documents that provide more detail on the new terms. In short, reading guidance consists of an iterative process of selecting and reading documentation, in which the auditor can use the architectural knowledge gained from reading suggested documents to steer the selection of new documents.

We applied LSA to a total of 80 documents that were subject to the audit that has been described earlier. The term-document frequency matrix that was constructed for these documents contained a total of 3290 terms found in the 80 documents. These 3290 terms did not contain very common words ('stopwords') such as 'a', 'the', or 'is'. It is common practice to disregard these stopwords, since they tend to be evenly spread over all documents and hence do not bear any distinctive meaning. The length of the document vectors that make up the term-document frequency matrix had been normalized using cosine normalization (Salton and Buckley, 1988) before LSA was applied. This normalization reduces the effect of document size (i.e., the number of terms in the document); without normalization, longer documents tend to be favored over shorter documents when a document selection is made.

Using the technique described in §16.4, we calculated a 5-dimensional approximation of the constructed term-document frequency matrix. The selection of the number of dimensions to retain is an empirical issue (Landauer et al., 1998), although some heuristics exist (Berry et al., 1999). The rank-5 approximation chosen here requires a 49% change relative to the original term-document frequency matrix. Although this might appear as a rather large change, the results obtained with this approximation suit our needs; they can be effectively used to construct a reading guide.

The case study presented here reconstructs the early phase of the software product audit, in which the auditors need to attain a global understanding of the software product in order to further assess its quality. As in the previous section, for reasons of non-disclosure the results presented here have been anonymized.

In general, when auditors commence a software product audit they want to gain an initial, high-level understanding of the software product. This global understanding is necessary to successfully perform the audit, since it is a prerequisite for subsequent audit activities. For instance, in scenario analyses the supplier of the software product is asked how the product reacts to certain change scenarios or failure scenarios. In order to judge the answer the supplier provides, an auditor needs to have a thorough understanding of the software product. Otherwise, the supplier might provide an answer that is incomplete or inconsistent with the real state of the product, without this being noticed.

Auditors who want to attain overall comprehension of the software product can be guided through the documentation using the semantic structure discovered by LSA. A route that is preferred by all auditors we interviewed is to start with high-level, global information and gradually descend to texts that contain more detailed and fine-grained information. A single term that can be expected to cover the high-level information about the software product well is the term ‘architecture’.

We can inspect the reduced 5-dimensional approximation of the original term-document frequency matrix that LSA has calculated to find the documents that best match the term ‘architecture’. In order to do so, it suffices to rank the documents by their respective values in the row that coincides with the term ‘architecture’ (the ‘architecture’ term vector). Documents that have a high value in the ‘architecture’ term vector correspond to the documents in which LSA expects the highest number of occurrences of the term ‘architecture’. Recall that the highest-ranking documents do not necessarily include the literal term ‘architecture’, but LSA inferred that it would be likely to encounter the term ‘architecture’ in these documents; they are semantically close to the meaning of ‘architecture’. In other words, these documents talk *about* architecture, perhaps without mentioning the word ‘architecture’ itself.

Table 16.1: Top-10 documents that match the term ‘architecture’, with the number of occurrences of ‘architecture’ in the document

Rank	*Document ID*	*# ‘architecture’*
1	79	44
2	39	1
3	44	3
4	41	2
5	78	10
6	46	0
7	45	1
8	42	1
9	40	2
10	49	0

The list in Table 16.1 shows the 10 highest ranked documents for the term ‘architecture’, together with the actual number of occurrences of ‘architecture’ in these documents. Given this list, an auditor can simply start reading top-down, in this case starting with document 79. However, some of these documents are fairly large while others are rather small. In fact, documents 46 and 45 both consist of only 2 pages. If an understanding of the software product can be attained by either reading a (very) small document or by ploughing through a large number of pages, the former is obviously preferred by the auditors. Table 16.2 lists the same top-10 documents for ‘architecture’ also shown in Table 16.1. In this table, however, the documents have been categorized according to their size. The size categories have been defined as: very small (< 5 pages), small (< 10 pages), medium (< 30 pages) and large (≥ 30 pages). The rank according to Table 16.1 is given in parentheses, to illustrate the differences.

Table 16.2 shows that, given a preference for smaller documents, an auditor looking for information about the architecture of the software product should first read document 46. Note that this document does not contain the term ‘architecture’ at all (see Table 16.1). Nevertheless, upon inspection this document indeed contains high-level ‘architectural’ information.

From document 46, the auditor learns for instance that the software product consists of three high-level components, which we will call X, Y, and Z. Furthermore, the document identifies two external systems that interact with the software product as

Table 16.2: Top-10 documents that match the term ‘architecture’, grouped by size

	Rank		*Doc. ID*	*# pages*
Very small	1	(6)	46	2
	2	(7)	45	2
Small	3	(5)	78	6
	4	(10)	49	9
Medium	5	(1)	79	24
	6	(2)	39	13
	7	(3)	44	21
Large	8	(4)	41	30
	9	(8)	42	48
	10	(9)	40	31

well as an organizational unit that will handle certain types of operational problems that might occur. Finally, the document contains a list of intended uses of the software product.

Now that the auditor knows a little more about the software product, the next document has to be selected. Since the auditor still has not read all ‘architecture’ documents, there are in principle two options: either remain with the ‘architecture’ documents and select a document from that list (e.g., document 45) or use the architectural knowledge obtained to delve into a particular topic.

Good candidates for further exploration of the documentation are the components X, Y, and Z. Since these components conceptually divide the software product in three distinct parts, auditors might want to examine each of these parts to further their global understanding. In its current form, the selection of the right terms for exploration is a matter of experience. It is from experience that the auditor knows that the term ‘architecture’ is likely to be related to high-level software product documentation. It is from experience that the auditor suspects that the three components are good candidates for further exploration.

Given the fact that the auditors want to gradually progress through the documentation, the degree of deviation of each of the components from the meaning of the term ‘architecture’ is an indication of the route that should be followed through the documents. In order to assess the deviation, a calculation can be performed of the similarity between the terms ‘architecture’ and ‘componentX’, ‘componentY’, and ‘componentZ’ respectively. This enables us to identify how much the texts for which LSA infers a high association with each of the components deviate from the text in document 46, which closely resembled the meaning of the term ‘architecture’.

A common measure of similarity between terms (or 'term-term similarity') is the cosine of the angle between the two term vectors (Berry et al., 1999). Let t_i and t_j be the term vectors for terms 'i' and 'j' respectively, i.e., the rows from A_k that correspond to the terms 'i' and 'j'. Then the cosine of the angle θ between these term vectors is $\cos\theta = \frac{t_i \cdot t_j}{||t_i|| ||t_j||}$.

Table 16.3 shows the similarity of each of the terms 'componentX', 'componentY', and 'componentZ' with the term 'architecture', calculated as the cosine between their respective term vectors. It becomes clear from these three values that 'componentX' is semantically closest to 'architecture' followed by 'componentZ', and that 'componentY' is the least similar to 'architecture'.

An interesting observation is that the relations between the three components are not readily apparent from the text in the document itself, nor from the names given to the components. Although the document does contain a picture that seems to suggest a layered ordering of the components, the text in document 46 does not mention or reflect such a layered approach at all. Here, by using LSA we have truly discovered architectural knowledge that the auditor could not have distilled from reading document 46 alone. This discovered knowledge can be used to further explore the documentation.

Based on the similarity of 'architecture' and each of the three components, a logical next step to read the documentation seems to first read the top-ranking documents for 'componentX', then for 'componentZ', and finally for 'componentY'. By following this route, the semantic distance between the document just read and the newly selected documents increases gradually.

Table 16.3: Cosine term-term similarity of 'architecture' and the high-level components

componentX	*componentY*	*componentZ*
0.9814	0.4096	0.7900

Analogous to the selection of the top-10 documents for the term 'architecture', we selected the top-10 documents for each of the terms 'componentX', 'componentY', and 'componentZ'. For each of the components, we analyzed the top-10 documents found. The results of this analysis show an interesting pattern in the selection of documents.

Due to its close semantic resemblance of 'architecture', shown in Table 16.3, the top-10 documents that were found for the term 'componentX' are in fact the same 10 documents that were found for 'architecture'. The only difference is a small change in the ranking of the documents. The top-10 documents found for the term 'componentZ' (the next closest term to 'architecture') comprises a mix of service specifications and architectural design descriptions, with a clear focus on service specifications (the first four documents in the list are service specifications). The top-10 documents found for ‘componentY’ are all use case definitions.

The route through the documentation found by analyzing the result of LSA suggests that, using ‘architecture’ as a starting point, the auditors should first read the architecture descriptions and similar high-level documentation, then proceed with service specifications, and finally read the use case definitions. Although LSA does not in and of itself know of the distinction between high-level documents (i.e., architecture descriptions) and low-level documents (i.e., use case definitions), the documents that it suggests to read are grouped along this axis. Moreover, from interviews with the auditors we learned that this is indeed a route they prefer to follow to familiarize themselves with a software product.

16.6 Validation of the Use of LSA

The previous section shows how the application of LSA delivers results that support auditors in finding a route through the documentation. The auditors indicate that the results show correspondence to their preferences for selecting and reading documents. In this section we empirically validate this correspondence.

The knowledge discovered by using LSA can only be regarded valid if it fits the expectations of the auditor. In other words, the discovered semantic structure must conform to the auditor’s mental model of software product documentation and the architectural knowledge contained within. This means that the validity of the result is in principle highly subjective.

Although an auditor’s mental model is subjective, the auditors who were interviewed in the light of our case study on architectural knowledge use appear to have commonalities in their mental model of software product documentation. Perhaps the most obvious commonality is the categorization of documentation according to its level of abstraction. However, when confronted with the question how they find the architectural knowledge needed without having a reading guide available, most auditors reply that it is largely a matter of experience. This means that, even if we can employ the commonalities in the auditors’ perception of software product audits, we still need to validate our results against tacit knowledge.

A method that can be used to elicit tacit knowledge is the repertory grid technique. This technique stems from personal construct theory, and was devised by Kelly (1955). The repertory grid technique can be used for exploring so-called ‘personal construct systems’; the collection of implicit personal theories. The repertory grid technique is “an attempt to stand in others’ shoes, to see their world as they see it, to understand their situation, their concerns” (Fransella and Bannister, 1977).

The repertory grid technique investigates bipolar constructs that form someones mental model of (part of) the world. Examples of such constructs are *past-future*, *good-bad*, and *everything-nothing*. A particular method to elicit these constructs is by presenting a person with triads of elements from the domain under investigation. In our case, we could regard the individual documents as elements. When three of these elements are presented, the person is asked in which way two of them are alike and how the third one is different. This identifies a bipolar construct or axis that is apparently part of the person's mental model.

Based on elicited constructs, a so-called rank order grid can be constructed. To do so, the person is asked to rank all elements according to how well they fit the construct poles (e.g., from 'past' to 'future'). This rank order grid can then be further analyzed, for instance to determine the distances between different elements or constructs (Jankowicz and Thomas, 1982). In our case, we derive the distance matrix for the distances between documents as perceived by an auditor.

Ideally, we would apply the repertory grid technique to the same documents that were used in the case study described in §16.5. Unfortunately, using 80 elements in a repertory grid experiment is infeasible. Because of the exponential growth of possible combinations (triads) of elements, the upper limit for the application of the repertory grid technique is somewhere around 20 elements. For a validation of the LSA results, we therefore selected a second audit project in which a much smaller number of documents were involved.

We understand that it is dangerous to attribute the validity of one project to the validation of another, especially so if the projects differ in size. Nevertheless, since auditors already acknowledge that the LSA results are sensible we argue that validation within a small project at the very least significantly adds to the credibility of our application of LSA in larger projects. This is even more so because LSA results are generally thought to improve for larger corpora (Landauer et al., 1998).

For the audit project in which we applied the repertory grid technique, the number of documents that had to be assessed was limited to 10. The audit team consisted of three members: one project manager and two auditors. The repertory grid technique was used in two experiments, one for each auditor. Each experiment lasted for approximately two hours, until the auditors felt they could not think of any more sensible (and distinctive) constructs. At the time we conducted the experiment, the audit project itself had been finished approximately two months earlier. To prevent distortions of the experiment results because of the two month gap between audit and experiment – part of the mental model might have been forgotten in the meantime – the triads presented to the auditors consisted of the actual (physical) documents, which the auditors could freely browse and read during the experiment. Two exceptions were documents AS and DB (see below), which were no longer physically or electronically available.

Those two documents were represented by a proxy: a single sheet of paper with the document's title.

The 10 documents will be referred to by the following abbreviations:

- FM contains a functional data model.
- FD describes the functional design.
- XX is an addendum to FD. This is not immediately clear from the title of the document, nor from the document's layout.
- PD contains the process design.
- TP contains a test plan.
- UM is the user manual.
- RN is a set of release notes.
- IM is the installation manual.
- AS is an administration manual for the application server.
- DB is an administration manual for the database server.

Tables 16.4 and 16.5 depict the rank order grids for auditors 1 and 2 respectively. The auditors were asked to rank the documents on a 1 to 5 scale. Hence, the first row in Table 16.4 should be read as follows: auditor 1 considers the contents of documents FM, FD, UM, and DB to be invariable over releases, whereas RN varies per release. PD, TP, and AS are considered to be more invariable than variable (but not completely invariable), while the opposite is true for IM. XX is exactly in the middle. The constructs should be interpreted from the auditor's point of view in the context of the performed audit. In other words, the construct 'used by me'/'not used by me' in Table 16.4 means '(not) used by auditor 1 in the audit'.

While many things could be said about the constructs that were elicited from both auditors – for instance regarding the commonalities and differences between the two grids – we will not perform such an analysis here. Within the grids themselves, clearly not all constructs (or dimensions) are orthogonal. This, too, does not matter for the discussion at hand. It suffices to take the two grids simply for what they are: a representation of the auditor's own mental model of the 10 documents used in the audit.

We analyzed the auditors' rank order grids to calculate the distances between the documents as perceived by the auditors. The resulting distance matrices can be further

Table 16.4: Rank order grid auditor 1

1	*XX*	*PD*	*FM*	*FD*	*TP*	*UM*	*RN*	*IM*	*AS*	*DB*	5
invariable	3	2	1	1	2	1	5	4	2	1	varies per release
used by me	5	1	1	1	2	4	5	5	3	3	not used by me
input for development	1	1	1	1	3	5	5	5	4	4	output of development
prescriptive	1	1	1	1	3	5	5	4	5	5	descriptive
development	1	1	1	1	2	2	3	4	5	5	deployment
functionality	2	2	1	1	2	2	4	5	5	5	no functionality
diagrams expected	3	2	1	1	5	2	5	5	5	5	no diagrams expected
whole application	5	1	1	1	1	1	1	5	5	5	part of application
use	4	2	2	2	3	1	3	5	5	5	deployment
test	1	2	3	2	1	1	1	5	5	5	deployment
data flow	1	1	5	3	1	2	3	3	3	4	data entity
too global	5	5	5	4	1	2	3	5	3	3	too detailed
good size	5	5	4	1	1	1	3	1	3	3	overwhelming

Table 16.5: Rank order grid auditor 2

1	*XX*	*PD*	*FD*	*FM*	*TP*	*RN*	*UM*	*IM*	*DB*	*AS*	5
abstract	2	1	1	2	4	5	5	5	5	5	concrete
content	1	1	1	1	3	5	5	5	4	4	packing
input for development	1	1	1	1	3	4	5	5	4	4	output of development
descriptive / static	1	1	1	1	3	3	5	5	4	4	use / time dimension
app. functionality	1	1	1	1	3	3	3	4	5	5	system administration
design	1	1	1	1	3	4	3	5	5	5	deployment
conceptual	1	1	1	2	2	4	3	4	5	5	concrete/instance
high level	5	3	1	1	2	5	3	4	4	4	detailed
absolute	1	1	1	1	3	5	2	2	2	2	relative (wrt prev. version)
application	1	1	1	1	5	2	2	2	2	2	organisation
used by me	4	2	1	2	5	2	1	1	3	3	not used by me

analyzed, for instance to determine clusters of documents that contain similar documents.

Figs. 16.6 and 16.7 depict the document clusters according to auditor 1 and 2 respectively. The clusters have been determined with the single linkage hierarchical clustering method (Johnson, 1967). The axis denotes the difference between the documents, calculated as 1 minus the similarity. For instance, for auditor 1 the similarity between documents FD and FM has been calculated as 0.87, therefore the difference between the two equals 0.13, as shown in Fig. 16.6.

Although there are some differences between the two cluster configurations, both auditors seem to agree that there are two large document clusters. One cluster contains documents FD, FM, PD, and XX. The other documents are grouped in the second cluster. Note that we left AS and DB out of the figures to allow for a fair comparison with the effect of LSA. Recall that those two documents were no longer available, due to which LSA was unable to process them. Had we included them in the cluster figures, they would have shown as a small sub-cluster of two very similar documents (similarity according to auditor 1: 0.96; auditor 2: 1.00). For both auditors this sub-cluster is most similar to document IM (auditor 1: 0.79, auditor 2: 0.84).

To illustrate the effect of LSA on the document vector-space model, we applied LSA to the 8 documents from the audit that were still available. We determined the

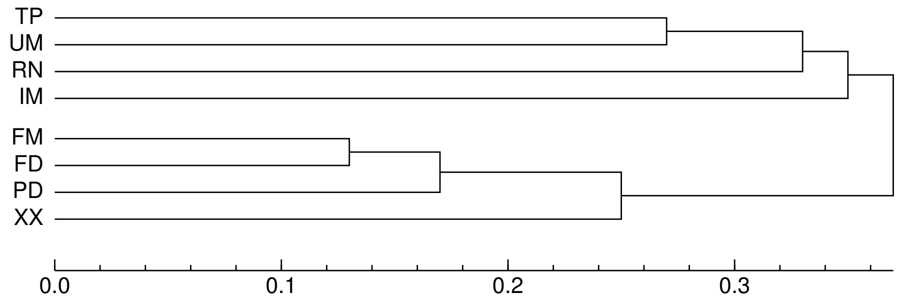


Figure 16.6: Auditor 1: Hierarchical documentation clusters

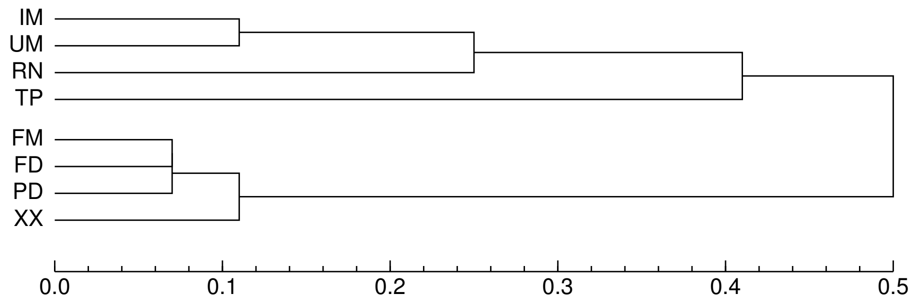


Figure 16.7: Auditor 2: Hierarchical documentation clusters

distance matrices for both the term-document frequency matrix and the LSA result. The distance measure used to calculate distances between two documents is the cosine of the angle between the two corresponding document vectors.

Fig. 16.8 depicts the clusters for the original term-document frequency matrix. Those clusters already show some correspondence to the two clusters which the auditors perceive. However, documents XX and RN are two clear outliers. Moreover, the distinction between the two clusters (the cut around 0.5 similarity) is not very obvious.

The distinction between the two clusters is much clearer in Fig. 16.9. This figure shows the clusters after LSA has been applied. The only outlier remaining is the set of release notes (RN). A possible (but unverified) explanation might be that the release notes follow a style different from the other documents. The release notes consist mainly of a list of short, somewhat stenographic messages that describe the changes from one version to another while the other documents contain more extensive text.

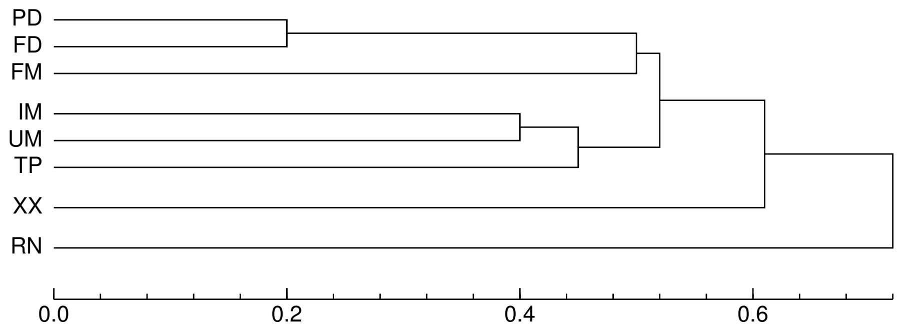


Figure 16.8: Term-frequency: Hierarchical documentation clusters

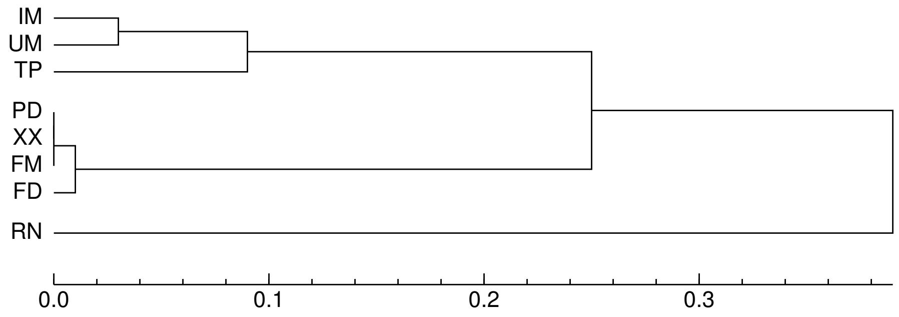


Figure 16.9: LSA: Hierarchical documentation clusters

The hierarchical clusters leave the impression that the application of LSA transforms the vector space model (initially represented by the term-document frequency matrix) to better resemble the auditors' mental models. A quantitative expression of this 'better resemblance' can be given by calculating the correlation between the various distance matrices.

To calculate the correlation between the distance matrices we have performed a simple Mantel test using the zt software tool (Bonnet and Peer, 2002). In this test, the null hypothesis is that the distances in one distance matrix are independent from those in another. The results of this test are shown in Table 16.6.

Table 16.6: Simple Mantel test r and p-values

	Auditor 1	*Auditor 2*
Term-document frequency matrix	r = 0.627, p = 0.00184	r = 0.666, p = 0.00017
LSA	r = 0.730, p = 0.00179	r = 0.871, p = 0.00057

The results of the simple Mantel test show that there is already a significant correlation between the distances according to the term-document frequency matrix and the auditors' mental models. This corresponds with how the two clusters already appear in Fig. 16.8. However, the correlation between the auditors' rank order grids and the result of LSA are clearly higher. As a matter of fact, the lowest correlation with LSA ($r = 0.730$ for auditor 1) is comparable to the correlation between the rank order grids of the auditors themselves. A simple Mantel test shows that the correlation between the two auditors' rank order grids is 0.738 ($p = 0.00020$).

In summary, the correlation coefficients calculated by the simple Mantel test indicate that application of LSA yields a vector space model more similar to an auditor's mental model than merely counting the words in the documents. Visually, this shows as a better correspondence of document clusters with the clusters according to the auditors' perception. This empirical validation supports our initial finding that the reading guide constructed by applying LSA intuitively made sense for the auditors.

One thing that should again be stressed here, however, is that the result of the application of LSA is very much dependent on the selected number of dimensions to reduce to. Unfortunately we currently have no better guideline than trial-and-error heuristics. Since there was only a small number of documents used in this particular case, we were able to calculate the results for all possible number of reduced dimensions – the original term-document frequency matrix has a maximum number of orthogonal dimensions equal to the number of documents. This allowed us to select the number of dimensions – 4, for the record – that looked most promising. When larger numbers of documents are involved, this approach becomes infeasible quite soon.

16.7 Future Work

The work presented in this chapter gives rise to a number of issues that warrant further research. An overall issue that remains to be investigated is the scalability of our approach. LSA proved to be feasible for a corpus of 80 documents, but in practice software product documentation might comprise many more documents. Document sets of several hundreds of documents are not uncommon.

Furthermore, the selection of the right number of reduced dimensions is still difficult. In this area, a comparison of the auditor's mental model with the result of LSA – as shown in §16.6 – could possibly provide guidance to selection of the right number of reduced dimensions, since the optimal number of dimensions yields the best match of the LSA result with the auditor's perception. This of course implies that the auditor's mental model is known. Although determining this model is feasible by means of the repertory grid technique, this is by no means a trivial task to perform. Especially not if this has to be done for every new project in which LSA is to be used.

Besides these global issues, we have identified three main areas that are to be further explored in the present context: enhancement of the workflow, the use of background knowledge, and user interaction. This section describes each of these areas in more detail.

16.7.1 Workflow enhancement

The 'workflow' presented in this chapter, i.e., the selection of terms to explore the documentation, is still rather ad hoc and depends heavily on the auditor's experience. One could wonder whether the same result would have been obtained had the process been started with another 'high-level' term, such as the name of the software product instead of the generic term 'architecture'.

As a matter of fact, using the name of the product, or the high-level term 'system' instead of 'architecture', would have yielded a different result. The documents that are suggested for these terms resemble the documents that were suggested for 'componentZ', i.e., with an emphasis on service specifications. Document 46 would not have been suggested for any of these terms. Depending on one's opinion this may or may not come as a surprise. Some might argue that 'system' indeed carries more of a notion of implementation than 'architecture'. It does show, however, the importance of the selection of the (initial) terms to explore. It also shows that the auditor would benefit from assistance in this selection instead of having to rely on experience alone. We would like to capture this kind of experience to enhance the workflow and aid the auditor in selecting new terms to explore.

We believe that we can capture relevant experience and enhance the workflow by introducing a feedback loop. By keeping track of terms that worked well in earlier projects, the auditor can be presented with suggestions as to which terms to explore in a new project. This helps the auditor to get the project started (e.g., by suggesting initial terms such as ‘architecture’), but can also circumvent potential dead-ends in the exploration by explicitly ignoring terms that are known to have led to dead-ends in previous audits. Such suggestions could perhaps also be mined from the documentation itself, using techniques such as frequency profiling to locate uncommon words (with respect to a standard corpus) that are likely to be part of a domain-specific vocabulary. Sawyer et al. (2005) report successful application of frequency profiling in extraction of domain terms from requirements engineering documents.

While a feedback loop could relatively easily handle common generic terms such as ‘architecture’, additional research is needed in particular to determine how to cope with product-specific terminology such as ‘componentX’. The reasons that auditors choose certain (product-specific) terms for further exploration need to be analyzed and translated to more generic ‘heuristics’ and best practices that are applicable to other projects as well. An example heuristic might be that, given the auditor’s goal of overall product comprehension, terms that signify components are better candidates for further exploration than terms that signify intended uses. This heuristic can change when the auditor’s goal changes. If the auditor is looking for the satisfaction of certain requirements, intended uses might be preferred over components.

16.7.2 Background knowledge incorporation

The ‘queries’ that are used in this chapter to explore the software product documentation are logical from an auditor’s point of view. The auditor starts with a high-level exploration of the software product’s architecture, gradually zooming in to reveal more detailed architectural knowledge. Through Latent Semantic Analysis, documents with diminishing levels of abstraction are identified: from architecture descriptions at the highest level through service definitions to use case specifications at the lowest level. However, this analysis sequence still requires extensive human interpretation. As remarked earlier, LSA itself has no notion of ‘high-level’ or ‘low-level’ documentation, nor of any other domain-specific knowledge.

To further enhance the support for auditors reading the software product documentation we intend to investigate the incorporation of relatively static background knowledge in the automated analysis of the documentation. The words ‘relatively static’ signify domain knowledge that does not change for each audit. Apart from often used classifications such as high-level vs. low-level documentation, examples of such background knowledge are:

- generic models, such as quality models (e.g., ISO/IEC 9126-1) and process models (e.g., ISO/IEC 14598-5);

- ontologies, for instance of architectural patterns (e.g., the Handbook of Software Architecture by Booch (online)) and their known relations with for example quality attributes, generic software engineering ontologies (e.g., the ontology by the SEONTOLOGY project team (http://www.seontology.org/)) and application-generic software architecture ontologies (e.g., (Babu T et al., 2007));

- 'heuristics', such as an auditor's general preference for smaller documents (see also §16.5).

Background knowledge can be incorporated when constructing a reading guide by using it in the selection of suggested documents to read. Models and ontologies can for instance be used to broaden the scope when exploring a certain term; they can be used to formulate rules of the form 'if auditors are interested in X (e.g., 'maintainability') they are (probably) also interested in Y (e.g., 'changeability', see also ISO/IEC 9126-1)'. Heuristics can for example be applied to change the suggested order in which the documents should be read, as demonstrated in §16.5.

Note that there is also some overlap of background knowledge incorporation with the workflow enhancements described in §16.7.1. The heuristics (or best practices) described in that section can in fact be regarded as background knowledge. The translation of product-specific terminology to generic terms used in these heuristics could very well be based on an ontology structure.

Background knowledge can hence be employed at two levels: to suggest terms to explore, steering the workflow; and to suggest documents to read, steering the analysis. Both affect the outcome of the process, the reading guide.

Using the correlation coefficient calculated by the simple Mantel test, we can closely monitor whether adjustment of our method improves the result. We could, for instance, assess the effect of the incorporation of background knowledge in the analysis of the documentation. Such an assessment consists of a comparison of the distances perceived by the auditors with the distances before and after incorporating background knowledge. If the result of the analysis corresponds more to the auditor's mental model when background knowledge is taken into account, this means that using this background knowledge is indeed an improvement. Finally, the effect of using other techniques instead of, or together with, LSA could be easily judged analogous to the assessment of the incorporation of background knowledge. Techniques that could further improve architectural knowledge discovery results include techniques that, complementary to LSA, exploit certain more structured properties of the documentation. If, for example, a set of documents is structured according to a particular template – which is not uncommon – knowledge of this template could be used to guide the reader to particular parts of the documentation.

16.7.3 User interaction and tool support

A final area in which further research is needed is the area of user interaction. The results presented in this chapter all show direct operations on the reduced-rank approximation of the original term-document frequency matrix. This matrix is probably not the best form of presentation for the end users, i.e., the auditors.

In order to be useful and used in an auditor's everyday practice, the techniques discussed in this chapter should be implemented in an interactive environment that abstracts away from the underlying estimated term frequencies. This environment should provide intuitive support for the workflow discussed in §16.7.1.

A particular area that requires further research is visualization of the reduced-dimensional term-document space. A desirable visualization supports both locating terms to explore and locating documents to read. Ideally, this would be presented to the auditors as a space through which they could navigate in search of the architectural knowledge they need. In this space, distances between terms and/or documents have actual meaning (cf. the three terms in Table 16.3). Such a visualization requires a projection of the reduced-dimensional term-document space to two - or at most three - dimensions. Through such a visualization, auditors can obtain quick visual clues as to which documents are closely related and how to proceed reading the document set.

16.8 Conclusion

Document inspection is a method used in software product audits to distill architectural knowledge from the software product documentation. Unfortunately, document inspection is often hard to perform. Auditors are in need of a reading guide that tells them where to start reading, how to progress reading, and which documents to consult for more detail on a particular topic.

We have demonstrated how auditors can be guided through the documentation in a case study in which we reconstructed the early phase of a software product audit. In this phase, the auditor has not read any documents yet and needs to attain a certain level of understanding of the software product.

The issue the auditors from our case study encounter can be extended to a more general public. All stakeholders that need to familiarize themselves with a software product for whatever reason will have to answer the same question: where in the documentation do I find the architectural knowledge I need? This is especially true when, unlike for some software product audits, no one is available to introduce the architecture to the stakeholder. The stakeholder then has to rely solely on the documentation that accompanies the product. A particular situation in which this can be the case is when a software product has to undergo maintenance. If the original architects are no longer available, the maintenance team only has the documentation at its disposal. We argue that a solution to this problem of insight lies in architectural knowledge discovery.

To construct a reading guide, we have employed the semantic structure discovered by Latent Semantic Analysis. This semantic structure is used as the basis for an interactive process in which auditors indicate terms that they want to explore and are subsequently given reading suggestions for documents containing information about these terms. The knowledge obtained from the suggested documents can give rise to new terms to explore, and the discovered semantic structure can be used to determine the order in which the terms - and corresponding documents - should be explored.

Using the repertory grid technique, we were able to elicit the mental model of documentation from two auditors. A comparison of the auditors' mental models with the result of the application of LSA shows that LSA's vector-space model is closer to the auditors' idea than a naive term-document frequency matrix.

17

Why Reading and Understanding Software Documentation Can Be Difficult

In Chapter 16 we have addressed the need in software product audits for a 'reading guide' for the product's documentation. This need might be partially due to the way the audit process works: auditors are relative outsiders to the development of the software who are only brought in (often late in the process) to assess the quality of the product. As a result, the auditors do not have a clear picture of the documentation and need to figure out first which information can be found where. We may ask ourselves, however, whether similar difficulties of comprehending documentation can occur within a development team too. We elicited and analyzed the mental models of software documentation from eight members of a single development team. We found indeed different levels of shared understanding between different people. To our surprise, the shared understanding within the team highly resembles the development process followed within the organization. From Conway's law we know that an organizations structure is mirrored in the structure of the software that the organization produces. Our findings suggest that the organization's development process may likewise be mirrored in the extent to which a common frame of reference exists within the development team. Hence, the development process followed may have implications for the effectiveness with which development knowledge – including architectural knowledge – can be shared through software documentation.

17.1 Introduction

During software development, important knowledge is shared between different members of the development team. Such knowledge can be shared for example face-to-face in more or less formal settings, but most software development projects rely to at least a certain extent on written documentation as a means of sharing knowledge. As a result, many software development projects end up with huge stacks of documents, in which one can easily lose track of the information needed. A question one may have is how effective this document-driven knowledge sharing is, and what its (in)effectiveness is based on.

The effectiveness of documentation within a development process is determined by the way in which the intentions of the authors correspond to the expectations of the potential readers. In a typical software development process, many different kinds of documents are produced and consumed at various points in time. The contents of those documents necessarily exhibit a certain amount of overlap. People may lose track of the meaning of individual documents; which information it contains and what its role is in the development process. When the expectations of the consumers of the documentation drift too far from the intentions of its producers, the ultimate consequence might be a need to rediscover already documented knowledge. In such a situation, the customer for example may need to explain his situation and requirements over and over again to different parties in the development process.

If an author's work is part of a team effort, as is the case in software development, the intention an author has with his work cannot be detached from his expectation of (the intention of) the work by other authors from that team. The author of, for example, a requirements specification may intend that document to be slightly more prescriptive than, say, a software architecture description, but only when the author has a certain expectation of the prescriptiveness of the software architecture description. In this sense, an author's intent can only be expressed in terms relative to his expectation of the other documents.

Ideally, the members of a development team share a certain understanding of (the role of) the different types of documentation. Such a shared understanding would imply that an author's intentions are perfectly clear to others in the development team. However, since the way in which one interprets and tacitly models the world (of which software documentation is but a small part) is co-determined by such intangible things as personal background, experience, and social interactions, we can expect different levels of shared understanding between different development team members.

Suppose I am looking for a particular piece of knowledge in a set of documents. Chances are that I will not read all documents, especially when the number of documents is high. Instead, I will rely on a mental model of the documentation that I tacitly built up. This model will tell me, for instance, whether some documents contain too much detail, or are instead too vague, to satisfy my knowledge need. This is far more likely to be a matter of heuristics rather than exact science, and I may even apply it to documents that I've never even read (or, as a consequence, will ever read). My understanding of the documentation, represented by this model, may be entirely different from the understanding of others – including that of the documents' author(s). Consequently, I may fail to find important information, or conversely the author may fail to relay important information to me. In other words, my expectation or impression of the contents and purpose of a document may not correspond to the contents or purpose intended by the author.

We elicited and analyzed the mental models of software documentation from eight members of a single development team. As expected, we found different levels of shared understanding between different people. To our surprise, however, the shared understanding within the team highly resembles the development process followed within the organization. It seems that people who work on similar tasks share much of the way in which they look at the project's documentation, but this shared understanding is lost when the number of 'process hops' increases.

From Conway's law we know that the structure of an organization is mirrored in the structure of the software that organization produces. Our findings suggest that the organization's development process may likewise be mirrored in the extent to which a common frame of reference exists within a development team. Hence, the development process followed may have implications for the effectiveness with which development knowledge can be shared through software documentation.

The remainder of this chapter is organized as follows. In the next section, we briefly introduce personal construct theory – the psychological theory upon which our research is based – and some of its associated methods. In §17.4 we describe the methodology we followed. In §17.5 we introduce the development organization in which our experiment took place. §17.6 presents the results of the experiment and reconstructs the shared understanding within the development team. In §17.7 we conclude this paper with a discussion of the results we obtained.

17.2 Personal Construct Theory

The methodology we employ builds on personal construct theory, a constructivist theory of personality and psychology developed by Kelly (1955). Personal construct theory says that every individual observes the world in terms of bipolar 'constructs', such as 'good-bad', 'right-wrong', et cetera. The use and definition of those constructs may differ from person to person. My comprehension of the construct ‘good-bad’ may not be the same as yours. Together, a person’s constructs make up a personal mental model of (part of) the world.

Kelly proposed not only a psychological theory, but also a method to systematically elicit and analyze personal constructs. This method, called the Repertory Grid Technique (RGT), can be seen as “an attempt to stand in others’ shoes, to see their world as they see it, to understand their situation, their concerns” (Fransella and Bannister, 1977).

The way RGT works is that it presents a subject with ‘triads’ of (i.e., groups of three) elements from the domain under investigation. In our case, since we are interested in the way people see software documentation, those elements are documents. When the three elements are presented, the subject is asked in which way two of them are alike and the third one is different. This identifies a bipolar construct or axis that is apparently part of the subject’s mental model. Given, for example, the triad {use case specification, test charter, software architecture description} a development team member might indicate that the use case specification and test charter have something in common, and that the software architecture description is different. The next question then would be: “What it is that makes the two documents similar and the other different?”, to which the answer could be: “The use case specification and test charter both show low-level details, and the software architecture description provides a high-level overview”. This indicates that the construct ‘low-level details/high-level overview’ is part of this person’s mental model.

Typical constructs that one may expect for software documentation include ‘abstract-concrete’, ‘overview-detail’, ‘specification-implementation’ and many more. Recall that those constructs are personal, hence their use and interpretation may differ between individuals. Two people who both distinguish ‘low-level detail’ documents from ‘high-level overview’ documents may disagree on the level of detail present in individual documents. Therefore, in a second step of RGT, the subject is asked to arrange all documents, including the three just used to elicit the construct, on a 5-point scale – from ‘low-level detail’ (1) to ‘high-level overview’ (5) – which provides insight into the way in which the subject applies the construct. This process is repeated until the method elicits no more new constructs.

The elements, constructs and rankings are kept in a tabular format called a ‘rank order grid’. In this grid, the rows correspond to personal constructs, the columns correspond to documents, and the cells contain a number from 1 to 5 that corresponds to how a particular document ranks on a particular construct. The data from this grid can be further analyzed.

The ‘Focus’ algorithm, for example, is a straightforward algorithm for cluster analysis of RGT data, which is simple enough that it can even be used to calculate element and construct clusters by hand (Jankowicz and Thomas, 1982). The algorithm involves a pairwise difference calculation and subsequent iterative reordering of constructs and elements “so that the differences between the resulting pairs are minimised”, eventually producing a similarity matrix from which construct or element clusters can be derived.

Shaw and Gaines (1989) present another method to analyze repertory grid data. The method entails a Focus-like comparison of constructs, but instead of comparing constructs within the same grid (as Focus does), this method compares constructs from different grids. Through this comparison, the relations between the ‘conceptual systems’ of two subjects can be analyzed. Shaw and Gaines define four types of relations: consensus, correspondence, conflict, and contrast.

When there is consensus, ‘experts use terminology and concepts in the same way’, i.e., two people use the same construct in a similar fashion. For example, two development team members may both use the construct ‘abstract-concrete’ to distinguish documents, If they do so in such a way that by and large documents that are abstract according to one team member are also abstract according to the other, and documents that are concrete according to the one are also concrete according to the other, there would be consensus among the two regarding the construct ‘abstract-concrete’.

Two people may also ‘use different terminology for the same concepts’. This is called correspondence and happens, for example, when two team members both distinguish documents based on their level of abstraction, but one of the team members uses the construct ‘abstract-concrete’ for this distinction whereas the other uses the construct ‘high level-low level’.

Conflict takes place when ‘experts use the same terminology for different concepts’, e.g., when two people use the same construct ‘abstract-concrete’ but disagree on which documents are abstract and which ones are concrete. And, finally, contrast is when ‘experts differ in terminology and concepts’, i.e., when two constructs from their respective rank order grids have nothing in common whatsoever.

17.3 Related Work

Several researchers have successfully applied personal construct theory and RGT to the software development process. The main applications reported in the literature can be summarized as ‘understanding the user’ and ‘understanding the developer’. Niu and Easterbrook (2007), for example, report on the use of RGT in goal-oriented requirements engineering to compare and clarify stakeholders’ terminology. And Karapanos and Martens (2007) use RGT as a means to characterize user diversity in (software-intensive) product development. Examples of the use of RGT to understand developers include the work of Baddoo and Hall (2002), who analyze the roles of practitioners in software process improvement, and Young et al. (2005), who analyze personality characteristics of the members of an agile development team. In our work, we use RGT to analyze the extent to which development team members have a shared understanding of a software product's documentation.

The topic of shared understanding, or shared mental models, between development team members has received considerable attention. Klimoski and Mohammed (1994) argue that such a model "implies a variety of content", which means that "the content of shared mental models might reference representations of tasks, of situations, of response patterns *or* of working relationships". In other words, there may be various types of shared understanding between team members of which the shared understanding of software documentation is only one. As far as we know, our study is the first to relate shared understanding to the production and consumption of software documentation. Still, our findings seem to be in line with those of other researchers such as Levesque et al. (2001) who show – through a focus not on documentation but on team process and expertise – how an increase in role differentiation leads to less interaction and eventually to a decrease in shared understanding.

17.4 Methodology

Based on the personal construct theory and analysis techniques introduced in §17.2, we constructed a method to analyze the mental models of various development team members regarding documentation. The method progresses through a number of steps: the first step is elicitation of the personal mental models of software documentation, in the next steps assessment of the levels of shared tacit and shared explicit understanding between team members takes place, and finally the results of those two assessments are combined to determine the aggregate level of shared understanding between team members.

The distinction between shared tacit and shared explicit understanding is taken from the field of knowledge management. In knowledge management, 'tacit knowledge' pertains to the kind of knowledge one possesses but cannot easily express. 'Explicit knowledge', on the other hand, is the knowledge that can be expressed in various forms, such as conversations, electronic communication, or writing (cf. (Nonaka and Takeuchi, 1995)). Likewise, with 'shared tacit understanding' we mean the 'alignment' of two mental models without the individuals necessarily being able to express the exact foundation upon which this alignment is based. With 'shared explicit understanding' we denote the possibility for two individuals to talk about the way they see software documentation in terms that each of them can understand, i.e., relate to their own mental models. Fig. 17.1 graphically depicts these two types of understanding.

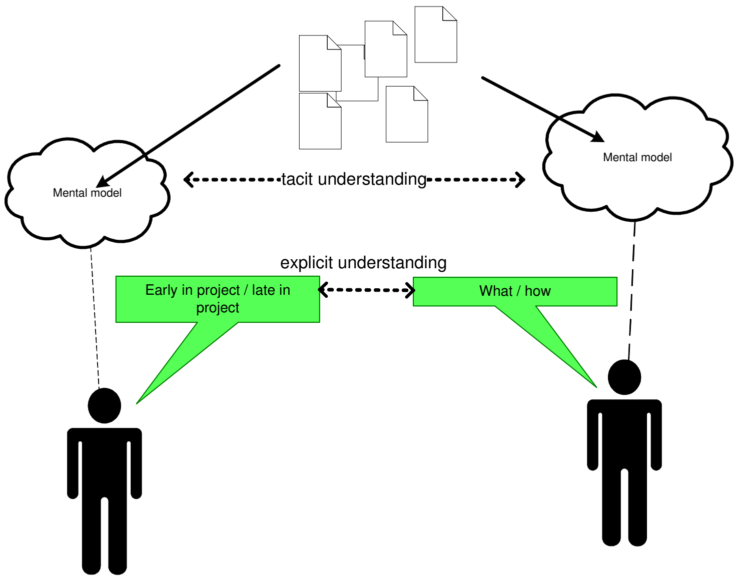


Figure 17.1: Shared tacit and explicit understanding between individuals

The steps of our method are further described below. This description includes the analysis methods we use, and their relation to personal construct theory from §17.2.

Elicit personal mental model We first elicit, using the Repertory Grid Technique, the individual expectations and impressions regarding the documentation. This provides us for each of the development team members with personal constructs that are used to distinguish and classify documents. Application of the Focus algorithm leads to a clustering of documents, which captures a personal mental model of the software documentation that may differ from person to person.

Determine level of shared tacit understanding A shared tacit understanding indicates that different team members have similar mental models of the document set as a whole, although it says nothing about how the individual mental models have been built up. We therefore take the alignment of two personal mental models as a measure for the level of shared tacit understanding between two team members, disregarding the constructs upon which those mental models have been based.

In order to compare the alignment of different mental models of documentation, we determine to what extent the documents clustering the team members perceive are correlated. The correlation between perceived document similarities can be calculated using the so-called Simple Mantel Test (cf. (Bonnet and Peer, 2002)). A high correlation found by this test indicates that two different mental models lead to similar document clusters, in other words that two individuals share a tacit understanding of the documentation. A correlation ≥ 0.5 is generally considered to be large (cf. (Cohen, 1988)), so we use a correlation of 0.5 as the lower bound to determine whether two development team members have a shared tacit understanding of the documentation.

Determine level of shared explicit understanding In addition to the levels of shared tacit understanding, we also determine the levels of shared explicit understanding between the team members. With a shared explicit understanding we mean that two development team members distinguish between documents in a similar fashion. For this distinction they need not necessarily employ the same terminology. For example, one team member may distinguish between abstract and concrete documents in much the same way as another team member distinguishes between early and late documents. Even though they do not use the same terminology, the distinction can in principle be expressed and communicated, provided that the two team members recognize that the terminology they use is synonymous. Shared explicit understanding is complementary to shared tacit understanding; a shared explicit understanding says nothing per se about similarities between the tacit mental models.

In order to determine the level of shared explicit understanding, we determine to what extent there is ‘consensus’ and/or ‘correspondence’ between two team members, i.e., the extent to which constructs from two mental models are being used in the same way, with the terminology used being the same (‘consensus’) or different (‘correspondence’, cf. §17.2). Shaw and Gaines (1989) propose a 80% similarity threshold as the lower bound for correspondence and consensus. We will consequently use this threshold to determine whether there is a shared explicit understanding between two development team members.

Determine level of shared understanding By combining the results of steps 2 and 3, we determine the structure of shared (tacit and explicit) understanding in the team. Whenever two individuals meet the thresholds of both shared tacit and shared explicit understanding, we say that they have a ‘shared understanding’. A shared understanding of the documentation implies similar expectations and/or intentions regarding individual documents (with respect to the other documents, cf. §17.1).

17.5 Development Team

We conducted our study within a development team that is part of a large software development organization. Software development is organized in what the organization calls ‘development streets’. A development street provides a standardized environment (i.e., processes and tools) for developing a particular type of software. There are for example different development streets for the development of Java-based software and for the development of dotNet-based software. The project we observed took place in one of those development streets.

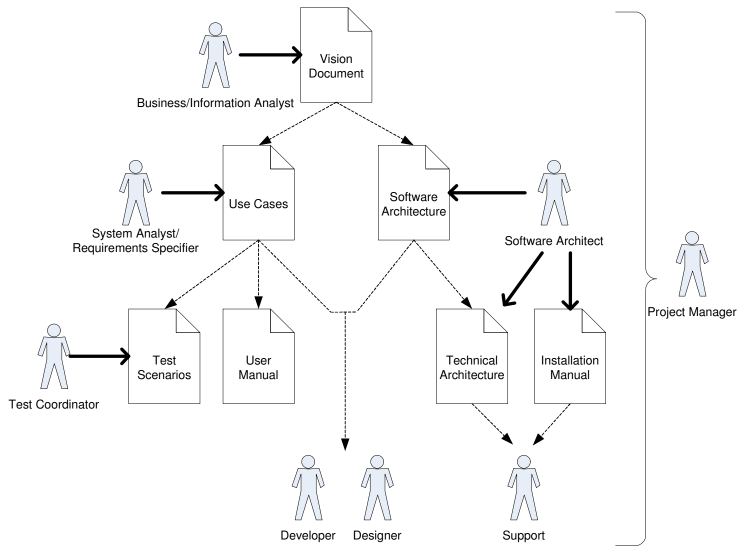


Figure 17.2: Development street

Fig. 17.2 shows the main stakeholders in the development street who produce or consume documentation. It also shows the documents with dependencies where information from one document is used in the production of another. The process can be summarized as follows. The business analyst starts off with determining the stakeholders, goals, and high-level functionality of the envisioned system, and reports the result of his analysis in a vision document. From the contents of this document requirements are derived, both functional and non-functional. The former are captured in use cases by the system analyst, while the latter are part of the software architecture description written by the software architect. Based on the use cases, the test coordinator devises test scenarios. The use cases are also reflected in the user manual (for which there usually is no dedicated author). From the software architecture description, the technical architecture is derived. This document, together with the installation manual, is primarily used by support personnel (for installation, configuration, administration, etc.). Both technical architecture and installation manual are written by the software architect. Finally, the use cases and software architecture documents are used by designers and developers in the construction of the actual software.

Fig. 17.2 originated from early discussions with one of the company's senior employees. Based on this figure, we invited a total number of 8 individuals who all participated, in different roles, in the same software development project. All invitees participated in our study. In a later phase, we independently verified with the participants that the process depicted in Fig. 17.2 is indeed the process followed within this development team[1].

At the time of our study, the team was working on the second release series (2.x) of the software. Together with the project manager, we selected a representative subset of software product documentation to be used in the repertory grid experiment. We had to make this selection since the number of documents produced in the project far outnumbered the maximum number of elements (± 20) that can feasibly be assessed by the repertory grid technique.

Table 17.1 summarizes the participants and documents that took part in the experiment. The roles listed are the job titles used by the participants themselves. Some of them differ slightly from the generic overview in Fig. 17.2. Where participants authored certain documents, those documents have been listed in their rows. The documents for which the author did not participate are listed at the bottom of the table.

17.6 Experiment and Results

In this section, we present the results for each of the steps of our methodology.

17.6.1 Personal construct elicitation

In eight separate interview sessions, we elicited personal construct for each of the eight development team members that took part in our research. Each session lasted for ap-

[1]We asked the document authors to list the top 3 roles for which their documents were mostly intended. By and large, their answers correspond to the process shown in Fig. 17.2. For example, the software architect listed himself, the lead developer and the developer as intended audience for the software architecture description, and the support role as the primary target for the technical architecture description (followed by developer and lead developer).

Table 17.1: Participants and documents

Participant	Role	Documents authored
A	Application developer	
B	Information architect	*Vision document*
C	Lead developer	
D	Application developer + support	
E	Project manager	
F	Application designer	*Vision document for release 2* *Use case specifications*
G	Tester/testcoordinator	*Master test plan* *Test charters*
H	Software architect	*Software architecture* *Technical architecture* *Installation manual*
		Infosec/risk assessment *Proposition for release 2* *User manual* *Glossary*

proximately two hours, and at each session there were two attendees: the development team member and the interviewer conducting the experiment. All interviews were conducted by the same interviewer.

Before the start of the experiment, the participants were asked to briefly introduce themselves and to describe their role in the project. During the experiment, no direct questions about (the use of) the documentation were posed; all data about the documents was elicited through triads (cf. §17.2). No constructs were provided to the participants in advance, and the interviewer took care not to provide or suggest any constructs during the interview; all constructs were in the course of the experiment proposed by the participants themselves.

We obtained eight rank order grids, one for each participant. Elicitation of personal constructs lasted until the participants indicated the could think of no more additional constructs different enough from the ones already provided. All participants provided between 10 and 15 constructs before the elicitation was completed. Table 17.2 shows the rank order grid of one of the participants as an illustration of the type of results we obtained.

Table 17.2: Rank order grid for one team member

1	*Infosec/risk assessment*	*Proposition for release 2*	*Vision document*	*Vision document for release 2*	*Test charters*	*Use case specifications*	*Master test plan*	*Software architecture*	*Technical architecture*	*Glossary*	*Installation manual*	*User manual*	5
earlier in project	1	3	2	3	4	4	4	4	4	4	5	5	later in project
execution	5	5	4	4	2	3	3	3	2	4	1	1	support
customer requirements	2	1	2	2	4	3	3	4	4	3	5	3	realisation
has to do with testing	5	5	5	5	2	1	1	5	5	3	5	2	has nothing to do with testing
time dependency	1	1	1	1	1	1	1	2	1	5	1	1	no time dependency
understood by user	4	4	4	4	1	2	4	5	5	1	5	1	not understood by user
functionality	2	2	2	2	1	1	2	4	5	2	5	1	technique
has to do with planning	5	1	1	1	5	1	1	3	5	5	5	5	has nothing to do with planning
has to do with functionality	4	3	3	3	1	1	3	3	5	4	5	1	has nothing to do with functionality
expectation	2	1	3	3	5	4	3	4	5	5	5	5	reality
concrete (for builders)	5	5	5	5	2	2	5	2	2	2	1	2	abstract (for builders)
says something about the application	4	4	4	4	1	4	4	3	3	4	3	1	says nothing about the application
has to do with infrastructure	3	3	3	3	5	5	4	2	1	5	1	5	has nothing to do with infrastructure

17.6.2 Shared tacit understanding

From the rank order grids of the eight development team members, the perceived similarities between the documents can be calculated using the Focus algorithm. Documents that are similarly ranked across the constructs in a grid are more similar to each other than documents that are ranked differently. This is illustrated in Fig. 17.3, which shows a hierarchical clustering of documents based on the data from Table 17.2. The numbers are the percental similarity scores calculated with Focus, e.g., the perceived similarity between the *Installation manual* and the *Technical architecture* is 92% and the similarity of that cluster with the *Software architecture* is 83%.

The perceived similarities between documents represent the individual's mental model of the software documentation. Table 17.3 shows the correlation between individuals' mental models of product documentation, i.e., the document similarity matrices. The correlations have been calculated using a simple Mantel test with 10000 randomizations. We interpret a high ($r \geq 0.5$) correlation of perceived document similarity as a shared tacit understanding of the documentation between team members.

From Table 17.3 we see that almost all correlations found are significant (low p-value). The only non-significant correlation is between E and A, which suggests that those two people do not share a tacit understanding of the documentation at all. Ap-

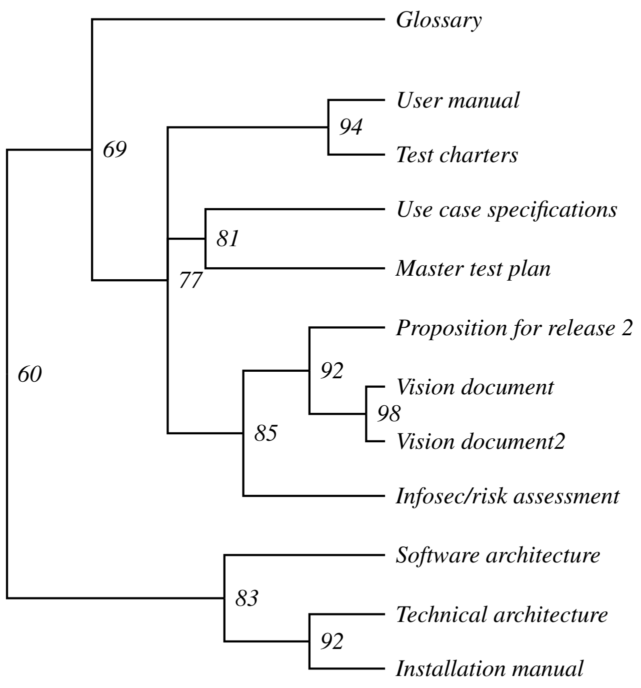


Figure 17.3: Hierarchical clustering of documents based on data from Table 17.2

proximately half of the remaining, significant correlations are high (≥ 0.5) and indicate a strong shared tacit understanding between the individuals involved. Some members of the development team (e.g., H) seem to share their understanding with most other team members, whereas some other people (e.g., B and E) strongly share an understanding with only one other team member. There is not a single person whose tacit understanding is completely detached from the rest of the team, however.

17.6.3 Shared explicit understanding

Whereas in the previous section we analyzed the shared mutual tacit understandings between development team members, in this section we will look at the explicit understanding that is shared between individuals. Below we show a summary of the team members who have corresponding constructs at a similarity of 0.80 or higher.

Table 17.4 shows the constructs that team members use in a similar fashion. For

Table 17.3: Shared tacit understanding

		A	B	C	D	E	F	G
B	r	0.395271						
	p	0.002700						
C	r	0.656839	0.420986					
	p	0.000100	0.005599					
D	r	0.540594	0.486415	0.599269				
	p	0.000300	0.002200	0.000300				
E	r	0.151361	0.408455	0.476889	0.400135			
	p	0.123588	0.002800	0.000300	0.007399			
F	r	0.336944	0.641328	0.601511	0.498014	0.430471		
	p	0.004500	0.000100	0.000300	0.002300	0.002300		
G	r	0.558442	0.469948	0.695907	0.544832	0.421918	0.560675	
	p	0.000100	0.002100	0.000100	0.000300	0.002300	0.000100	
H	r	0.471787	0.439568	0.702911	0.519408	0.600942	0.500522	0.754730
	p	0.000400	0.002500	0.000100	0.001900	0.000100	0.000800	0.000100

example, the way in which G applies the ‘concept’ vs. ‘final’ construct resembles the way in which H applies the ‘realisation’ vs. ‘customer requirements’ construct. In other words, documents that G regards as ‘final’ tend to be regarded by H as containing customer requirements. From this table we can also see that F and D as well as F and H share more than one way of applying constructs; for all the others correspondence is limited to a single pair of constructs. F and H also account for the only instance of consensus: they were the only ones to independently come up with literally the same construct (functionality vs. technique) and to apply it similarly too.

Based on the data from Table 17.4, Table 17.5 shows which team members have (one or more) correponding constructs. Hence, from Table 17.5 we can easily identify who in the development team share an explicit understanding of the documentation.

17.6.4 Shared understanding

If we combine the data from Tables 17.3 and 17.5, we can identify which team members share both a tacit and an explicit understanding. Table 17.6 depicts this combination, and shows which team members share a tacit understanding(t), explicit understanding (e), or both (t+e). Based on the data from Table 17.6, Fig. 17.4 shows a graph-like structure in which team members that have a shared understanding (t+e) are connected. For the sake of clarity, the roles of the team members and the documents they authored have been added to the figure (cf. Table 17.1).

Fig. 17.4 seems to suggest that there is a shift in understanding of the product documentation within the development team. A chain of people $A \leftrightarrow B \leftrightarrow C$ is

Table 17.4: Shared explicit understanding: Correspondence and consensus

Team members	**Corresponding constructs**
A ↔ C	before start of project/end of project begin/end
A ↔ G	written earlier/later formal/informal language
B ↔ D	early/late in project what/how
B ↔ F	quality/customer domain unimportant/important (for me)
B ↔ G	solution domain/problem domain concrete/abstract
C ↔ H	begin/end supporting/execution
D ↔ F	how we will make the sw/how the sw has been made about type of app/about app as made first project phase/last project phase about type of app/ about app as made
F ↔ H	about type of app/about app as made abstract/concrete - functionality / technique
G ↔ H	concept/final realisation/customer requirements

Table 17.5: Shared explicit understanding

	A	B	C	D	E	F	G
B							
C	X						
D		X					
E							
F		X		X			
G	X	X					
H			X			X	X

Table 17.6: Shared understanding (tacit + explicit)

	A	B	C	D	E	F	G
B							
C	t+e						
D	t	e	t				
E							
F		t+e	t	e			
G	t+e	e	t	t		t	
H			t+e	t	t	t+e	t+e

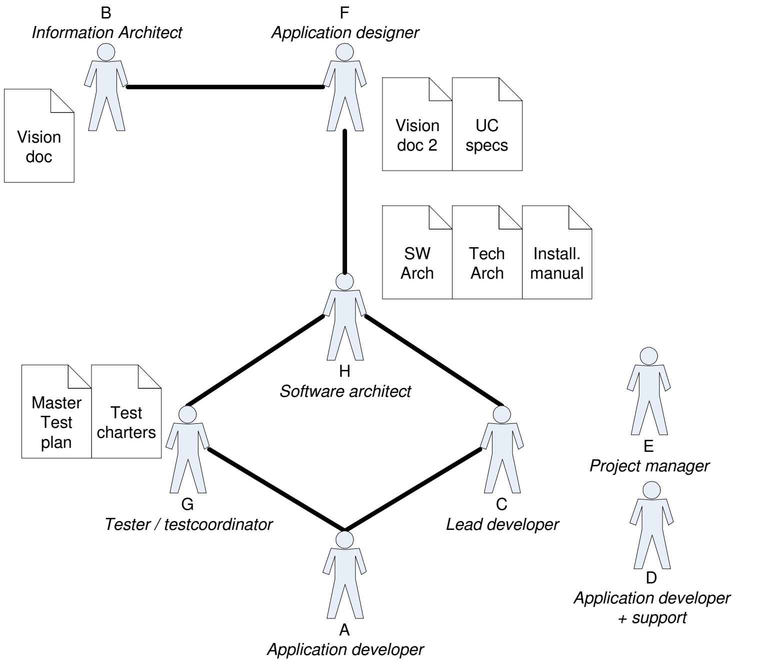


Figure 17.4: Shared understanding

formed in which one person B may have a shared understanding with persons A and C, whereas between A and C such a shared understanding does not exist. In a way, the figure resembles a familiar children's game known as 'Chinese whispers'. In this game, a sentence is whispered in the ear of the first child, who whispers it to another child, and so on, until the last child is reached who says the sentence out loud. Hardly ever will the sentence that the last child heard be the same sentence that was given to the first child, since the message that was passed suffers from accumulated errors and distortions.

This chain-like structure of shared understanding obviously has repercussions for the way in which people's expectations of documentation do or do not match the intentions of the authors in the chain. If we look again at Fig. 17.4, we see that the author of the *Vision document* (B) has a shared tacit and explicit understanding with F only. This implies that his intentions for the *Vision document* are really only fully understood by F. The further down the chain, the less likely it is that people fully appreciate this document. A central role in this picture is for H, which suggests that other team members' expectations regarding the *Software architecture*, *Technical architecture*, and *Installation manual* are probably most aligned to the intentions of their author. The figure also suggests that, for whatever reason, E and D may experience the most trouble when exploring the documentation since both of them share no tacit *and* explicit understanding with any of the other participants in our experiment (although they do have a shared tacit *or* explicit understanding with various team members). Note, however, that a lack of shared understanding does not imply that team members cannot or do not read the information in the documentation. Rather, the documents' contents may be placed and interpreted in a context different than the one the author envisioned – for example, a document may be seen by the reader as providing guidelines rather than prescriptions as the original author intended.

17.7 Discussion and Conclusion

One of the most striking observations from the structure in Fig. 17.4 is its obvious resemblance of the process in Fig. 17.2. People that are close to each other in the development process – because one of them uses a document that was produced by the other – also tend to have a shared understanding of the software documentation[2]. Although not a perfect match, the process followed in which high level business information is stepwise refined into more technical information is highly visible in Fig. 17.4.

[2]Note that, even though participant F uses the job title 'application designer', his role seems to have been more that of an analyst and requirements specifier evidenced by the documents he authored.

We should reiterate here that the derivation of Fig. 17.4 is completely detached from the derivation of Fig. 17.2. In the latter case, we directly asked for information about the development process, while the structure of shared understanding in Fig. 17.4 has been fully based on participants' statements about the software documentation. That we were able to reconstruct an approximation of the development process based on statements solely about the documentation is a remarkable (and unexpected) result.

We understand that our findings should be further tested and verified. We know for example that the team we observed has had some issues related to documentation, especially the tendency to go back to the customer for clarifications. Although we can explain this behavior with the results we obtained, we cannot be completely sure that our explanation is the correct one. We would also like to know whether our method obtains similar results in different organizations.

Nevertheless, there seems to be an important lesson in our findings: it appears that the development process one follows is reflected in the way in which individual team members understand each other. This has obvious implications for areas such as process management and process improvement. A long, chain-like process for instance may lead to knowledge dissipation more easily than a compact, web-like structure. Consequently, if one wants to enhance a team's common frame of reference, the way in which the development process supports collaboration between individuals could be key.

18 Conclusions

In this part we have studied how to support auditors in discovering architectural knowledge. We have devised tools that support the discovery of SOLL-AK and IST-AK in the input respectively throughput stages of an audit. We have also addressed the relation between requirements and architecture, and explored the reason why sharing architectural knowledge through written documentation is hard. In this chapter, we revisit the research questions answered in this part and summarize the answers our research provided, and discuss additional pointers to further research.

18.1 Contributions

In this section, we revisit the research questions answered in this part and summarize the answers our research provided. Recall that auditors rely on architectural knowledge pertaining to the desired state of a software product, as well as to the current state of that product. Hence, in order to support auditors in discovering relevant architectural knowledge, we must enhance the discoverability of SOLL-AK as well as that of IST-AK. Our answer to the main research question (RQ-III.1) is therefore provided by the answers to the two subquestions RQ-III.2 and RQ-III.3 in the following two subsections.

18.1.1 How can the discoverability of relevant SOLL-AK be enhanced? (RQ-III.2)

In Chapters 14 and 15 we have examined how the discoverability of relevant SOLL-AK, i.e., architectural knowledge pertaining to the desired state of a software product, can be enhanced. Recall that this particular type of architectural knowledge consists of a customer's high-level quality requirements which are subsequently enhanced by the auditor to form quality criteria: concrete measures that should or should not be present in a software product.

To answer RQ-III.2, in Chapter 14 we first answered subquestion RQ-III.4: what is the relation between requirements and architecture? In that chapter, we argued that there is no fundamental distinction between architecturally significant requirements and architectural design decisions. Drawing on the similarity between requirements and architecture, in Chapter 15 we answered subquestion RQ-III.5 (how can the reuse of quality criteria be supported?) by drawing analogies between quality criteria and architectural design decisions. This led to the definition of QuOnt, an ontology for codification of quality criteria. QuOnt is a partial extension to Kruchten's ontology of architectural design decisions, and augments that ontology with explicit, reified 'effect' relations between quality criteria (or design decisions) and quality attributes, as well as comparisons between these effects.

In that same chapter, we introduced ontology-driven visualization as a novel way to visualize architectural design decisions, especially quality criteria. This visualization combines the strengths of two existing types of design decision visualization and overcomes their weaknesses. A prototype decision support system based on QuOnt and ontology-driven visualization was discussed as a proof of concept. This proof of concept showed how this combination of QuOnt and ontology-driven visualization aids auditors in reusing quality criteria, i.e., helps them discover relevant SOLL-AK.

18.1.2 How can the discoverability of relevant IST-AK be enhanced? (RQ-III.3)

In Chapters 16 and 17 we have examined how the discoverability of relevant IST-AK, i.e., architectural knowledge pertaining to the actual state of a software product, can be enhanced. We have focused especially on the architectural knowledge that is reflected in the documentation that accompanies a software product.

In Chapter 16, we answered the question how the discovery of architectural knowledge in software product documentation can be supported (RQ-III.6). We argued how Latent Semantic Analysis (LSA) can be used to uncover the latent semantic structure that underlies the software product documentation, and how this LSA-space can be explored to construct a 'reading guide'. We also introduced a way to reconstruct the auditors' mental models of the documentation by means of the repertory grid technique. We compared the mental models of two auditors with the model resulting from LSA to validate the working of our proof of concept.

We also used the repertory grid technique in Chapter 17 to answer the question why sharing architectural knowledge through written documentation is so hard (RQ-III.7).

In this chapter, we compared the mental models of software documentation from eight members of a single development team. We concluded that the extent to which team members have a shared understanding of the documentation reflects the development process that the team follows.

Since auditors are generally not part of the development team that produces the product to be audited, our findings from Chapter 17 lead to a dual answer to the question how the discoverability of relevant IST-AK can be enhanced (RQ-III.3). On the one hand, the use of text mining techniques such as LSA has a proven advantage to locate relevant architectural knowledge in product documentation. On the other hand, such technological solutions are probably not a panacea. Auditors will have to stay in touch with the product's supplier and talk with members of the development team, since being in touch seems the key to the development of shared understanding. In short, engineering high-quality software is as much about people as it is about technology.

18.2 Discussion

A research project is never finished. Answers lead to new questions, and results lead to new ideas. Some ideas for future work have already been discussed in the previous chapters. In this section, we conclude Part III with a discussion on the results presented in this part and additional pointers to further research.

18.2.1 Requirements and architecture: Towards tighter collaboration

Our claim that architectural design decisions and architecturally significant requirements are really the same, only observed from different directions, is one that many may find provocative, or even controversial. Even though it may be controversial, our claim is intended to be constructive since we feel it may open the road towards tighter collaboration between the requirements and architecture fields.

Currently, the challenges in requirements management and architectural knowledge management are approached as if there are two separate 'information wells'. Both communities perform research on comparable challenges without paying too much attention to what the others are doing. If we recognize and acknowledge that both communities are in fact looking at the same 'magic well' from different angles, we open the door for tighter collaboration between the fields as well as reuse of each other's research results.

Although there are still many open issues in requirements management, when compared to architectural knowledge management that field is much more mature. This is not to say that the current work in researching architectural knowledge management is in vain. On the contrary, innovations in architectural knowledge management might have a beneficial impact on requirements management problems as well. However, we as a software architecture community can probably learn a great deal from our colleagues in requirements engineering.

The magic well has yet another implication: software architecture is not merely the domain of the architect. Each architecturally significant requirement is already a decision that shapes the architecture. Requirements engineering and software architecture have considerable overlap, but also use different perspectives. We should, therefore, further align our tools, methods, and techniques by focusing more on our common ground, rather than on our differences. Further exploring and exploiting the *commonalities* between architecturally significant requirements and architectural design decisions would serve that goal.

18.2.2 QuOnt and LSA: A quality evaluation toolkit

Our research has led to the implementation of two proofs of concept: a decision support system for quality criteria reuse, based on the QuOnt ontology, and an LSA-based text analysis prototype. We have presented and discussed them as separate tools, but a combination of the two could lead to a synergetic ‘quality evaluation toolkit’.

Through such a quality evaluation toolkit, parts of the software product documentation could be automatically related to particular quality criteria. To this end, the textual description of a quality criterion could be used as a ‘document’ in the LSA-space. The quality criterion document could even be augmented with further background information, e.g., the Wikipedia page for ‘Failover’ could be used for the quality criterion that the product should use failover. After the quality criterion document has been placed in the LSA-space, parts of the documentation that have a meaning closely related to the meaning of that quality criterion (i.e., that ‘talk about’ issues related to that criterion) can be identified through similarity calculations. Such a toolkit would create a dynamic index to the product documentation from the quality criteria used in the audit.

Since the QuOnt ontology defines relations between quality criteria and between quality attributes, further relations between product documentation parts can be inferred. For example, the toolkit could infer that an interest in performance implies an interest in both resource behavior and time behavior, or that an interest in user authentication measures includes an interest in such topics as passwords and single sign-on. The toolkit could then locate product parts that talk about topics related to a topic of interest through a combination of inference and latent semantic analysis.

18.2.3 Personal construct theory as knowledge management instrument

In Chapters 16 and 17 we have seen the application of personal construct theory, and especially the repertory grid technique, to the elicitation and comparison of personal mental models of software documentation. The assessment of 'levels of shared understanding' between development team members in Chapter 17 connects the realm of explicit architectural knowledge (in documents) with that of tacit or implicit architectural knowledge (in people's minds). In other words, this assessment addresses aspects related to both codification and personalization.

An interesting question, therefore, is whether we can use personal construct theory as a knowledge management instrument in its own right. We can think of several applications that might have a beneficial impact on software development. For example, one of the goals of knowledge management is to preserve knowledge possessed by key individuals in the organization. A valid question, therefore, is who *are* the key individuals in the organization, in terms of the type of knowledge they possess. Analysis of personal constructs could help the identification of 'knowledge hubs', i.e., people who form a connection between different kinds of knowledge in the organization. In Chapter 17, for instance, the software architect seems to act as such a knowledge hub between analysts and developers.

Other applications of personal construct theory and assessment of shared levels of understanding might be more proactive. For instance, one could steer the training of development team members to repair 'gaps' in shared understanding (this would, by the way, probably work better for the improvement of shared explicit understanding – the vocabulary – than for tacit understanding). Or, to go one step further, one could try to form a development team for a new project based on *a priori* (historical) data on their mutual levels of shared understanding, so that the development team attains a high number of 'knowledge connections' between the team members, and hence a high overall level of shared understanding within the team.

Such applications as the ones we sketch above may be worthwhile to investigate. Further research would be necessary to assess the validity and applicability of these ideas.

18.2.4 Architectural knowledge discovery in a broader scope

In this thesis, we have equalled 'architectural knowledge discovery' (AKD) to an enhanced discoverability of relevant architectural knowledge in the context of software product audits. We believe, however, that AKD has merit in a broader scope.

We envision knowledge discovery techniques being used in a broad range of architectural knowledge management tools and methods. The role of AKD would include the refinement of existing architectural knowledge from such diverse sources as documents, email, meeting minutes, and source code. Those sources contain mainly unstructured (documents, etc.) or at best semi-structured (source code) information. Refinement by AKD would therefore mainly consist of adding structure to this information. We may also envision AKD in a forward-engineering sense, to judge the quality of the evolving (architectural or otherwise) documentation of a system, and to give guidelines as to where the documentation needs attention.

Architectural knowledge refinement could play a major role in the transition from personalization to codification (cf. §3.4). In the early phases of architecting, often little attention is paid to a structured description (codification) of the architectural knowledge that is created, such as the architectural design decisions taken and the alternatives considered. More emphasis is usually put on personalization, i.e., knowing who knows what and discussing options and issues directly with peers. It is usually only after the fact – if ever – that the architectural knowledge created during this process is captured in architectural descriptions and other documents. However, this early architecting phase often leaves many traces in for example minutes, emails, or forum discussions. AKD could be employed to mine and structure those traces of unstructured information, thereby providing semi-automated support for effective codification of the architectural knowledge.

Conclusion

19 Quo Vadis?

In this thesis, we have discussed how architects and auditors can be supported by employing architectural knowledge management practices to their daily work. Notable contributions of this thesis – discussed in more detail in the concluding chapters 5, 12, and 18 of Parts I to III – include:

- Theoretical models of architectural knowledge and architectural knowledge management, including the development of a core model of architectural knowledge and the recognition of different philosophies of architectural knowledge management.
- The development and evaluation of knowledge management tools, including knowledge sharing portals, wikis, decision support systems, and text mining systems.
- Insights regarding the nature of architectural knowledge management, including the recognition that architecturally significant requirements and architectural design decisions are really the same and the observation that architectural knowledge codification alone is not enough.

In their overview of the maturation of the software architecture field, Shaw and Clements (2006) conclude with an outlook on future work in software architecture research. They argue that promising topics for research include a focus on architectural design decisions and their link to quality attributes, and the organization of architectural knowledge to create reference materials. In a follow-up on their article, Shaw and Clements (2009) reiterate the strong potential of a focus on design decisions. We like to think that our research proves them right.

Like Shaw and Clements, we like to take this opportunity to provide our own view on software architecture research in the foreseeable future. We see four main trends that mostly focus on specific use cases for architectural knowledge management:

1. *Sharing architectural knowledge*, especially in a globally distributed setting.
2. *Further aligning architecting with requirements engineering*, focusing on the commonalities instead of differences between the two domains;
3. *Intelligent support for architecting*, focusing on specific architecting use cases related to intelligent and/or real-time support for architects;
4. *Establishing a body of application-generic architectural knowledge*, to further promote learning and reuse.

These trends correspond to current areas of active research, which are related to the research in this thesis. Sharing in a globally distributed setting is a step beyond the more 'local' sharing of architectural knowledge that was the subject of Part II. Alignment of requirements engineering and architecting may build on the similarities between the two fields as identified in Chapter 14. Intelligent support for architecting may use aggregations of knowledge technologies such as the ones described throughout this thesis. And, finally, the establishment of a body of architectural knowledge may on the one hand draw on knowledge structures such as the core model from Chapter 4 and the QuOnt ontology from Chapter 15, and on the other hand provide a consensus of expert knowledge that intelligent architecting decision support systems necessarily rely on.

An additional area that currently seems not to be on the research agenda is the investigation of commercial aspects of architectural knowledge management. Apart from the 'technocratic' (i.e., codification) and 'behavioral' (i.e., personalization) schools, Earl's taxonomy of knowledge management strategies identifies the 'economic schools', which focus more on knowledge assets than on knowledge bases or knowledge exchange. In our opinion, this warrants future research in the direction of commercialization of architectural knowledge, i.e., the protection and exploitation of architectural knowledge to produce revenue streams.

Samenvatting

Het is onmogelijk ons nog een wereld zonder software voor te stellen. Software is een onlosmakelijk deel gaan uitmaken van ons dagelijks leven, soms zonder dat we ons er al te zeer van bewust zijn. In de supermarkt worden onze boodschappen aan de hand van barcodes ‘gescand’. We betalen met onze pinpas, of halen met diezelfde pas geld ‘uit de muur’. Bij autopech wordt eerst de boordcomputer uitgelezen voordat de motorkap open gaat. Steeds meer mensen bellen via ‘Voice over IP’ (VoIP), waarbij telefoongesprekken feitelijk over computernetwerken lopen. Aandelenbeurzen, nieuwsdiensten, treinen, vliegtuigen, ziekenhuizen, mobiele telefoons, televisies en hard disk recorders; overal waar we kijken draait de wereld op software.

Het bouwen van die software, en met name het bouwen van zeer grote softwaresystemen, staat ook wel bekend als ‘software engineering’. Binnen de software engineering is men continu op zoek naar betere technieken om betere software te bouwen. Hierbij dient opgemerkt te worden dat software engineering als discipline eigenlijk nog maar betrekkelijk jong is. De term ‘software engineering’ ontstond weliswaar als analogie naar andere ingenieursdisciplines – met name bouwkunde was een bron van inspiratie – maar de 40 jaar die verstreken zijn sinds de term in gebruik kwam staan in geen verhouding tot de duizenden jaren bouwkundige historie.

De bouwkundemetafoor is ook terug te vinden in het deelgebied dat bekend is komen te staan als ‘software-architectuur’. Software-architectuur behelst het structureren en modelleren van software in modulaire eenheden op een hoger abstractieniveau dan dat van de implementatie (de uiteindelijke ‘code’). Door code op een logische manier te groeperen in modules en de interne werking van een module als het ware af te schermen voor de buitenwereld (waaronder andere modules) wordt het mogelijk op een andere manier naar de werking van software te kijken dan wanneer we de details van de software regel voor regel zouden bestuderen. En door vooraf over de architectuur na te denken kan deze een leidraad vormen voor de latere implementatie van de software. Van de software-architectuur wordt dan ook wel gezegd dat deze de manifestatie is van de vroege ontwerpbeslissingen.

Sinds begin jaren zeventig, toen het begrip software-architectuur in zwang raakte, is er in het vakgebied veel vooruitgang geboekt met het beschrijven en analyseren van software-architecturen, vaak in termen van ‘componenten’ (de bouwblokken) en ‘connectoren’ (de verbindingen tussen die bouwblokken). Het beschouwen van architectuur als samenstelling van componenten en connectoren betekent echter dat we architectuur altijd zien als het eindresultaat, en nooit als de weg naar dat resultaat toe. De componenten en connectoren *per se* vertellen ons immers niets over de afwegingen die de software-architect heeft gemaakt, de alternatieven die hij heeft overwogen (en verworpen), de bij deze overwegingen betrokken belangen, et cetera. Kortom, door alleen te kijken naar het eindresultaat wordt belangrijke kennis omtrent de architectuur buiten beschouwing gelaten. Wanneer architectuurkennis slechts gedeeltelijk wordt overgedragen, dan leidt dat tot additionele, moeilijk te kwantificeren kosten. Bovendien kan deze onvolledige kennisoverdracht leiden tot zogenaamde 'ontwerp-erosie'.

Het begrip 'architectuurkennis' krijgt de laatste jaren meer en meer nadruk. Er zijn vier belangrijke perspectieven op wat zulke architectuurkennis precies inhoudt, gedreven door vier dominante toepassingen van die kennis. Ten eerste is er de min of meer klassieke benadering van architectuurkennis als (geformaliseerde) kennis over de componenten en connectoren. Dit perspectief komt met name voort uit de toepassing van dynamische architecturen voor bijvoorbeeld 'zelfhelende' systemen: systemen die hun eigen software kunnen analyseren en de architectuur van die software zelf aan kunnen passen om bijvoorbeeld defecte componenten uit te schakelen en te vervangen door alternatieven.

Vanuit een tweede perspectief wordt architectuurkennis gedocumenteerd in een groeiende collectie van zogenaamde 'ontwerppatronen' die kennis over herhaald optredende problemen koppelt aan bewezen oplossingen. Deze patronen (of 'patterns') worden benoemd en besproken in boeken en op conferenties, en zijn daarmee zowel een vorm van kennisoverdracht als een verrijking van het vocabulaire dat de software-architect ter beschikking staat.

Een derde perspectief op architectuurkennis behelst de traceerbaarheid van de architectuur naar het onderliggende pakket van eisen en wensen. Slechts wanneer deze traceerbaarheid in orde is kunnen wijzigingen in de eisen (bijvoorbeeld door een veranderende klantvraag, maar ook door veranderingen in wet- en regelgeving of nieuwe technologische doorbraken) vertaald worden naar wijzigingen in de architectuur. Omgekeerd kan een bestaande architectuur invloed hebben op de (on)mogelijkheid om nieuwe of veranderende eisen te realiseren.

Ten slotte is er een vrij recente benadering van architectuurkennis als de ontwerpbeslissingen die leiden tot het architectuurontwerp. Vanuit dit perspectief dienen de ontwerpbeslissingen als concrete entiteiten te worden onderkend. In tegenstelling tot het uiteindelijke architectuurontwerp hebben de ontwerpbeslissingen intrinsieke relaties met de overwogen alternatieven, de uiteindelijk gekozen oplossing, en de verschillende belangen – waaronder economische, politieke, en technologische randvoorwaarden – die een rol gespeeld hebben in het beslissingsproces. In een groter verband kan een focus op ontwerpbeslissingen dan ook gebruikt worden om de verschillende 'vormen' van architectuurkennis op een lijn te brengen en aan elkaar te relateren.

Dit proefschrift is een resultaat van het GRIFFIN onderzoeksproject waarin industriële en academische partners samengewerkt hebben aan betere manieren voor architectuurkennismanagement. Ons onderzoek is gestoeld op een beslissingsgeoriënteerd perspectief op architectuurkennis. Wij beschouwen het beslissingsproces als een lus, waarin afgewogen wensen en eisen leiden tot ontwerpbeslissingen die vervolgens weer leiden tot nieuwe wensen en eisen die in vervolgbeslissingen meegewogen worden, et cetera, et cetera. Een analyse van vier verschillende organisaties aan de hand van een model waarin deze 'beslislus' centraal staat heeft geleid tot de identificatie van vier probleemgebieden met betrekking tot architectuurkennismanagement: het *delen* van architectuurkennis binnen en tussen organisaties, het *ontdekken* van relevante architectuurkennis in bestaande documenten en informatiesystemen, het *voldoen aan* opgestelde architectuurregels en richtlijnen, en het *traceren* van de verbanden tussen ontwerpbeslissingen en andere kennisentiteiten. Hieruit zijn binnen het GRIFFIN project vier overeenkomstige onderzoekslijnen ontstaan. In dit proefschrift gaan we in op twee van de geïdentificeerde problemen: het delen van architectuurkennis, vanuit het perspectief van de software-architect die verantwoordelijk is voor het ontwerpen van de software; en het ontdekken van architectuurkennis, vanuit het perspectief van de auditor die een kwaliteitsanalyse van de ontwikkelde software uitvoert.

Het delen van architectuurkennis is belangrijk omdat architecten typische kenniswerkers zijn. Zij spelen vaak een centrale rol in het softwareontwikkelproces en zijn verantwoordelijk voor het nemen van belangrijke ontwerpbeslissingen. Hiervoor communiceren ze met allerlei belanghebbenden over technische en niet-technische zaken, en proberen ze de beste afwegingen te maken rekening houdend met de wensen en eisen van deze belanghebbenden. Effectieve methoden en technieken om architecten te helpen kennis te delen zijn zeer gewenst. In dit proefschrift presenteren we verschillende van deze methoden en technieken.

Om na te gaan wat voor methoden en technieken architecten kunnen helpen in het delen van architectuurkennis hebben we vier praktijkstudies uitgevoerd bij een softwareontwikkelorganisatie. Om de verkregen resultaten nader te valideren hebben we aansluitend een vijfde studie uitgevoerd: een vragenlijstonderzoek waaraan architecten van vier organisaties in Nederland hebben deelgenomen. Deze vijf studies passen goed binnen een zogehete 'action research' cyclus die bestaat uit een diagnostische fase, gevolgd door enkele ontwikkelfases en een evaluatie- en reflectiefase. Deze kwalitatieve onderzoeksmethodiek stelde ons in staat om in nauwe samenwerking met de architecten de huidige problematiek met betrekking tot het delen van architectuurkennis boven water te krijgen, en tot bruikbare oplossingen en verbeteringen te komen die goed aansluiten bij hun activiteiten en kennisbehoeften.

Tijdens een eerste diagnose hebben we het gebruik van bestaande hulpmiddelen voor het delen van architectuurkennis onderzocht. Wat hierbij opviel, was dat architecten deze hulpmiddelen grotendeels naast zich neer legden. Tijdens interviews werden hier verschillende verklaringen voor gegeven, waaronder de steile leercurve van deze hulpmiddelen, de tijd die het kost om deze hulpmiddelen te gebruiken en de beperkte toegevoegde waarde ten opzichte van meer traditionele communicatievormen waaronder telefoon of e-mail. Architectuurkennis bleek in deze organisatie veelal gedeeld te worden tijdens informele bijeenkomsten of op de gang bij de koffieautomaat. Het bleek dat kennisdeling middels specialistische hulpmiddelen iets is waar architecten gemotiveerd voor moeten zijn. In ons onderzoek hebben we een aantal voorwaarden gedefinieerd die noodzakelijk zijn om een omgeving te creëren waarbinnen architecten aangemoedigd worden om hun kennis te verspreiden.

De volgende fase van ons onderzoek behelste het ontwerpen van efficiëntere hulpmiddelen voor het delen van architectuurkennis. Na een uitgebreide inventarisatie van de wensen van architecten zelf en door gebruik te maken van inzichten bekend uit de literatuur, hebben we een zevental eigenschappen opgesteld waaraan hulpmiddelen dienen te voldoen. Op basis van deze eigenschappen hebben we vervolgens in twee opeenvolgende praktijkstudies multifunctionele, webgebaseerde IT-omgevingen geïmplementeerd die het mogelijk maken om verschillende typen architectuurkennis efficiënt en op het juiste moment te delen. De evaluaties van deze omgevingen door architecten laten zien dat dit soort 'all-round'-oplossingen uitermate geschikt zijn om het delen van architectuurkennis binnen een organisatie te bevorderen.

Ons onderzoek naar efficiënte ondersteuning voor het delen van architectuurkennis heeft een aantal belangrijke inzichten opgeleverd. Een van deze inzichten is dat architecten naast het maken van architectuuroplossingen zich met veel andere kennisintensieve activiteiten bezig blijken te houden. Dit inzicht is op zichzelf niet heel verrassend, maar dit betekent wel dat kennismanagementondersteuning voor architecten zich moet richten op al deze activiteiten en niet beperkt moet blijven tot bijvoorbeeld een hulpmiddel om architectuurmodellen te maken.

Een ander belangrijk inzicht is dat niet alle architectuurkennis eenvoudig expliciet te maken is. Kennismanagementonderzoekers, en zeker degenen die zich richten op het IT-domein, hebben vaak de neiging om de oplossing voor kennisdeling te zoeken in sjablonen, databanken of andere opslagmechanismen waarmee de kennis in een bepaalde vorm gegoten wordt en vervolgens eenvoudig te hergebruiken is. Dit principe heet 'codificatie' van kennis. Veel expertkennis blijkt echter heel lastig om op deze manier vast te leggen. Voor het delen van dit soort kennis is het beter om een andere kennismanagementstrategie te volgen. Dit noemen we 'personalisatie' van kennis, wat inhoudt dat niet de kennis zelf opgeslagen wordt maar informatie over de kennisbron, dat wil zeggen de persoon die die kennis bezit. Kennisdeling kan dan plaatsvinden door de persoon te benaderen en hiermee te communiceren. Op deze manier kan bepaalde expertise ook overgedragen worden aan collega's. Uit ons onderzoek blijkt dat architectuurkennis het meest efficiënt gedeeld wordt in een organisatie door gebruik te maken van zowel codificatie- als personalisatiemechanismen. We pleiten daarom voor hybride ondersteuning voor het delen van architectuurkennis en laten in dit proefschrift zien welke voordelen zo'n strategie heeft in de praktijk.

Daar waar architecten verantwoordelijk zijn voor het ontwerpen van een software-architectuur, zijn auditors verantwoordelijk voor een kwaliteitsanalyse van het uiteindelijke softwareproduct. Voor deze taak is het ontdekken van relevante architectuurkennis in bestaande documenten en auditspecifieke informatiebronnen van belang. Immers, de auditor zal de bestaande, gedocumenteerde architectuur moeten vergelijken met de (kwaliteits)eisen van de klant – vaak door eerst concrete maatregelen ('kwaliteitscriteria') te definiëren die al dan niet in het softwareproduct aanwezig moeten zijn. Dit kennisintensieve proces wordt in de regel gekenmerkt door een overdaad aan zowel documentatie als potentiële kwaliteitscriteria.

De al eerder beschreven beslislus maakt duidelijk dat er geen strikt onderscheid gemaakt kan worden tussen 'eisen' en 'ontwerpbeslissingen'. In feite zijn eisen die invloed hebben op de architectuur namelijk ook ontwerpbeslissingen, in die zin dat ze – net als andere ontwerpbeslissingen – randvoorwaarden scheppen voor vervolgbeslissingen. Kwaliteitscriteria bepalen dus ook een architectuur, zij het als ideaalbeeld: ze beschrijven hoe de architectuur van het softwareproduct er uit zou *moeten* zien.

Er is sprake van een zekere mate van herbruikbaarheid van kwaliteitscriteria in verschillende audits. Zo zal voor een willekeurig systeem waarvoor veiligheid een belangrijke kwaliteitseis is vrijwel altijd een vorm van gebruikersauthenticatie (bijvoorbeeld middels een wachtwoord) noodzakelijk zijn. Het vóórkomen van gebruikersauthenticatie is dus een kwaliteitscriterium dat voor al deze systemen geldt.

In een auditorganisatie waar wij het hergebruik van kwaliteitscriteria hebben onderzocht, bleek zulk hergebruik in eerste instantie te bestaan uit het herlezen van eerdere auditrapporten om daarin kwaliteitscriteria toepasbaar binnen een nieuwe audit te identificeren. Om het hergebruik van kwaliteitscriteria beter te ondersteunen ondernam deze organisatie ook pogingen om een elektronische 'catalogus' van kwaliteitscriteria op te zetten, waarin echter geen ruimte was voor onderlinge relaties tussen de criteria. Op basis van een bestaande ontologie van ontwerpbeslissingen hebben wij een ontologie voor kwaliteitscriteria voorgesteld die gebruikt kan worden om kennis omtrent zulke criteria vast te leggen en vervolgens met deze kennis te redeneren, ondersteund door een vernieuwende visualisatie van ontwerpbeslissingen. Middels scenario's hebben we aangetoond hoe deze combinatie van ontologie en visualisatie de auditor bij aanvang van een nieuwe audit ondersteunt bij het ontdekken van relevante kwaliteitscriteria.

De selectie van kwaliteitscriteria is echter pas een eerste stap in een audit. Daarna zal het softwareproduct zelf moeten worden geanalyseerd, om zodoende vast te stellen of en waar het product niet aan de kwaliteitscriteria voldoet. De documentatie die bij een softwareproduct wordt geleverd is voor deze analyse een belangrijke informatiebron. Zulke productdocumentatie beslaat echter meestal tientallen tot soms wel honderden documenten, die door de auditor vaak onder enige tijdsdruk moeten worden bekeken. Het spreekt voor zich dat het onmogelijk is om alle documenten tot in detail te bestuderen. Omdat een leeswijzer vrijwel altijd ontbreekt zijn auditors op hun ervaring en gevoel aangewezen om een selectie te maken tussen teksten die wel tot in detail worden geanalyseerd en teksten die slechts oppervlakkig worden gelezen of ter kennisgeving worden aangenomen.

Uit ons onderzoek blijkt dat het heel goed mogelijk is auditors bij deze selectie te ondersteunen door toepassing van *latent semantic analysis* (LSA), een vorm van geautomatiseerde tekstanalyse. Met behulp van LSA waren we in staat een interactieve leeswijzer op te stellen, waarmee auditors die documenten kunnen vinden die voor hen van belang zijn. Zo willen auditors in de regel aan het begin van een audit snel inzicht krijgen in de architectuur van een softwareproduct, om daarna langzaam maar zeker in te zoomen op verschillende onderdelen van het systeem. Onze leeswijzer wees een *factsheet* van slechts twee pagina's aan als eerst te lezen document. Deze factsheet bleek, hoewel het woord 'architectuur' er niet letterlijk in voorkwam, inderdaad een goede eerste indruk van de architectuur van het betreffende product te geven. Bovendien konden we tegemoet komen aan de wens van de auditors om langzaam af te dalen naar steeds gedetailleerdere informatie door bij vervolgsuggesties rekening te houden met de 'afstand' tussen het eerder gelezen document en nog te lezen documenten.

Om de validiteit van onze leeswijzer te bepalen, hebben we de nabijheid van documenten in het LSA-model vergeleken met de nabijheid van documenten volgens twee auditors. Hiervoor hebben we de individuele mentale modellen bepaald die deze twee auditors van de documenten hadden. Na correlatieberekeningen bleek dat er weliswaar verschillen zijn tussen het mathematische documentenmodel van LSA en de mentale documentenmodellen van de twee auditors, maar dat LSA niet meer afwijkt van de auditors dan de auditors onderling. Mensen blijken dus ook ten opzichte van elkaar op verschillende manieren naar dezelfde verzameling documenten te kijken.

In een vervolgstudie hebben we gekeken naar de verschillende mentale documentenmodellen binnen één ontwikkelteam. Hieruit kwam het beeld naar voren van een bekend kinderspelletje waarin kinderen in een ketting een woord doorfluisteren. Wie dit spel weleens gespeeld heeft weet dat het woord dat het laatste kind hoort nooit hetzelfde is als het woord dat het eerste kind werd ingefluisterd. Zoals het woord in de fluisterketting steeds een beetje verandert, zo veranderde ook het mentale model van de documenten in het ontwikkelteam. Auditors maken geen deel uit van het ontwikkelteam, en staan dus volledig buiten de fluisterketting van documenten. Juist de locatie in deze ketting lijkt het begrip van de softwaredocumentatie te bepalen. Technische oplossingen om auditors te ondersteunen kunnen dan ook niet meer zijn dan ondersteuning: een waardevol hulpmiddel, maar geen eindoplossing. Auditors moeten in contact komen met de softwareleverancier, en praten met leden van het ontwikkelteam. Het ontwikkelen van kwalitatief hoogwaardige software is niet slechts een kwestie van techniek; het is evenzeer een kwestie van mensen.

SIKS Dissertation Series

1998-1	Johan van den Akker (CWI)
	DEGAS - An Active, Temporal Database of Autonomous Objects
1998-2	Floris Wiesman (UM)
	Information Retrieval by Graphically Browsing Meta-Information
1998-3	Ans Steuten (TUD)
	A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
1998-4	Dennis Breuker (UM)
	Memory versus Search in Games
1998-5	E.W.Oskamp (RUL)
	Computerondersteuning bij Straftoemeting
1999-1	Mark Sloof (VU)
	Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products
1999-2	Rob Potharst (EUR)
	Classification using decision trees and neural nets
1999-3	Don Beal (UM)
	The Nature of Minimax Search
1999-4	Jacques Penders (UM)
	The practical Art of Moving Physical Objects
1999-5	Aldo de Moor (KUB)
	Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
1999-6	Niek J.E. Wijngaards (VU)
	Re-design of compositional systems
1999-7	David Spelt (UT)
	Verification support for object database design
1999-8	Jacques H.J. Lenting (UM)
	Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.
2000-1	Frank Niessink (VU)
	Perspectives on Improving Software Maintenance
2000-2	Koen Holtman (TUE)
	Prototyping of CMS Storage Management
2000-3	Carolien M.T. Metselaar (UVA)
	Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.
2000-4	Geert de Haan (VU)
	ETAG, A Formal Model of Competence Knowledge for User Interface Design
2000-5	Ruud van der Pol (UM)
	Knowledge-based Query Formulation in Information Retrieval.
2000-6	Rogier van Eijk (UU)
	Programming Languages for Agent Communication
2000-7	Niels Peek (UU)
	Decision-theoretic Planning of Clinical Patient Management
2000-8	Veerle Coup (EUR)
	Sensitivity Analyis of Decision-Theoretic Networks
2000-9	Florian Waas (CWI)
	Principles of Probabilistic Query Optimization
2000-10	Niels Nes (CWI)
	Image Database Management System Design Considerations, Algorithms and Architecture
2000-11	Jonas Karlsson (CWI)
	Scalable Distributed Data Structures for Database Management
2001-1	Silja Renooij (UU)
	Qualitative Approaches to Quantifying Probabilistic Networks
2001-2	Koen Hindriks (UU)
	Agent Programming Languages: Programming with Mental Models

2001-3	Maarten van Someren (UvA) Learning as problem solving
2001-4	Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-5	Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style
2001-6	Martijn van Welie (VU) Task-based User Interface Design
2001-7	Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization
2001-8	Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9	Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
2001-10	Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design
2001-11	Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design
2002-01	Nico Lassing (VU) Architecture-Level Modifiability Analysis
2002-02	Roelof van Zwol (UT) Modelling and searching web-based document collections
2002-03	Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval
2002-04	Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05	Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
2002-06	Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain
2002-07	Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
2002-08	Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
2002-09	Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems
2002-10	Brian Sheppard (UM) Towards Perfect Play of Scrabble
2002-11	Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications
2002-12	Albrecht Schmidt (Uva) Processing XML in Database Systems
2002-13	Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications
2002-14	Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
2002-15	Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling
2002-16	Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications
2002-17	Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance
2003-01	Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments
2003-02	Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems

2003-03	Martijn Schuemie (TUD)	Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
2003-04	Milan Petkovic (UT)	Content-Based Video Retrieval Supported by Database Technology
2003-05	Jos Lehmann (UVA)	Causation in Artificial Intelligence and Law - A modelling approach
2003-06	Boris van Schooten (UT)	Development and specification of virtual environments
2003-07	Machiel Jansen (UvA)	Formal Explorations of Knowledge Intensive Tasks
2003-08	Yongping Ran (UM)	Repair Based Scheduling
2003-09	Rens Kortmann (UM)	The resolution of visually guided behaviour
2003-10	Andreas Lincke (UvT)	Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture
2003-11	Simon Keizer (UT)	Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
2003-12	Roeland Ordelman (UT)	Dutch speech recognition in multimedia information retrieval
2003-13	Jeroen Donkers (UM)	Nosce Hostem - Searching with Opponent Models
2003-14	Stijn Hoppenbrouwers (KUN)	Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
2003-15	Mathijs de Weerdt (TUD)	Plan Merging in Multi-Agent Systems
2003-16	Menzo Windhouwer (CWI)	Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
2003-17	David Jansen (UT)	Extensions of Statecharts with Probability, Time, and Stochastic Timing
2003-18	Levente Kocsis (UM)	Learning Search Decisions
2004-01	Virginia Dignum (UU)	A Model for Organizational Interaction: Based on Agents, Founded in Logic
2004-02	Lai Xu (UvT)	Monitoring Multi-party Contracts for E-business
2004-03	Perry Groot (VU)	A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
2004-04	Chris van Aart (UVA)	Organizational Principles for Multi-Agent Architectures
2004-05	Viara Popova (EUR)	Knowledge discovery and monotonicity
2004-06	Bart-Jan Hommes (TUD)	The Evaluation of Business Process Modeling Techniques
2004-07	Elise Boltjes (UM)	Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes
2004-08	Joop Verbeek(UM)	Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise
2004-09	Martin Caminada (VU)	For the Sake of the Argument; explorations into argument-based reasoning
2004-10	Suzanne Kabel (UVA)	Knowledge-rich indexing of learning-objects
2004-11	Michel Klein (VU)	Change Management for Distributed Ontologies

SIKS Dissertation Series

2004-12	The Duy Bui (UT) Creating emotions and facial expressions for embodied agents
2004-13	Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play
2004-14	Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium
2004-15	Arno Knobbe (UU) Multi-Relational Data Mining
2004-16	Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning
2004-17	Mark Winands (UM) Informed Search in Complex Games
2004-18	Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models
2004-19	Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval
2004-20	Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams
2005-01	Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications
2005-02	Erik van der Werf (UM)) AI techniques for the game of Go
2005-03	Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language
2005-04	Nirvana Meratnia (UT) Towards Database Support for Moving Object data
2005-05	Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing
2005-06	Pieter Spronck (UM) Adaptive Game AI
2005-07	Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems
2005-08	Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications
2005-09	Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages
2005-10	Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments
2005-11	Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search
2005-12	Csaba Boer (EUR) Distributed Simulation in Industry
2005-13	Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen
2005-14	Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics
2005-15	Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes
2005-16	Joris Graaumans (UU) Usability of XML Query Languages
2005-17	Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components
2005-18	Danielle Sent (UU) Test-selection strategies for probabilistic networks
2005-19	Michel van Dartel (UM) Situated Representation
2005-20	Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives

2005-21	Wijnand Derks (UT)
	Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics
2006-01	Samuil Angelov (TUE)
	Foundations of B2B Electronic Contracting
2006-02	Cristina Chisalita (VU)
	Contextual issues in the design and use of information technology in organizations
2006-03	Noor Christoph (UVA)
	The role of metacognitive skills in learning to solve problems
2006-04	Marta Sabou (VU)
	Building Web Service Ontologies
2006-05	Cees Pierik (UU)
	Validation Techniques for Object-Oriented Proof Outlines
2006-06	Ziv Baida (VU)
	Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling
2006-07	Marko Smiljanic (UT)
	XML schema matching – balancing efficiency and effectiveness by means of clustering
2006-08	Eelco Herder (UT)
	Forward, Back and Home Again - Analyzing User Behavior on the Web
2006-09	Mohamed Wahdan (UM)
	Automatic Formulation of the Auditor’s Opinion
2006-10	Ronny Siebes (VU)
	Semantic Routing in Peer-to-Peer Systems
2006-11	Joeri van Ruth (UT)
	Flattening Queries over Nested Data Types
2006-12	Bert Bongers (VU)
	Interactivation - Towards an e-cology of people, our technological environment, and the arts
2006-13	Henk-Jan Lebbink (UU)
	Dialogue and Decision Games for Information Exchanging Agents
2006-14	Johan Hoorn (VU)
	Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change
2006-15	Rainer Malik (UU)
	CONAN: Text Mining in the Biomedical Domain
2006-16	Carsten Riggelsen (UU)
	Approximation Methods for Efficient Learning of Bayesian Networks
2006-17	Stacey Nagata (UU)
	User Assistance for Multitasking with Interruptions on a Mobile Device
2006-18	Valentin Zhizhkun (UVA)
	Graph transformation for Natural Language Processing
2006-19	Birna van Riemsdijk (UU)
	Cognitive Agent Programming: A Semantic Approach
2006-20	Marina Velikova (UvT)
	Monotone models for prediction in data mining
2006-21	Bas van Gils (RUN)
	Aptness on the Web
2006-22	Paul de Vrieze (RUN)
	Fundaments of Adaptive Personalisation
2006-23	Ion Juvina (UU)
	Development of Cognitive Model for Navigating on the Web
2006-24	Laura Hollink (VU)
	Semantic Annotation for Retrieval of Visual Resources
2006-25	Madalina Drugan (UU)
	Conditional log-likelihood MDL and Evolutionary MCMC
2006-26	Vojkan Mihajlović (UT)
	Score Region Algebra: A Flexible Framework for Structured Information Retrieval
2006-27	Stefano Bocconi (CWI)
	Vox Populi: generating video documentaries from semantically annotated media repositories
2006-28	Borkur Sigurbjornsson (UVA)
	Focused Information Access using XML Element Retrieval

SIKS Dissertation Series

2007-01	Kees Leune (UvT) Access Control and Service-Oriented Architectures
2007-02	Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach
2007-03	Peter Mika (VU) Social Networks and the Semantic Web
2007-04	Jurriaan van Diggelen (UU) Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach
2007-05	Bart Schermer (UL) Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance
2007-06	Gilad Mishne (UVA) Applied Text Analytics for Blogs
2007-07	Natasa Jovanovic' (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings
2007-08	Mark Hoogendoorn (VU) Modeling of Change in Multi-Agent Organizations
2007-09	David Mobach (VU) Agent-Based Mediated Service Negotiation
2007-10	Huib Aldewereld (UU) Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols
2007-11	Natalia Stash (TUE) Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System
2007-12	Marcel van Gerven (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty
2007-13	Rutger Rienks (UT) Meetings in Smart Environments; Implications of Progressing Technology
2007-14	Niek Bergboer (UM) Context-Based Image Analysis
2007-15	Joyca Lacroix (UM) NIM: a Situated Computational Memory Model
2007-16	Davide Grossi (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems
2007-17	Theodore Charitos (UU) Reasoning with Dynamic Networks in Practice
2007-18	Bart Orriens (UvT) On the development an management of adaptive business collaborations
2007-19	David Levy (UM) Intimate relationships with artificial partners
2007-20	Slinger Jansen (UU) Customer Configuration Updating in a Software Supply Network
2007-21	Karianne Vermaas (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005
2007-22	Zlatko Zlatev (UT) Goal-oriented design of value and process models from patterns
2007-23	Peter Barna (TUE) Specification of Application Logic in Web Information Systems
2007-24	Georgina Ramrez Camps (CWI) Structural Features in XML Retrieval
2007-25	Joost Schalken (VU) Empirical Investigations in Software Process Improvement
2008-01	Katalin Boer-Sorbn (EUR) Agent-Based Simulation of Financial Markets: A modular,continuous-time approach
2008-02	Alexei Sharpanskykh (VU) On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03	Vera Hollink (UVA)
	Optimizing hierarchical menus: a usage-based approach
2008-04	Ander de Keijzer (UT)
	Management of Uncertain Data - towards unattended integration
2008-05	Bela Mutschler (UT)
	Modeling and simulating causal dependencies on process-aware information systems from a cost perspective
2008-06	Arjen Hommersom (RUN)
	On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective
2008-07	Peter van Rosmalen (OU)
	Supporting the tutor in the design and support of adaptive e-learning
2008-08	Janneke Bolt (UU)
	Bayesian Networks: Aspects of Approximate Inference
2008-09	Christof van Nimwegen (UU)
	The paradox of the guided user: assistance can be counter-effective
2008-10	Wauter Bosma (UT)
	Discourse oriented summarization
2008-11	Vera Kartseva (VU)
	Designing Controls for Network Organizations: A Value-Based Approach
2008-12	Jozsef Farkas (RUN)
	A Semiotically Oriented Cognitive Model of Knowledge Representation
2008-13	Caterina Carraciolo (UVA)
	Topic Driven Access to Scientific Handbooks
2008-14	Arthur van Bunningen (UT)
	Context-Aware Querying; Better Answers with Less Effort
2008-15	Martijn van Otterlo (UT)
	The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.
2008-16	Henriette van Vugt (VU)
	Embodied agents from a user's perspective
2008-17	Martin Op 't Land (TUD)
	Applying Architecture and Ontology to the Splitting and Allying of Enterprises
2008-18	Guido de Croon (UM)
	Adaptive Active Vision
2008-19	Henning Rode (UT)
	From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search
2008-20	Rex Arendsen (UVA)
	Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven
2008-21	Krisztian Balog (UVA)
	People Search in the Enterprise
2008-22	Henk Koning (UU)
	Communication of IT-Architecture
2008-23	Stefan Visscher (UU)
	Bayesian network models for the management of ventilator-associated pneumonia
2008-24	Zharko Aleksovski (VU)
	Using background knowledge in ontology matching
2008-25	Geert Jonker (UU)
	Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency
2008-26	Marijn Huijbregts (UT)
	Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled
2008-27	Hubert Vogten (OU)
	Design and Implementation Strategies for IMS Learning Design
2008-28	Ildiko Flesch (RUN)
	On the Use of Independence Relations in Bayesian Networks
2008-29	Dennis Reidsma (UT)
	Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

SIKS Dissertation Series

2008-30	Wouter van Atteveldt (VU)
	Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content
2008-31	Loes Braun (UM)
	Pro-Active Medical Information Retrieval
2008-32	Trung H. Bui (UT)
	Toward Affective Dialogue Management using Partially Observable Markov Decision Processes
2008-33	Frank Terpstra (UVA)
	Scientific Workflow Design; theoretical and practical issues
2008-34	Jeroen de Knijf (UU)
	Studies in Frequent Tree Mining
2008-35	Ben Torben Nielsen (UvT)
	Dendritic morphologies: function shapes structure
2009-01	Rasa Jurgelenaite (RUN)
	Symmetric Causal Independence Models
2009-02	Willem Robert van Hage (VU)
	Evaluating Ontology-Alignment Techniques
2009-03	Hans Stol (UvT)
	A Framework for Evidence-based Policy Making Using IT
2009-04	Josephine Nabukenya (RUN)
	Improving the Quality of Organisational Policy Making using Collaboration Engineering
2009-05	Sietse Overbeek (RUN)
	Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality
2009-06	Muhammad Subianto (UU)
	Understanding Classification
2009-07	Ronald Poppe (UT)
	Discriminative Vision-Based Recovery and Recognition of Human Motion
2009-08	Volker Nannen (VU)
	Evolutionary Agent-Based Policy Analysis in Dynamic Environments
2009-09	Benjamin Kanagwa (RUN)
	Design, Discovery and Construction of Service-oriented Systems
2009-10	Jan Wielemaker (UVA)
	Logic programming for knowledge-intensive interactive applications
2009-11	Alexander Boer (UVA)
	Legal Theory, Sources of Law & the Semantic Web
2009-12	Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin)
	Operating Guidelines for Services
2009-13	Steven de Jong (UM)
	Fairness in Multi-Agent Systems
2009-14	Maksym Korotkiy (VU)
	From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)
2009-15	Rinke Hoekstra (UVA)
	Ontology Representation - Design Patterns and Ontologies that Make Sense
2009-16	Fritz Reul (UvT)
	New Architectures in Computer Chess
2009-17	Laurens van der Maaten (UvT)
	Feature Extraction from Visual Data
2009-18	Fabian Groffen (CWI)
	Armada, An Evolving Database System
2009-19	Valentin Robu (CWI)
	Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
2009-20	Bob van der Vecht (UU)
	Adjustable Autonomy: Controling Influences on Decision Making
2009-21	Stijn Vanderlooy (UM)
	Ranking and Reliable Classification
2009-22	Pavel Serdyukov (UT)
	Search For Expertise: Going beyond direct evidence

2009-23	Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment
2009-24	Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations
2009-25	Alex van Ballegooij (CWI) RAM: Array Database Management through Relational Mapping
2009-26	Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services
2009-27	Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web
2009-28	Sander Evers (UT) Sensor Data Management with Probabilistic Models
2009-29	Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications
2009-30	Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage
2009-31	Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text
2009-32	Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors

Bibliography

Abello, J. and F. van Ham. Matrix zoom: A visual interface to semi-external graphs. In *IEEE Symposium on Information Visualization (InfoVis)*, pages 183–190. IEEE, 2004. Cited on page 260.

Ali Babar, M. and I. Gorton. A Tool for Managing Software Architecture Knowledge. In SHARK/ADI'07. Cited on pages 19, 20, 147, 164, and 179.

Ali Babar, M., I. Gorton, and R. Jeffery. Toward a Framework for Capturing and Using Architecture Design Knowledge. Technical Report UNSW-CSE-TR-0513, The University of New South Wales, June 2005. Cited on pages 32 and 131.

Ali Babar, M., I. Gorton, and B. Kitchenham. A Framework for Supporting Architecture Knowledge and Rationale Management. In Dutoit et al. (2006), pages 237–254. Cited on pages 19 and 20.

Ali Babar, M., T. Dingsøyr, P. Lago, and H. van Vliet, editors. *Software Architecture Knowledge Management: Theory and Practice*. Springer, 2009. Cited on pages 12, 351, and 352.

Althoff, K.-D., F. Bomarius, and C. Tautz. Knowledge Management for Building Learning Software Organizations. *Information Systems Frontiers*, 2(3/4):349–367, 2000. Cited on pages 76 and 127.

Argote, L. *Organizational Learning : Creating, Retaining, and Transferring Knowledge*. Kluwer Academic, Boston, 1999. Cited on page 105.

Argyris, C. and D. A. Schön. *Organizational Learning: A Theory of Action Perspective*. Series on Organization Development. Addison-Wesley, 1978. Cited on page 93.

Aurum, A. and C. Wohlin, editors. *Engineering and Managing Software Requirements*. Springer, 2005. Cited on pages 344 and 356.

Aurum, A., R. Jeffery, C. Wohlin, and M. Handzic. *Managing Software Engineering Knowledge*. Springer, 2003. Cited on page 127.

Avgeriou, P., P. Kruchten, P. Lago, P. Grisham, and D. Perry. Sharing and Reusing Architectural Knowledge–Architecture, Rationale, and Design Intent. In *29th International Conference on Software Engineering (ICSE) - Companion*, pages 109–110, Minneapolis, MN, USA, 2007a. IEEE Computer Society. Cited on pages 19 and 20.

Avgeriou, P., P. Kruchten, P. Lago, P. Grisham, and D. Perry. Architectural Knowledge and Rationale - Issues, Trends, Challenges. *ACM SIGSOFT Software Engineering Notes*, 32(4):41–46, 2007b. Cited on pages 159 and 178.

Avison, D., F. Lau, M. Myers, and P. A. Nielsen. Action Research. *Communications of the ACM*, 42(1):94–97, 1999. Cited on pages 50 and 92.

Babu T, L., M. Seetha Ramaiah, T. V. Prabhakar, and D. A. Rambabu. ArchVoc–Towards an Ontology for Software Architecture. In SHARK/ADI'07, page 5. Cited on pages 239 and 291.

Bachmann, F. and P. Merson. Experience Using the Web-Based Tool Wiki for Architecture Documentation. Technical Report CMU/SEI-2005-TN-041, Carnegie Mellon University, Software Engineering Institute, 2005. Cited on pages 161 and 165.

Bachmann, F., L. Bass, and M. Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design. Technical Report CMU/SEI-2003-TR-004, Carnegie Mellon University, Software Engineering Institute, March 2003. Cited on page 239.

Baddoo, N. and T. Hall. Practitioner Roles in Software Process Improvement: An Analysis using Grid Technique. *Software Process improvement and Practice*, 7: 17–31, 2002. Cited on page 299.

Bahsoon, R. Defining Dependable Dynamic Data-Driven Software Architectures. In *IEEE International Conference on Information Reuse and Integration, (IRI)*, pages 691–694, 2007. Cited on pages 19, 20, and 21.

Basili, V. R., G. Caldiera, and D. H. Rombach. The Experience Factory. In *Encyclopedia of Software Engineering*, pages 469–476. John Wiley & Sons Inc., 1994. Cited on page 127.

Baskerville, R. L. Investigating Information Systems with Action Research. *Communications of the AIS*, 2(3es), 1999. Cited on pages 92 and 206.

Bass, L., P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley Pearson Education, Boston, second edition, 2003. Cited on pages 3, 54, 125, 232, 237, and 246.

Bengtsson, P., N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147, 2004. Cited on page 232.

Berry, M. W., S. T. Dumais, and G. W. O'Brien. Using Linear Algebra for Intelligent Information Retrieval. *SIAM Review*, 37(4):573–595, 1995. Cited on page 274.

Berry, M. W., Z. Drmac, and E. R. Jessup. Matrices, Vector Spaces, and Information Retrieval. *SIAM Review*, 41(2):335–362, 1999. Cited on pages 278 and 281.

Biolchini, J., P. G. Mian, A. C. C. Natali, and G. H. Travassos. Systematic Review in Software Engineering. Technical report, COPPE/UFRJ/PESC, May 2005. Cited on page 39.

Boer, N.-I., P. J. van Baalen, and K. Kumar. The Importance of Sociality for Understanding Knowledge Sharing Processes in Organizational Contexts. Technical Report ERS-2002-05-LIS, Erasmus Research Institute of Management (ERIM), Rotterdam School of Management / Faculteit Bedrijfskunde, 2002. Cited on page 130.

de Boer, R. C. and H. van Vliet. Constructing a Reading Guide for Software Product Audits. In WICSA 2007, pages 11–20. Cited on page 19.

Bonnet, E. and Y. V. d. Peer. zt: A Software Tool for Simple and Partial Mantel Tests. *Journal of Statistical Software*, 7(10), 2002. Cited on pages 288 and 302.

Booch, G. Handbook of Software Architecture. `http://www.booch.com/architecture/`, online. Cited on pages 25 and 291.

Bosch, J. Software Architecture: The Next Step. In Oquendo, F., B. Warboys, and R. Morrison, editors, *1st European Workshop on Software Architectures (EWSA)*, volume 3074, LNCS, pages 194–199, St. Andrews, UK, 2004. Springer. Cited on pages 3, 16, 21, 88, 126, 191, and 196.

Bradbury, J. S., J. R. Cordy, J. Dingel, and M. Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In *1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, Newport Beach, California, 2004. Cited on pages 15, 22, and 27.

Bredemeyer, D. and R. Malan. The Role of the Software Architect. Technical Report White paper 12/10/04, Bredemeyer Consulting, 2004. Cited on page 178.

Bresman, H., J. Birkinshaw, and R. Nobel. Knowledge Transfer in International Acquisitions. *Journal of International Business Studies*, 30(3):439–462, 1999. Cited on page 105.

van den Brink, P. *Social, Organization, and Technological Conditions that Enable Knowledge Sharing*. PhD thesis, Technische Universiteit Delft, 2003. Cited on pages 128 and 130.

de Bruin, H. and H. van Vliet. Quality-driven software architecture composition. *Journal of Systems and Software*, 66(3):269–284, 2003. Cited on pages 28 and 29.

Burge, J. and D. Brown. Software Engineering Using RATionale. *Journal of Systems and Software*, 81(3):395–413, 2008. Cited on page 30.

Buschmann, F., R. Meunier, H. Rohnert, and P. Sommerlad. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996. Cited on pages 15 and 25.

Bush, A. A. and A. Tiwana. Designing Sticky Knowledge Networks. *Communications of the ACM*, 48(5):66–71, 2005. Cited on pages 131, 141, and 158.

Capilla, R., F. Nava, S. Pérez, and J. C. Dueñas. A Web-based Tool for Managing Architectural Design Decisions. *ACM SIGSOFT Software Engineering Notes*, 31 (5), 2006. Cited on pages 32, 131, and 164.

Capilla, R., F. Nava1, and J. C. Dueñas. Modeling and Documenting the Evolution of Architectural Design Decisions. In SHARK/ADI'07. Cited on pages 147, 164, and 179.

Caplinskas, A. and O. Vasilecas. Information systems research methodologies and models. In *5th International Conference Computer Systems and Technologies (CompSysTech)*, pages 1–6, Rousse, Bulgaria, 2004. ACM. Cited on page 217.

Carayannis, E. and J. Coleman. Creative System Design Methodologies: the Case of Complex Technical Systems. *Technovation*, 25(8):831–840, 2005. Cited on pages 19 and 21.

Card, S., J. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999. Cited on page 260.

Carlsen, L. Hierarchical Partial Order Ranking. *Environmental Pollution*, 155(2):247–253, 2008. Cited on page 250.

Chen, C. *Information Visualization - Beyond the Horizon*. Springer, 2004. Cited on page 260.

Chen, S. Task Partitioning in New Product Development Teams: A Knowledge and Learning Perspective. *Journal of Engineering and Technology Management*, 22(4): 291–314, 2005. Cited on pages 19 and 21.

Chung, L., D. Gross, and E. Yu. Architectural Design to Meet Stakeholder Requirements. In WICSA1, pages 545 – 564. Cited on page 224.

Clements, P. Origins of Software Architecture Study. `http://www.sei.cmu.edu/architecture/roots.html`, online. Cited on page 2.

Clements, P., R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002. Cited on page 168.

Clements, P., R. Kazman, M. Klein, D. Devesh, S. Reddy, and P. Verma. The Duties, Skills, and Knowledge of Software Architects. In WICSA 2007, page 20. Cited on pages 164, 175, 178, and 179.

Clerc, V. Towards Architectural Knowledge Management Practices for Global Software Development. In SHARK'08. Cited on page 163.

Clerc, V., P. Lago, and H. van Vliet. The Architect's Mindset. In QoSA 2007, pages 231–249. Cited on pages 143 and 196.

Clerc, V., P. Lago, and H. van Vliet. Assessing a Multi-Site Development Organization for Architectural Compliance. In WICSA 2007, page 10. Cited on page 34.

Cohen, J. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, second edition, 1988. Cited on page 302.

Conklin, J. and K. Burgess-Yakemovic. A process-Oriented Approach to Design Rationale. *Human Computer Interaction*, 6:357–391, 1991. Cited on page 127.

Conteh, N. Y., G. Forgionne, W. D. Schulte, and K. J. O'Sullivan. The Merits of a Just-In-Time Knowledge Management (JITKM) Approach to Decision-Making Support. *Journal of Information & Knowledge Management*, 5(4):269–277, 2006. Cited on page 146.

Coplien, J. O. and N. B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice Hall, 2004. Cited on page 15.

Coplien, J. O. and D. C. Schmidt. *Pattern Languages of Program Design*. Addison Wesley, 1995. Cited on page 15.

Cormican, K. and L. Dooley. Knowledge Sharing in a Collaborative Networked Environment. *Journal of Information & Knowledge Management*, 6(2):105–114, 2007. Cited on page 198.

Cummings, J. Knowledge Sharing: A Review of the Literature. Technical report, The World Bank Operations Evaluation Department, 2003. Cited on pages 105 and 127.

Dahlstedt, Å. G. and A. Persson. Requirements Interdependencies: State of the Art and Future Challenges. In Aurum and Wohlin (2005), pages 95–116. Cited on page 233.

Davison, R. M., M. G. Martinsons, and N. Kock. Principles of canonical action research. *Information Systems Journal*, 14(1):65–86, 2004. Cited on page 50.

Deerwester, S., S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science (JASIS)*, 41(6):391–407, 1990. Cited on pages 271 and 273.

Desouza, K. C., Y. Awazu, and P. Baloh. Managing Knowledge in Global Software Development Efforts: Issues and Practices. *IEEE Software*, 23(5):30–37, 2006. Cited on pages 37 and 128.

Diaz-Pace, A., H. Kim, L. Bass, P. Bianco, and F. Bachmann. Integrating Quality-attribute Reasoning Frameworks in the ArchE Design Assistant. In QoSA 2008. Cited on page 239.

Diehl, S. *Software visualization: Visualizing the structure, behaviour, and evolution of software*. Springer, 2007. Cited on page 260.

Dingsøyr, T. and R. Conradi. A Survey of Case Studies of the Use of Knowledge Management in Software Engineering. *International Journal of Software Engineering and Knowledge Engineering*, 12(4):391–414, 2002. Cited on page 35.

Dingsøyr, T. and E. Røyrvik. An Empirical Study of an Informal Knowledge Repository in a Medium-Sized Software Consulting Company. In ICSE 2003, pages 84–92. Cited on page 37.

Dingsøyr, T., P. Lago, and H. van Vliet. Rationale Promotes Learning about Architectural Knowledge. In *8th International Workshop on Learning Software Organizations (LSO)*, Rio de Janeiro, Brazil, 2006. Cited on page 76.

Dueñas, J. C. and R. Capilla. The Decision View of Software Architecture. In *2nd European Workshop on Software Architecture (EWSA)*, Pisa, Italy, 2005. Cited on page 32.

Dutoit, A. H. and B. Paech. Rationale Management in Software Engineering. In Chang, S. K., editor, *Handbook of Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Co., 2001. Cited on page 4.

Dutoit, A. H., R. McCall, I. Mistrík, and B. Paech, editors. *Rationale Management in Software Engineering*. Springer, 2006. Cited on pages 127, 339, and 359.

Dybå, T., T. Dingsøyr, and G. K. Hanssen. Applying Systematic Reviews to Diverse Study Types: An Experience Report. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 225–234. IEEE, 2007. Cited on pages 39 and 41.

Earl, M. Knowledge Management Strategies: Toward a Taxonomy. *Journal of Management Information Systems*, 18(1):215–233, 2001. Cited on page 34.

Eeles, P. The Process of Software Architecting. IBM DeveloperWorks, 2006a. Cited on page 126.

Eeles, P. Characteristics of a Software Architect. IBM DeveloperWorks, 2006b. Cited on pages 110 and 178.

Eeles, P. The Benefits of Software Architecting. IBM DeveloperWorks, 2006c. Cited on page 125.

Englemore, R. and T. Morgan, editors. *Blackboard Systems*. Addison Wesley Pub. Co., 1988. Cited on page 142.

Erfanian, A. and F. Shams Aliee. An Ontology-Driven Software Architecture Evaluation Method. In SHARK'08. Cited on page 239.

Falessi, D., M. Becker, and G. Cantone. Design Decision Rationale: Experiences and Steps Ahead Towards Systematic Use. *ACM SIGSOFT Software Engineering Notes*, 31(5), 2006. Cited on pages 31, 131, 147, and 164.

Fan, W., L. Wallace, S. Rich, and Z. Zhang. Tapping the Power of Text Mining. *Communications of the ACM*, 49(9):77–82, 2006. Cited on pages 142 and 182.

Farenhorst, R. and H. van Vliet. Understanding How to Support Architects in Sharing Knowledge. In SHARK'09. Cited on pages 185 and 199.

Farenhorst, R., P. Lago, and H. van Vliet. Effective Tool Support for Architectural Knowledge Sharing. In *1st European Conference on Software Architecture (ECSA)*, pages 123–138, Madrid, Spain, 2007. Cited on page 19.

Farenhorst, R., J. F. Hoorn, P. Lago, and H. van Vliet. What Architects Do and What They Need to Share Knowledge. Technical Report IR-IMSE-003, VU University Amsterdam, April 2009. Cited on pages 184, 190, 191, and 193.

Finkelstein, A., J. Grundy, A. van den Hoek, I. Mistrik, and J. Whitehead, editors. *Collaborative Software Engineering*. Springer, to appear. Cited on pages 351 and 352.

Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997. Cited on page 15.

Fowler, M., D. Rice, M. Foemmel, E. Hieatt, and R. Mee. *Patterns of Enterprise Application Architecture*. Pearson, 2002. Cited on page 15.

Fransella, F. and D. Bannister. *A Manual for Repertory Grid Technique*. Academic Press, 1977. Cited on pages 282 and 298.

Galster, M., A. Eberlein, and M. Moussavi. Transition from Requirements to Architecture: A Review and Future Perspective. In *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, pages 9–16, 2006. Cited on page 16.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. Cited on page 15.

Garlan, D. and B. Schmerl. The RADAR Architecture for Personal Cognitive Assistance. *International Journal of Software Engineering and Knowledge Engineering*, 17(2), 2007. Cited on page 181.

Georgas, J. C. and R. N. Taylor. Towards a Knowledge-Based Approach to Architectural Adaptation Management. In *1st ACM SIGSOFT Workshop on Self-managed systems*, pages 59–63, Newport Beach, California, 2004. Cited on page 15.

Ghosh, T. Creating Incentives for Knowledge Sharing. Technical report, MIT Sloan School of Management, Cambridge, Massachusetts, USA, 2004. Cited on pages 103, 105, and 127.

Girvan, M. and M. Newman. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002. Cited on page 44.

Glass, R. L., I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44:491–506, 2002. Cited on page 98.

Golub, G. H. and C. F. V. Loan. *Matrix Computations*. The John Hopkins University Press, third edition, 1996. Cited on page 273.

Gorton, I. *Essential Software Architecture*. Springer, Secaucus, NJ, USA, 2006. Cited on page 78.

Habli, I. and T. Kelly. Capturing and Replaying Architectural Knowledge through Derivational Analogy. In SHARK/ADI'07. Cited on pages 19 and 20.

Haldin-Herrgard, T. Difficulties in Diffusion of Tacit Knowledge in Organizations. *Journal of Intellectual Capital*, 1(4):357–365, 2000. Cited on pages 103, 105, 106, and 128.

Hall, H. Input-Friendliness: Motivating Knowledge Sharing Across Intranets. *Journal of Information Science*, 27(3):139–146, 2001. Cited on pages 128 and 130.

Hansen, M. T., N. Nohria, and T. Tierney. What's Your Strategy for Managing Knowledge? *Harvard Business Review*, 77(2):106–116, 1999. Cited on pages 31, 34, 36, 95, and 128.

Harrison, N. B., P. Avgeriou, and U. Zdun. Using Patterns to Capture Architectural Decisions. *IEEE Software*, 24(4):38–45, 2007. Cited on pages 21 and 195.

Hayes, J. H., A. Dekhtyar, and S. K. Sundaram. Improving After-the-Fact Tracing and Mapping: Supporting Software Quality Predictions . *IEEE Software*, 22(6):30–37, 2005. Cited on page 272.

Hofmeister, C., P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America. A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, 80(1):106–126, 2007. Cited on pages 29, 79, 109, 125, 147, 159, 178, 180, and 226.

Holyoak, K. J. and D. Simon. Bidirectional Reasoning in Decision Making by Constraint Satisfaction. *Journal of Experimental Psychology: General*, 128(1):3–31, 1999. Cited on page 230.

Holz, H. J., A. Applin, B. Haberman, D. Joyce, H. Purchase, and C. Reed. Research Methods in Computing: What are they, and how should we teach them? In *ACM SIGCSE Bulletin, Workshop session: ITiCSE-2006 working group reports*, 2006. Cited on pages 98 and 99.

van den Hooff, B. and M. Huysman. Managing Knowledge Sharing: Emergent and Engineering Approaches. *Information and Management*, 46:1–8, 2009. Cited on pages 90 and 208.

Hoorn, J. F. *Software Requirements: Update, Upgrade, Redesign. Towards a Theory of Requirements Change*. PhD thesis, VU University Amsterdam, the Netherlands, 2006. Cited on page 148.

Hoorn, J. F., M. E. Breuker, and E. Kok. Shifts in Foci and Priorities. Different Relevance of Requirements to Changing Goals Yields Conflicting Prioritizations and is Viewpoint-dependent. *Software Process Improvement and Practice*, 11:465–485, 2006. Cited on page 186.

Huysman, M. and D. de Wit. Practices of Managing Knowledge Sharing: Towards a Second Wave of Knowledge Management. *Knowledge and Process Management*, 11(2):81–92, 2004. Cited on page 175.

Huysman, M. and V. Wulf. IT to Support Knowledge Sharing in Communities, Towards a Social Capital Analysis. *Journal of Information Technology*, 21:40–51, 2006. Cited on pages 147 and 159.

ICSE 2003. *25th International Conference on Software Engineering (ICSE)*, Portland, Oregon, USA, 2003. IEEE Computer Society. Cited on pages 344 and 357.

IEEE 1471. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. Std, 2000. Cited on pages 16, 56, 59, 62, 70, 71, and 109.

ISO/IEC 14598-1. Information technology - Software product evaluation - Part 1: General overview. International Standard, 1999. Cited on page 211.

ISO/IEC 14598-5. Information technology - Software product evaluation - Part 5: Process for evaluators. International Standard, 1998. Cited on page 291.

ISO/IEC 9126-1. Software engineering - Product quality - Part 1: Quality model. International Standard, 2001. Cited on pages 67, 213, 242, 243, and 291.

ISO/IEC WD3 42010. Systems and Software Engineering – Architectural Description. Draft International Standard, 2009. Cited on page 196.

Jackson, M. *Problem Frames: Analyzing and structuring software development problems*. ACM Press Books, Addison-Wesley, 2001. Cited on pages 222, 223, and 227.

Jankowicz, D. and L. Thomas. An Algorithm for the Cluster Analysis of Repertory Grids in Human Resource Development. *Personnel Review*, 11(4):15–22, 1982. Cited on pages 283 and 299.

Jansen, A. *Architectural Design Decisions*. PhD thesis, Rijksuniversiteit Groningen, 2008. Cited on page 4.

Jansen, A. and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In WICSA 2005, pages 109–120. Cited on page 16.

Jansen, A., J. S. van der Ven, P. Avgeriou, and D. K. Hammer. Tool Support for Architectural Decisions. In WICSA 2007, page 4. Cited on pages 29, 131, 147, 164, and 179.

Johnson, S. C. Hierarchical Clustering Schemes. *Psychometrika*, 32(3):241–254, 1967. Cited on page 285.

Kankanhalli, A., B. C. Y. Tan, and K.-K. Wei. Contributing Knowledge to Electronic Knowledge Repositories: An Empirical Investigation. *MIS Quarterly*, 29(1):113–143, 2005. Cited on pages 37 and 127.

Karapanos, E. and J.-B. Martens. Characterizing the Diversity in Users' Perceptions. In *INTERACT*, volume 4662 of *LNCS*, pages 515–518. Springer, 2007. Cited on page 299.

Kazman, R., L. Bass, M. Webb, and G. Abowd. SAAM: A Method for Analyzing the Properties of Software Architectures. In *16th International Conference on Software Engineering (ICSE)*, pages 81–90, Sorrento, Italy, 1994. IEEE Computer Society / ACM Press. Cited on page 232.

Kazman, R., M. Barbacci, M. Klein, S. J. Carrièrre, and S. G. Woods. Experience with Performing Architecture Tradeoff Analysis. In *21st International Conference on Software Engineering (ICSE)*, pages 54–63, Los Angeles, CA, USA, 1999. ACM. Cited on page 232.

Keim, D., G. Andrienko, J.-D. Fekete, C. Goerg, J. Kohlhammer, and G. Melancon. Visual analytics: Definition, process, and challenges. In *Information Visualization - Human-Centered Issues and Perspectives (eds. A. Kerren* et al.*)*, pages 154–175. Springer, 2008. Cited on page 261.

Kelly, G. A. *The Psychology of Personal Constructs*. Norton, New York, 1955. Cited on pages 282 and 297.

Kerschberg, L. and H. Jeong. Just-in-Time Knowledge Management. In *Professional Knowledge Management*, volume 3782, pages 1–18. Springer Berlin / Heidelberg, 2005. Cited on page 146.

Kitchenham, B. Procedures for Performing Systematic Reviews. Technical Report TR/SE-0401, Keele University / NICTA, July 2004. Cited on page 39.

Kitchenham, B. and S. Pfleeger. Principles of Survey Research, Parts 1 to 6. *ACM SIGSOFT Software Engineering Notes*, 2001-2002. Cited on pages 100 and 184.

Kitchenham, B. A., S. L. Pfleeger, D. C. Hoaglin, and J. Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE transactions on Software Engineering*, 28(8), 2002. Cited on page 199.

Klimoski, R. and S. Mohammed. Team Mental Model: Construct or Metaphor? *Journal of Management*, 20(2):403–437, 1994. Cited on page 300.

Kock, N. The three threats of action research: a discussion of methodological antidotes in the context of an information systems study. *Decision Support Systems*, 37(2), 2004. Cited on page 206.

Kozaczynski, W. Requirements, Architectures and Risks. In *10th Anniversary Joint IEEE International Requirements Engineering Conference (RE)*, page 6, 2002. Cited on page 226.

Kristen, G. *Object Orientation: The KISS Method*. Academic Service, 1995. Cited on page 51.

Kruchten, P. The Architects - The Software Architecture Team. In WICSA1, pages 565–584. Cited on page 195.

Kruchten, P. An Ontology of Architectural Design Decisions in Software-Intensive Systems. In *2nd Groningen Workshop on Software Variability Management*, Groningen, NL, 2004. Cited on pages 30, 70, 73, 225, 233, 235, 239, 240, and 262.

Kruchten, P. What do Software Architects Really do? *The Journal of Systems and Software*, 81:2413–2416, 2008. Cited on page 177.

Kruchten, P., P. Lago, H. van Vliet, and T. Wolf. Building up and Exploiting Architectural Knowledge. In WICSA 2005, pages 291–292. Cited on pages 19 and 20.

Kruchten, P., P. Lago, and H. van Vliet. Building Up and Reasoning About Architectural Knowledge. In QoSA 2006, pages 43–58. Cited on pages 16, 19, and 20.

Kruchten, P., H. Obbink, and J. Stafford. The Past, Present, and Future for Software Architecture. *IEEE Software*, 23(2):22–30, 2006b. Cited on page 19.

Kruchten, P., R. Capilla, and J. C. Dueñas. The Decision View's Role in Software Architecture Practice. *IEEE Software*, 26(2):36–42, 2009. Cited on page 196.

Kunz, W. and H. W. J. Rittel. Issues as Elements of Information Systems. Working Paper 131, Heidelberg-Berkeley, 1970. Cited on page 109.

Lago, P. Establishing and Managing Knowledge Sharing Networks. In Ali Babar et al. (2009). Cited on page 173.

Lago, P. and P. Avgeriou. First Workshop on Sharing and Reusing Architectural Knowledge. *ACM SIGSOFT Software Engineering Notes*, 31(5):32–36, 2006. Cited on pages 19, 20, 22, 29, 31, 126, 178, 180, and 192.

Lago, P., P. Avgeriou, R. Capilla, and P. Kruchten. Wishes and Boundaries for a Software Architecture Knowledge Community. In WICSA 2008, pages 271–274. Cited on page 199.

Lago, P., R. Farenhorst, P. Avgeriou, R. C. de Boer, V. Clerc, A. Jansen, and H. van Vliet. The GRIFFIN Collaborative Virtual Community for Architectural Knowledge Management. In Finkelstein et al. (to appear). Cited on page 78.

Lakshminarayanan, V., W. Liu, C. L. Chen, S. Easterbrook, and D. E. Perry. Software Architects in Practice: Handling Requirements. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 329–332. IBM, 2006. Cited on pages 103 and 127.

van Lamsweerde, A. Goal-Oriented Requirements Engineering: A Guided Tour . In *5th IEEE International Symposium on Requirements Engineering (RE)*, pages 249–263, 2001. Cited on pages 229 and 233.

van Lamsweerde, A. From System Goals to Software Architecture. In Bernardo, M. and P. Inverardi, editors, *Formal Methods for Software Architectures*, volume 2804 of *LNCS*, pages 25–43. Springer, 2003. Cited on pages 21 and 230.

Landauer, T. K. and S. T. Dumais. A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction, and Representation of Knowledge. *Psychological Review*, 104(2):211–240, 1997. Cited on page 271.

Landauer, T. K., P. W. Foltz, and D. Laham. An Introduction to Latent Semantic Analysis. *Discourse Processes*, 25:259–284, 1998. Cited on pages 271, 273, 274, 278, and 283.

Lee, L. and P. Kruchten. Capturing Software Architectural Design Decisions. In Kruchten, P., editor, *Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 686–689, 2007. Cited on page 19.

Lee, L. and P. Kruchten. A Tool to Visualize Architectural Design Decisions. In QoSA 2008, pages 43–54. Cited on page 244.

Leffingwell, D. and D. Widrig. *Managing Software Requirements: A Use Case Approach*. Pearson Education, 2nd edition, 2003. Cited on page 249.

Letsche, T. A. and M. W. Berry. Large-Scale Information Retrieval with Latent Semantic Indexing. *Information Sciences*, 100(1-4):105–137, 1997. Cited on pages 271 and 274.

Levesque, L. L., J. M. Wilson, and D. R. Wholey. Cognitive divergence and shared mental models in software development project teams. *Journal of Organizational Behavior*, 22:135–144, 2001. Cited on page 300.

Liang, P. and P. Avgeriou. Tools and Technologies for Architectural Knowledge Management. In Ali Babar et al. (2009). Cited on page 198.

Liang, P., A. Jansen, and P. Avgeriou. Collaborative Software Architecting through Knowledge Sharing. In Finkelstein et al. (to appear). Cited on page 181.

Liebowitz, J. and T. J. Beckman. *Knowledge Organizations: What Every Manager Should Know*. CRC Press, 1998. Cited on page 34.

Lindvall, M., I. Rus, R. Jammalamadaka, and R. Thakker. Software Tools for Knowledge Management: A DACS State-of-the-Art Report. Technical report, Fraunhofer Center for Experimental Software Engineering Maryland and The University of Maryland, 2001. Cited on page 127.

Loza Torres, J. C. *Reusability of Quality Criteria in Software Audits*. Master's thesis, Vrije Universiteit, 2008. Cited on page 244.

Lukka, K. The Constructive Research Approach. In Ojala, L. and O.-P. Hilmola, editors, *Case Study Research in Logistics*, Publications of the Turku School of Economics and Business Administration. Series B 1, pages 83–101. Turku School of Economics and Business Administration, 2003. Cited on page 217.

Mackenzie Owen, J. Tacit Knowledge in Action: Basic Notions of Knowledge Sharing in Computer Supported Work Environments. In *European CSCW Workshop on Managing tacit knowledge*, Bonn, Germany, 2001. Cited on page 213.

MacLean, A., R. Young, V. Bellotti, and T. Moran. Questions, Options and Criteria. In Moran, T. and J. Carroll, editors, *Design Rationale, Concepts, Techniques and Use*, pages 53–106. Lawrence Erlbaum Associates, Mahwah, NJ, 1996. Cited on page 127.

Mader, S. *Wiki Patterns*. Wiley, 2007. Cited on pages 97, 166, and 171.

Maletic, J. I. and A. Marcus. Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding . In *12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 46–53, Vancouver, BC, Canada, 2000. IEEE Computer Society. Cited on page 272.

Maletic, J. I. and N. Valluri. Automatic Software Clustering via Latent Semantic Analysis. In *14th IEEE international conference on Automated Software Engineering (ASE)*, pages 251–254, Cocoa Beach, FL, USA, 1999. IEEE Computer Society. Cited on page 272.

McDermott, R. Why Information Technology Inspired But Cannot Deliver Knowledge Management. *California Management Review*, 41(4):103–117, 1999. Cited on page 36.

Medvidovic, N. and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. Cited on pages 16 and 27.

Medvidovic, N., P. Grünbacher, A. Egyed, and B. W. Boehm. Bridging models across the software lifecycle. *Journal of Systems and Software*, 68(3):199–215, 2003. Cited on page 222.

Moran, T. and J. Carroll. *Design Rationale: Concepts, Techniques, and Use*. Computers, Cognition, and Work. Lawrence Erlbaum Associates, Inc., 1996. Cited on page 76.

Nardi, B. A., D. J. Schiano, M. Gumbrecht, and L. Swartz. Why We Blog. *Communications of the ACM*, 47(12):41–46, 2004. Cited on page 142.

Naur, P. and B. Randell. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, Garmisch, Germany, 1968. Cited on page 2.

Newman, M. and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2), 2004. Cited on page 44.

Niu, N. and S. Easterbrook. So, You Think You Know Others' Goals? *IEEE Software*, 24(2):53–61, 2007. Cited on page 299.

Noblit, G. W. and R. D. Hare. *Meta-Ethnography: Synthesizing Qualitative Studies*. Qualitative Research Methods. Sage Publications, 1988. Cited on pages 17 and 41.

Nonaka, I. and H. Takeuchi. *The Knowledge-Creating Company*. Oxford University Press, 1995. Cited on pages 22, 24, 27, 28, 103, 106, 127, 128, and 300.

Nuseibeh, B. Weaving Together Requirements and Architectures. *IEEE Computer*, 34 (3):115–117, 2001a. Cited on pages 16 and 28.

Nuseibeh, B. Weaving Together Requirements and Architectures. *IEEE Computer*, 34 (3):115–117, 2001b. Cited on page 222.

O'Brien, R. An Overview of the Methodological Approach of Action Research. Technical report, Faculty of Information Studies, University of Toronto, 1998. Cited on page 98.

OMG. Software Process Engineering Metamodel Specification. Technical Report formal/05-01-06, Object Management Group, January 2005a. Cited on pages 56 and 64.

OMG. Unified Modeling Language Specification. Technical report, Object Management Group, August 2005b. Cited on page 59.

Orme, B. K. *Getting Started with Conjoint Analysis: Strategies for Product Design and Pricing Research*. Research Publishers LLC, 2006. Cited on page 208.

Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058, 1972. Cited on page 224.

Perry, D. E. and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992. Cited on page 3.

Perry, D. E., A. A. Porter, and L. G. Votta. Empirical studies of software engineering: a roadmap. In *Conference on The Future of Software Engineering*, pages 345–355, Limerick, Ireland, 2000. Cited on page 199.

Philips. FRS Method. Technical Report PR-TN-2004/00898, Koninklijke Philips Electronics N.V., 2004. Cited on page 51.

Pohl, K. and E. Sikora. Structuring the Co-design of Requirements and Architecture. In Sawyer, P., B. Paech, and P. Heymans, editors, *13th Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, volume 4542 of *LNCS*, pages 48–62, Trondheim, Norway, 2007. Springer. Cited on pages 16 and 28.

Poort, E. R. and P. H. de With. Resolving Requirement Conflicts through Non-Functional Decomposition. In *4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 145–154, Oslo, Norway, 2004. IEEE Computer Society. Cited on page 226.

QoSA 2006. *2nd International Conference on the Quality of Software-Architectures (QoSA)*, Västerås, Sweden, 2007. Springer. Cited on pages 350 and 358.

QoSA 2007. *3rd International Conference on the Quality of Software-Architectures (QoSA)*, Boston, USA, 2007. Springer. Cited on pages 12 and 343.

QoSA 2008. *4th International Conference on the Quality of Software-Architectures (QoSA)*, Karlsruhe, Germany, 2008. Springer. Cited on pages 344 and 352.

van der Raadt, B., S. Schouten, and H. van Vliet. Stakeholder Perception of Enterprise Architecture. In *2nd European Conference on Software Architecture (ECSA)*, volume LNCS 5292, pages 19–34, Paphos, Cyprus, 2008. Springer. Cited on page 178.

Ramesh, B. and M. Jarke. Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001. Cited on page 76.

Ran, A. and J. Kuusela. Design Decision Trees. In *8th International Workshop on Software Specification and Design*, pages 172–175, 1996. Cited on pages 18, 19, 21, and 30.

Rapanotti, L., J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven Problem Decomposition. In *12th IEEE International Requirements Engineering Conference (RE)*, pages 80–89, 2004. Cited on pages 16 and 224.

Regli, W., X. Hu, M. Atwood, and W. Sun. A Survey of Design Rationale Systems: Approaches, Representation, Capture and Retrieval. *Engineering with Computers*, 16(3-4):209–235, 2000. Cited on page 30.

Robbins, J. E., D. M. Hilbert, and D. F. Redmiles. Using critics to analyze evolving architectures. In *Joint proceedings of the second international software architecture* *workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, San Francisco, California, United States, 1996. ACM. Cited on pages 22 and 29.

Roeller, R., P. Lago, and H. van Vliet. Recovering Architectural Assumptions. *Journal of Systems and Software*, 79(4):552–573, 2006. Cited on page 232.

Röll, M. Distributed KM - Improving Knowledge Workers' Productivity and Organisational Knowledge Sharing with Weblog-based Personal Publishing. In *European Conference on Weblogs (Blogtalk 2.0)*, Vienna, 2004. Cited on pages 129 and 130.

Rolland, C. and C. Salinesi. Modeling Goals and Reasoning with Them. In Aurum and Wohlin (2005), pages 189–217. Cited on page 233.

Ruhe, G. Software Engineering Decision Support - A new Paradigm for Learning Software Organizations. In *4th Workshop on Learning Software Organizations (LSO)*, pages 104–113, Chicago, Illinois, USA, 2002. Springer. Cited on pages 76 and 127.

Ruhe, G. and F. Bomarius. Enabling techniques for learning organizations. In *11th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, volume 1756 of *LNCS*, pages 11–16. Springer, 1999. Cited on page 77.

Rus, I. and M. Lindvall. Knowledge Management in Software Engineering. *IEEE Software*, 19(3):26–38, 2002. Cited on pages 76, 88, and 127.

Salton, G. and C. Buckley. Term-Weighting Approaches in Automatic Text Retrieval. *Information Processing & Management*, 24(5):513–523, 1988. Cited on page 278.

Savolainen, J. and J. Kuusela. Framework for Goal Driven System Design. In *26th Annual International Computer Software and Applications Conference*, page 749, 2002. Cited on page 226.

Sawyer, P. and N. Maiden. How to Use Web Services in Your Requirements Process. *IEEE Software*, 26(1):76–78, 2009. Cited on page 239.

Sawyer, P., P. Rayson, and K. Cosh. Shallow knowledge as an aid to deep understanding in early phase requirements engineering. *IEEE Transactions on Software Engineering*, 31(11):969–981, 2005. Cited on page 290.

Schaffert, S., F. Bry, J. Baumeister, and M. Kiesel. Semantic Wikis. *IEEE Software*, 25(4):8–11, 2008. Cited on page 168.

Schmidt, D. C. and F. Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In ICSE 2003. Cited on page 25.

Schmidt, D. C., M. Fayad, and R. E. Johnson. Software patterns. *Communications of the ACM*, 39(10):37–39, 1996. Cited on page 25.

Schuster, N., O. Zimmermann, and C. Pautasso. ADkwik: Web 2.0 Collaboration System for Architectural Decision Engineering. In *19th International Conference on Software Engineering & Knowledge Engineering (SEKE)*, Skokie, USA, 2007. Cited on pages 161 and 165.

Seaman, C. B., M. G. Mendonça, V. R. Basili, and Y.-M. Kim. User Interface Evaluation and Empirically-Based Evolution of a Prototype Experience Management Tool. *IEEE Transactions on Software Engineering*, 29(9):838–850, 2003. Cited on page 127.

SEONTOLOGY project team. The Software Engineering Ontology, http://www.seontology.org/. Cited on page 291.

SHARK'08. *3rd international workshop on SHAring and Reusing architectural Knowledge (SHARK)*, Leipzig, Germany, 2008. ACM. Cited on pages 11, 343, and 345.

SHARK'09. *4th Workshop on SHAring and Reusing architectural Knowledge (SHARK)*, Vancouver, Canada, 2009. IEEE Computer Society. Cited on pages 102, 219, and 345.

SHARK/ADI'07. *2nd Workshop on SHAring and Reusing architectural Knowledge - Architecture, rationale, and Design Intent (SHARK/ADI)*, Minneapolis, MN, USA, 2007. IEEE Computer Society. Cited on pages 11, 339, 340, 342, and 347.

Shaw, M. and P. Clements. The Golden Age of Software Architecture. *IEEE Software*, 23(2):31–39, 2006. Cited on page 321.

Shaw, M. and P. Clements. "The Golden Age of Software Architecture" Revisited. *IEEE Software*, 26(4):70–72, 2009. Cited on page 321.

Shaw, M. and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996. Cited on page 3.

Shaw, M. L. G. and B. R. Gaines. Comparing conceptual structures: consensus, conflict, correspondence and contrast. *Knowledge Acquisition*, 1(4):341–363, 1989. Cited on pages 299 and 302.

Shekaran, C., D. Garlan, M. Jackson, N. R. Mead, C. Potts, and H. B. Reubenstein. The Role of Software Architecture in Requirements Engineering. In *First International Conference on Requirements Engineering (ICRE)*, pages 239–245, 1994. Cited on page 221.

Software Engineering Institute. Published Software Architecture Definitions. `http://www.sei.cmu.edu/architecture/published_definitions.html`, online. Cited on page 3.

Spence, R. *Information visualization: Design for interaction.* Prentice Hall, 2nd edition, 2007. Cited on page 250.

Susman, G. I. and R. D. Evered. An Assessment of the Scientific Merits of Action Research. *Administrative Science Quarterly*, 23(4):582–603, 1978. Cited on pages 50 and 92.

Swan, J., H. Scarbrough, and J. Preston. Knowledge Management - The Next Fad to Forget People? In *7th European Conference on Information Systems (ECIS)*, Copenhagen, Denmark, 1999. Cited on page 36.

Swartout, W. and R. Balzer. On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–440, 1982. Cited on page 222.

Szabó, G. *Visualization of Complex Quality Criteria – Tool Support for the Software Audit Process*. Master's thesis, VU University Amsterdam, 2008. Cited on page 261.

Tang, A., M. Ali Babar, I. Gorton, and J. Han. A Survey of the Use and Documentation of Architecture Design Rationale. In WICSA 2005, pages 89–98. Cited on page 30.

Tang, A., Y. Jin, and J. Han. A Rationale-Based Architecture Model for Design Traceability and Reasoning. *The Journal of Systems and Software*, 80(6):2007, 2007. Cited on pages 30 and 147.

Tyree, J. and A. Akerman. Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22(2):19–27, 2005. Cited on pages 16, 30, 56, 71, 72, and 116.

van der Ven, J. S., A. Jansen, P. Avgeriou, and D. K. Hammer. Using Architectural Decisions. In QoSA 2006. Cited on page 107.

van der Ven, J. S., A. Jansen, J. Nijhuis, and J. Bosch. Design decisions: The Bridge between Rationale and Architecture. In Dutoit et al. (2006), pages 329–346. Cited on pages 4 and 16.

Vitruvius Pollo, M. De Architectura libri decem, ca. 80 - ca. 20 BC. Cited on page 2.

van Vliet, H. *Software Engineering: Principles and Practice*. John Wiley & Sons Ltd, West Sussex, England, third edition, 2008. Cited on page 1.

WICSA 2005. *5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Pittsburgh, Pennsylvania, USA, 2005. IEEE Computer Society. Cited on pages 349, 350, and 358.

WICSA 2007. *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Mumbai, Maharashtra, India, 2007. IEEE Computer Society. Cited on pages 219, 341, 343, and 349.

WICSA 2008. *7th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Vancouver, BC, Canada, 2008. IEEE Computer Society. Cited on pages 101 and 351.

WICSA1. *Software Architecture, TC2 First Working IFIP Conference on Software Architecture*, volume 140 of *IFIP Conference Proceedings*, San Antonio, Texas, USA, 1999. Kluwer. Cited on pages 343 and 350.

Wieringa, R. J. and J. M. G. Heerkens. The Methodological Soundness of Requirements Engineering Papers: A Conceptual Framework and Two Case Studies. *Requirements Engineering*, 11(4):295–307, 2006. Cited on page 217.

Young, S. M., H. M. Edwards, S. McDonald, and J. Barrie Thompson. Personality Characteristics in an XP Team: A Repertory Grid Study. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005. Cited on page 300.

Yu, E. S. K. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *3rd IEEE International Symposium on Requirements Engineering (RE)*, pages 226–235, 1997. Cited on page 233.

Zakas, N. C., J. McPeak, and J. Fawcett. *Professional Ajax*. Wiley, 2006. Cited on page 136.

van Zeist, B., P. Hendriks, R. Paulussen, and J. Trienekens. Het Extended ISO-Model voor Softwarekwaliteit. In *Kwaliteit van softwareprodukten*, pages 97–156. Kluwer Bedrijfsinformatie B.V., Deventer, 1996. Cited on pages 213 and 246.

Zhuge, H. *The Knowledge Grid.* World Scientific Publishing Co., 2004. Cited on page 78.