TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Veli-Pekka Eloranta

Techniques and Practices for Software Architecture Work in Agile Software Development

Julkaisu 1293 • Publication 1293

Tampere 2015

Tampereen teknillinen yliopisto. Julkaisu 1293
Tampere University of Technology. Publication 1293

Veli-Pekka Eloranta

Techniques and Practices for Software Architecture Work in Agile Software Development

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 8th of May 2015, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2015

ISBN 978-952-15-3507-9 (printed)
ISBN 978-952-15-3511-6 (PDF)
ISSN 1459-2045

ABSTRACT

Eloranta, Veli-Pekka
Techniques and Practices for Software
Architecture Work in Agile Software Development

Keywords: software architecture, agile software development, software architecture knowledge management, software architecture evaluation, software architecture knowledge repository

Since the publishing of Agile Manifesto in 2001, the agile software development has taken the world by storm. Agile software development does not emphasize the importance of software architecture, even though software architecture is often seen as a key factor for achieving the quality goals set for the software system. It has been even said that agile software development and software architecture are a clash of two cultures.

In many software projects there is no need to consider software architecture anymore. For example, when designing a mobile application, the ecosystem forces the developer to use certain architecture style provided by the platform. In web development ready-made frameworks and project templates are available offering complete software architecture designs for the application developer.

There are still domains and systems where careful architecture design is needed. When developing complex systems or systems with a long life-cycle, diligent software architecture design is a key to avoid massive rework during the development. It cannot be a coincidence that companies developing these kinds of systems struggle with agile software development the most.

On the one hand, the goal of this thesis was to study software architecture practices used in agile software development in the industry. On the other hand, the goal was to develop new methods and techniques to support incremental software architecture working practices which can be aligned with agile methods such as Scrum. The study applied case study, interviews and design science as the main research methods.

The results show that there are four main ways to carry out software architecture work while using agile methods. Basing on this result, models for aligning software architecture knowledge management were developed. These models can be used as guidelines for selecting the appropriate software architecture practices in an organization.

As a part of the research work, an architecture knowledge repository was developed for sharing the knowledge in agile projects and for automatic software architecture document generation. Additionally, the results of this study show that by taking a decision-centric approach to software architecture evaluation, the evaluation method can be lightweight enough to make incremental evaluation a viable option. Similarly, existing software architecture evaluation methods can be boosted to fit agile software development by utilizing domain knowledge.

PREFACE

This journey began in 2008 in Sulake project and continued in another Tekes funded project: Sulava. During those projects I have met and worked with numerous people who have gave me valuable feedback and input for carrying out the research for this thesis. I would like to express my gratitude to those people.

First and foremost, I want to thank my supervisor Kai Koskimies for his guidance, ideas and support. He has read numerous versions of my publications and gave me very valuable feedback during the process. Especially, I want to point out the freedom Professor Koskimies gave me to do the right thing instead of always following the rules. I want also give a very special thank you to my second supervisor Tommi Mikkonen, who helped me to finish the thesis.

Additionally, I want to thank my co-authors of the publications. Especially I want to thank “team Groningen”: Uwe van Heesch, Paris Avgeriou, Neil Harrison and Rick Hilliard for their support in DCAR work. Furthermore, I want to thank the co-authors of the pattern book: Johannes Koskinen, Marko Leppänen and Ville Reijonen. I am confident that those discussions we had in sauna are reflected in this work as well.

I also want to thank folks from the pattern community. You have been very supportive and I have learned a lot from you guys. Especially I want to thank Jim Coplien for introducing me to patterns and guiding me during the first steps into the world of patterns. I am also grateful to Jim for his insights on Scrum.

This work has been supported by Nokia Foundation, Ulla Tuomisen säätiö and Eemil Aaltosen säätiö.

Finally, I want to thank my wife Aija for supporting me in writing this thesis. Especially during the last months of writing, your support was invaluable. Special thanks to my big brother Tero, who has always encouraged me to pursue post-graduate studies. I also want to thank my parents who have always been there for me. Special thanks my passed away uncle Erkki, who also constantly encouraged me to go further with my research work.

Konkolankatu, Tampere 06.01. 2015

Veli-Pekka Eloranta

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF INCLUDED PUBLICATIONS

This thesis is based on the following publications. The publications are presented as their own chapters in this thesis. In addition, the role of the author of this thesis is discussed paper by paper in the following listing.

I Veli-Pekka Eloranta, Otto Hylli, Timo Vepsäläinen and Kai Koskimies. 2012. TopDocs: Using Software Architecture Knowledge Base for Generating Topical Documents. In *Proceedings Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA) 2012*, IEEE Computer Society, pages 191-195.

> The candidate was the main responsible author for writing this publication. The candidate wrote the most of the content for this paper. Candidate was also responsible for carrying out the actual research for this contribution. The second author of the paper described the implementation details of the TopDocs approach. The third author contributed to the paper by describing TopDocs extension mechanism and giving an example on that. The fourth author was responsible for constructive research setting and guiding the data collection process. This publication is presented as a verbatim copy in Chapter 8.

II Veli-Pekka Eloranta and Kai Koskimies. 2010. Using domain knowledge to boost software architecture evaluation. In *Proceedings of the 4th European conference on Software architecture* (ECSA'10), Editors: Muhammad Ali Babar and Ian Gorton. Springer-Verlag, Berlin, Heidelberg, 319-326.

> The candidate was responsible for the method development and for writing of this publication. The second author helped in the conceptualization of the research work and supported in the writing process of the paper. This publication is presented as a verbatim copy in Chapter 5.

III Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen and Ville Reijonen-2014. Section 4.3: The story of the Patterns in this Book. In: Designing Distributed Control Systems – A Pattern Language approach. John Wiley & Sons, ISBN: 978-1-118-69415-2, pages 86-90.

> The candidate was the responsible author for this section of the book. Other authors provided feedback and suggestions for improvement which was then incorporated in the text by the author. This publication is presented as a verbatim copy in Section 4.1.

IV Veli-Pekka Eloranta and Johannes Koskinen. 2011. Messaging patterns for distributed machine control systems. In *Proceedings of the 16th European Conference on Pattern Languages of Programs* (EuroPLoP '11). ACM, New York, NY, USA, Article 12 , 18 pages.

> The candidate was responsible author for this publication. The candidate wrote all the patterns presented in the publication. The second author gave feedback on the writing and participated in the data gathering, but the contribution was written completely by the candidate. This work is further extended in candidate's contribution [1] where pattern mining process and refined versions of all of the patterns is presented. This publication is presented as a verbatim copy in Section 4.2.

V Veli-Pekka Eloranta and Kai Koskimies. 2013. Software Architecture Practices in Agile Enterprises. Book chapter in *Aligning Enterprise, System, and Software Architectures,* ed. Ivan Mistrik, Antony Tang, Rami Bahsoon and Judith A. Stafford, Elsevier ISBN 978-1466-62199-2, pages 230-249.

> The candidate was responsible for conducting the research for this contribution and was also the main contributor on the publication written basing on the research. The second author contributed to the questionnaire design and to the analysis of the results. This publication is presented as a verbatim copy in Chapter 7.

VI Uwe van Heesch, Veli-Pekka Eloranta, Paris Avgeriou, Kai Koskimies, and Neil Harrison. 2014. DCAR - Decision-Centric Architecture Reviews. *IEEE Software,* IEEE Computer Society, pages 69-76.

> The research leading to this publication was joint work of the first author and the candidate. Both contributed to the design of evaluation method and to the writing of this publication. Other authors contributed to the method design by participating in the case studies where the evaluation method was used in the practice. The other authors also provided feedback and improvement suggestions basing on the industry cases. The publication and the main research, however, were carried out by the first author and the candidate. The distribution of work between the first author and the candidate was equal. This work is further extended in candidate's contribution [2] by describing the method and results in more detail. This publication is presented as a verbatim copy in Chapter 6.

VII Veli-Pekka Eloranta and Kai Koskimies. 2013. Chapter 8 - Lightweight Architecture Knowledge Management for Agile Software Development. Book chapter in *Agile Software Architecture*, Editors: Muhammad Ali Babar, Alan Brown and Ivan Mistrik, ISBN: 978-0-12-407772-0. pages 189-213.

> The candidate was main responsible for conducting the research and writing the publication. The second author guided the study design and contributed to the paper by revising it. This publication is presented as a verbatim copy in Chapter 9.

The permission of the copyright holders of the original publications to republish them in this thesis is hereby acknowledged.

PART I - INTRODUCTION

This part describes the motivation for this study and introduces the research questions, research process and research methods used in this study. The main contributions of the work are also introduced.

1 INTRODUCTION

1.1 Motivation

There exists a plethora of definitions for the term software architecture and no universal definition exists (see e.g. [3]). In standard [4] the software architecture is defined as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. More recent descriptions have defined software architecture as a set of architecture decisions (e.g. [5]) and currently the prevailing perception is that the software architecture decisions are the central part in software architecture work.

Regardless of the selected definition, the software architecture can be seen as the first and the most fundamental description of the software system being built, and as such, a key element in creating a solid and flexible software system. Software architecture is invisible to the end user of the software. *Internal quality* of the system [6] is determined by the quality of software architecture, and *external quality* [6] that can be experienced by the end user, often reflects the internal quality of the system. In other words, software architecture is a key factor for achieving the quality goals set for the software system.

During the last decade, the software architecture community has been focusing on the problem of capturing and utilizing the software architecture knowledge in the design process. The focus has shifted from just documenting the result of the design process to capturing the rationale behind the design – why the system has been designed in a certain way. This kind of information will help the developers to understand the architecture of the system better and guide them while making changes to the system.

The agile software development has taken off in the software industry with a rapid pace after the publication of Agile Manifesto [7]. Nowadays, it is hard to find a company that claims that they do not use agile methods. Especially, Scrum has been popular software development framework in the industry [8]. In Scrum, there is no role named software architect and Scrum does not have a separate architecture design phase.

As each system clearly has a software architecture, undefined way of designing architecture work on agile software development has resulted in a clash of two cultures [9]. Partly this is due to agilists who see that software architecture necessarily leads to big up-front planning and exhaustive documentation, which both are seen as something that do not belong to agile software development. On the other hand, architects may regard agile methods as immature as there is no place for software architecture work.

Surely, the clash of two cultures is emphasized by the fact that nowadays one does not always need to design the high level software architecture. It has already been designed by someone else. For example, when writing a mobile app for iOS, Apple forces the developer to follow the MVC model [10]. Similarly, many web application frameworks already define the most fundamental architecture. Only the lower level design needs to be carried out by the developer or architect.

In traditional software engineering domains, such as the development of desktop software or web systems, agile methods work very well. It is possible to release a new version whenever it is ready to be shipped, even multiple times a day. Many companies have invested in rapid and automatic deployment cycles so that the software can be delivered automatically in an efficient way. In this kind of environment, delivering a bug fix might be carried out in 15 minutes from finding the bug to the deployment of a new version [11]. Agile delivery strategy has also been identified as one of the critical success factors in agile development [12]. Rapid delivery cycles allow fast feedback loops from customers to developers.

In such domain, the software architecture is refactored when needed to allow rapid changes to the functionality of the system. Furthermore, ready-made scalable technologies are often used as a basis of the design. Ready-made technologies also offer some basic architecture documentation. Thus, there is rarely a need for additional in-house architecture documentation. Only interfaces and APIs that the system offers are documented by having interface descriptions, which can be generated from the code.

There are still domains, where the software architecture plays a central role. For example, distributed control systems or medical devices are so complex hardware-dependent systems that the software architecture needs to be carefully designed in advance – at least on a high level. Additionally, these kinds of embedded systems often have a long life cycle. For example, an elevator system might have a life cycle of 20-30 years [13]. While some web system might stand the similar test of time, the hardware of a web system is likely to change. However, the elevator system will probably have the same mechanical parts for the whole life cycle. In addition, often the software is run on the same or similar hardware for the whole life cycle, too. In case of a hardware malfunction, there might not be suitable hardware available anymore; the software system must be prepared for the situation. Thus, the quality and consequently the software architecture of the system become even more important, as the system needs to be prepared for the changes in the environment. In addition, the role of architecture knowledge management becomes more important, as the architecture knowledge should be accessible years from now. In this thesis, machine control systems are used in many cases as the target system category although the presented results can be applied also in other areas where software architecture plays a central role.

The connection between software development paradigms and software architecture has not always been clearly understood. In a traditional waterfall [14] software development, the software architecture has been designed up-front before the implementation. However, Royce's method [14] has been misinterpreted in multiple ways, and iterations between different stages (e.g. between design and implementation) has been neglected. Unfortunately, the agile manifesto has often been misunderstood, too.

Agile Manifesto does not say anything about up-front planning, which is an often heard argument against software architecture work in agile software development. For example, Coplien & Bjørnvig (2010) claim that planning is crucial, but the plan or following the plan is not. You embrace change when it comes, but you still plan up-front what you are going to do. When developing a fairly large system, one often needs to plan up-front as otherwise it would be impossible to start implementing the system. Particularly in the context of Scrum – one of the main agile software development methods - it has been stated that you cannot successfully use it without up-front planning [15].

Similarly, Agile Manifesto does not indicate that one should not write any documents. The manifesto just says that one should emphasize working software over comprehensive documentation. Since the customer or other developers benefit from the actual working software more than from the documentation, the focus should be on working software. However, if it is valuable for the customer or for the other developers, documentation should be created, too. Additionally, the documentation does not have to be comprehensive; it just needs to contain the essential information.

Unfortunately, many of the existing software architecture practices are such that they have been designed for heavy up-front planning mindset. For example, one of the most successful architecture evaluation methods, ATAM [16], is designed to be non-iterative one-shot evaluation that takes several days to carry out. To fully support agile development in domains such as machine control systems, the architecture practices need to be more light-weight and support more broadly the incremental nature of agile development methods.

When the companies use agile software methods, the software architecture work is still being carried out in one way or another. In this thesis, it is studied how companies align agile software development with software architecture work. This thesis aims to study how software architecture work practices, such as software architecture evaluation and software architecture knowledge management can be aligned with agile software development without losing the benefits of both worlds.

1.2 Scope of the Work

This doctoral dissertation discusses several topics in the field of software architecture, software architecture knowledge management, and agile software development. The relations of the contributions of this thesis to the existing body of knowledge are illustrated in Figure 1. The gray bubbles present the contributions in this thesis. The lines between the bubbles show how the contributions are related to white bubbles representing the existing body of knowledge. Greater than and less than signs represent the reading direction. For example, TopDocs is a software architecture knowledge repository.

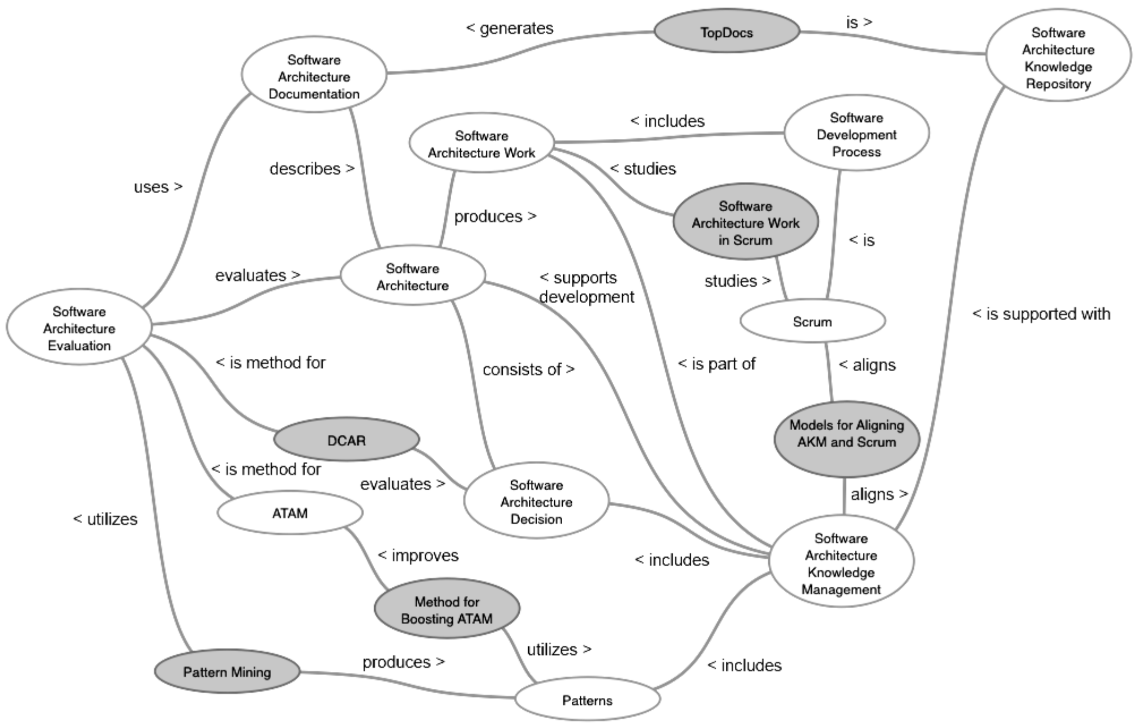


Figure 1 - Relations of the contributions of this thesis to the existing body of knowledge.

Architecture and agile software development have been said to be a clash of two different cultures and aligning them has been recognized to be problematic [17]. On one hand agile methodologies promote deciding on the last responsible moment (see e.g. [18]) and software architecture is seen to be pushing too much to the world of up-front planning and anticipation of things to come. However, architecture work is still carried out in companies using agile software methods. This thesis presents the results of a survey where aligning architecture work and Scrum was studied.

Software architecture evaluation methods such as ATAM (Architecture Trade-Off Analysis Method) [16] are applied to evaluate the capability of the software architecture to meet the goals set for it. This thesis introduces a new decision-centric way to evaluate software architecture. The decision-centric approach allows one to focus on a single architecture decision and re-evaluate the rationale of the decision again and again during the project as new knowledge emerges.

This dissertation also presents an extension to the ATAM evaluation method by describing a way how to boost ATAM by utilizing domain knowledge such as design patterns. This dissertation also describes how design patterns can be recognized during the architecture evaluations and refined to domain-specific patterns that contribute to the architecture knowledge that can be used in the system design. The dissertation introduces TopDocs software architecture knowledge repository that can be used to store design knowledge and generate documentation automatically for a specific stakeholder need.

The overall goal is to identify different ways how to align software architecture and software architecture knowledge management practices with Scrum. The objective is not to present a comprehensive guide of agile software architecture work, but rather to propose various approaches to help in managing software architecture work in agile context.

1.3 Organization of this Dissertation

This dissertation is organized in five parts. Each of the parts contains multiple chapters. The main contributions of this thesis are presented in parts III and IV. These parts present the published papers with some stylizing so that they fit the thesis better. Additionally, repeating parts such as abstracts have been removed along with general information which is already presented in the introductory part of this thesis. In this chapter, the overview of different parts and chapters is presented to help the reader to create overall perception on how this thesis is organized.

Part I – Introduction

In Chapter 1, this part presents an introduction to the software architecture, architecture knowledge management and agile software development and gives motivation why this field is studied in the context of machine control systems. In addition, the structure of the thesis is explained in this chapter. Chapter 2 gives an overview to the work that led to this thesis. The research problems, selected research methods, research process and contributions are discussed in their own sections.

Part II – Setting the landscape

Part II has only one chapter providing the background information required to understand the contributions in parts III and IV. Three themes of background information can be identified in Chapter 3. First, the software architecture work in general is discussed, focusing especially on the software architecture evaluation. Second, software architecture knowledge management is discussed and how domain specific knowledge, i.e. domain-specific patterns can be utilized. Finally, agile software development methods are presented and the clash between agile software development and software architecture is discussed.

Part III – Domain Knowledge & Software Architecture Evaluation

The third part of the thesis presents how domain knowledge can be captured in patterns and how this kind of knowledge can be utilized in software architecture evaluation. Chapter 4 describes how to mine software architecture patterns from the existing systems. Messaging patterns for machine control systems are presented in Chapter 4 as an example of domain-specific patterns which are mined from the systems. These patterns give an example on how to increase domain knowledge quickly. Utilization of domain knowledge and domain-specific patterns in software architecture evaluation is demonstrated in Chapter 5. Chapter 6 takes a different approach to software architecture evaluation and presents a novel method for assessing software architecture decisions. Decisions can again be seen as a specific instance of domain knowledge.

Part IV – Software Architecture Knowledge Management

Part IV addresses the problem of combining software architecture work with agile software development and how software architecture knowledge management could alleviate the surfacing problems. Chapter 7 presents the current ways the practitioners are carrying out architecture work in the industry while using agile methods. Chapter 8 presents TopDocs architecture knowledge base which can be used to generate software architecture documents automatically for a certain stakeholder for a certain need.

Chapter 9 covers the topic of aligning software architecture knowledge management with agile software development methods.

Part V – Closure

The fifth part concludes this thesis. Chapter 11 first revisits the contributions of this thesis by discussing how the research questions were answered in this work. The limitations of this study and future work are presented. Finally, concluding remarks for the thesis are given.

Related work is discussed in detail in Chapters 4-9 where the related research is presented. Additional related work on software architecture and agile software development is presented in Chapter 3.

2 RESEARCH OVERVIEW

2.1 Research Problem

The main goal of this thesis is to research how architecture work and architecture knowledge management can be aligned with Scrum in the domains where the rigor ways of working are required. The main research questions of this study are:

A. How can agile software development and architecture knowledge management be aligned without losing the benefits of both worlds?
B. How can software architecture evaluation be aligned with agile software development?

To study these two problems, the context of a work machine domain is particularly interesting as in this domain there is strong focus on the software architecture and the need for the adoption of agile methods is evident. From these two high level research problems, more concrete and detailed research questions are derived. Concrete research questions A.1 to A.3 concern the main research question A, and questions B.1 and B.2 concern the question B. These more detailed research questions are as follows:

A.1 How to carry out software architecture work in the context of agile software development methods?

> Architecture work and architecture knowledge management still co-exist with agile development methods in the industry. Therefore, to study how architecture work and architecture knowledge management can be aligned with agile software development methods, the status quo needs to be clarified. Once the current way of working is studied, the improvements to the current way of working can be made.

A.2 How could an architecture knowledge repository help to provide up-to-date information for stakeholders and could architecture documentation be replaced with documents generated from the data in an architecture knowledge repository?

> Often the architecture information produced in projects is overly fragmented: meeting memos, Powerpoint presentations, designs memos, wikis, etc. Tradi-

tionally, the most important issues have been written in the documentation. However, Agile Manifesto [7] is often interpreted as stating that no documentation or only minimal documentation should be written. This is not true; documentation should be done if there is value in it. However, in the rapidly changing environment, the documentation gets easily outdated.

Therefore, it needs to be investigated, what kind of architecture knowledge repository could assist the developers in codifying the architecture information, e.g. designs, architecture decisions, day to day memos, presentations, etc. Furthermore, the suitability of an architecture knowledge repository (AKR) for automatically producing documents that correspond to documents written by developers needs to be studied.

A.3 How can general and domain-specific architecture knowledge be collected and presented in reusable form?

Architecture knowledge can be distinguished to different types of architectural knowledge: application-generic vs. application-specific [19], tacit vs. explicit architecture knowledge [20]. Farenhorst and de Boer [21] have combined these two to get four distinct categories.

Application-generic or domain-specific architecture knowledge could be reused from one project to another if it is made explicit. Tacit knowledge needs to be codified until it can be reused. However, traditionally, architecture documentation contains often only application-specific knowledge which is difficult to reuse. Therefore, a way to collect application-specific information and to transform it to application-generic knowledge is required. Furthermore, it needs to be studied how to complement the application-specific information in the architecture documents with application-generic knowledge without increasing the work load of an architect significantly.

B.1 How can software architecture evaluation methods be made more efficient to support agile development methods?

Many of the existing architecture evaluation methods are considered heavyweight and time consuming, as architecture evaluation can take up to two full days [22], [23]. In a two week sprint this means that 20 % of the sprint would be spent on the architecture evaluation. However, the main problem is to decide the correct moment for software architecture evaluation? When are there enough architecture decisions for sensible architecture evaluation, but still possible to make the required changes with reasonable costs?

Thus, there is a need to explore more efficient ways to carry out the existing software architecture evaluation methods. Furthermore, it needs to be investigated if the current evaluation methods can be carried out in an iterative way or if new kinds of methods are needed.

B.2 How to make architecture decisions first class entities in software architecture evaluation to support iterative and incremental evaluation?

The concept of architecture decisions has been adopted by architecture knowledge management community during the last 15 years. The concept captures the rationale of the design decisions which is often missing from traditional architecture documentation [24]. However, the concept of architecture decisions has not been studied as a key element of architecture evaluation. Thus, in this thesis we will study if the design rationale captured in the architecture decisions could be utilized in software architecture evaluation.

2.2 Research Method

In this thesis, multiple research methods have been applied. The research has been carried out using the following methods (in order of importance): case studies, design science, survey research, empirical grounded theory, and action research. In the following subsections each method is briefly summarized. In addition, each subsection explains how the method is applied in this research.

2.2.1 Case Study

Case study [25] [26] [27] is a research method which involves detailed investigation of a single individual, a group, an event, a period or a project [26]. The case or the subject of the study is an instance of a class of contemporary phenomena and the instance is used to study the class. Case study research method is well-suited for software engineering as the objects in the field of SE are often hard to study in isolation. Well-designed case studies provide deep understanding about the phenomena under study. As a downside, case studies do not provide the same results on causal relationships as controlled experiments do [28]. In addition, case studies have been criticized for being hard to generalize the results from and being biased by researchers [29]. However, when proper research practices are applied, the results are trustworthy.

The research leading to this thesis utilized the case study research method in several contributions. The Decision-Centric Architecture Review method (DCAR) was evaluated in industrial context using case studies. A full-scale DCAR was organized in a company manufacturing distributed control systems and the results were analyzed as a case study. Similarly the method to boost ATAM evaluation using general scenarios and design patterns was evaluated as a case study. TopDocs AKR was built in the spirit of design science, but the result was evaluated using case studies to find out how useful the generated architecture documents really were. The pattern mining method developed in publication **III** was applied to the domain of distributed control systems resulting in publication **IV**. The domain can be seen as a case, where the usefulness of the method was studied.

2.2.2 Design Science

Research in software engineering often involves building artifacts, such as prototype systems or developing new methods. Design science is a research paradigm that is essentially aimed at such problem solving. For example, Hevner et al. [30] states that design science can be applied when constructing models, methods and prototype systems. In design science, the research constructs an artifact that aims to improve the effectiveness and efficiency of the organization [30].

March and Smith [31] identify two parts of the design science research: build and evaluate. The build part relies on the existing kernel theories and how a researcher applies them in order to solve the problem at hand ([32], [33]). Then the result of the build phase is evaluated to acquire new information as a basis for the next build cycle.

In the research presented in this thesis, multiple artifacts were constructed in line with the design science paradigm. These artifacts are a method for boosting architecture evaluation using domain knowledge (Chapter 5), the DCAR evaluation method (Chapter 6), TopDocs architecture knowledge repository (Chapter 8), and models to align agile development with architecture knowledge management (Chapter 9). The aforementioned artifacts were first built and then evaluated with case studies and other research methods. After the first case studies, refinements were constructed and they were evaluated again.

2.2.3 Survey Research

The research methods mentioned in the previous subsections were backed up with survey research. Survey research is used to identify the characteristics or opinions of large population of individuals by having a representative sample of the population [34]. Structured interview studies [35] that can be seen as a method for survey research were also utilized in this research.

The research involved the construction of multiple artifacts using design science. Validating and evaluating the benefits and liabilities of the constructed artifacts were challenging through the long-term and exhaustive usage of the artifacts in practice. Thus, case studies were selected to answer specific questions about the artifact usage in practice.

As demonstrated by [36], survey research is a suitable method for finding out how a tool or method is used in industrial setting. Therefore, in this research survey research and specifically structured interviews were used to identify how software architecture work is carried out in agile software development (Chapter 7) facilitating the development of models for aligning architecture knowledge management with agile development methods became possible.

2.2.4 Grounded Theory

Grounded theory [37] is a research method originating from social sciences where the data is collected first without a hypothesis. Data can be collected using various methods such as interviews, observations, or surveys. Then the data is analyzed in order to discover new theories basing on the collected data.

Key issues in the data are marked as *codes* in the data analysis. Codes are grouped into *concepts* and from concepts, *categories* are created. Concepts and categories help the researcher to notice the emerging theory behind the data. So, in a sense the hypothesis is reverse engineered from the data.

The identification of domain-specific patterns can be seen as Glaserian grounded theory study [38]. Hentrich et al. [39] also argue that pattern mining is the Glaserian grounded theory process and they present a method for mining the patterns using this approach. In research reported in this thesis, a similar pattern mining process was constructed. Thus, the patterns found using this method can be seen as a result of Glaserian grounded theory approach (Chapter 4). Such an approach is quite natural to pattern mining, as by definition, the solution must be identified multiple times in practice until it can be called a pattern. Once the researcher has identified a number of instances of the solution from the collected data, the researcher creates the pattern description. Although the pattern description cannot be regarded as a theory per se, it heavily leans on the empirical data grounded to practice and grounded theory can help to systematize the pattern mining process.

2.2.5 Action Research

Action research is a research method where practice is combined with theory by introducing a change to existing processes and by observing the effects. Action research focuses on practical problem solving and tries to identify theoretical results from the action intervention. Action research emphasizes finding out what practitioners do instead of focusing on what practitioners say they do [40]. Thus, it is regarded as the most realistic research setting as the results will be applied exactly in the same setting as the one where problem was found from [41].

In this thesis action research was used during the period when the author worked in a company one day a week for almost one year. The author was participating in the work of an architecture team in a Finnish company in the machine control system domain and was involved in the development of organization’s software architecture documentation practices. Even though action research is not explicitly mentioned in any of the contributions in this thesis, it largely affected the process of constructing TopDocs architecture knowledge repository (Chapter 8) and the design of models for aligning architecture work with agile software development (Chapter 9). Action research was applied in the research as it made it possible for the author to understand the complex organization and the problems the practitioners were facing and thus was seen as a suitable method to approach the problems.

According to Avison et al. [40] action research consists of the following three steps: *Diagnosis of the problem*, *action intervention* and *reflective learning*. Similarly, Susman & Evered [42] present a slightly more refined division of steps: *Diagnosing*, *action planning*, *action taking*, *evaluating* and *specifying learning*.

Problem diagnosis means that the researcher analyses the current situation in the organization to identify the problem. This can be carried out by interviewing stakeholders and by analysing the available material. In this thesis, the diagnosis step revealed that one of the major problems in the organization was outdated architecture documents and their ill-suited structure.

In the action intervention step, the researcher constructs a solution to the identified problem and applies it in the practice with the help of practitioners. Then in the reflective learning step, the results are analysed and lessons learned are documented [40]. In this research, different architecture document structures and different number and types of documents were tried out in the company. Basing on the feedback from the practitioners and users of the documents the type and number of the documents were changed and the structure of the documents was modified. Furthermore, the action research carried out in the company and the lessons learned from it were incorporated in the development of TopDocs approach. The action research had revealed the need for stakeholder specific documents. In addition, the author's time spent in the company also affected the models on integrating architecture knowledge management into agile software development.

2.3 Research Process

In this section, the research process and timeline of the conducted research are presented. The research resulting in the contributions of this thesis was carried out during the time period from 2008 to 2013. Figure 2 illustrates the timeline of the conducted research.

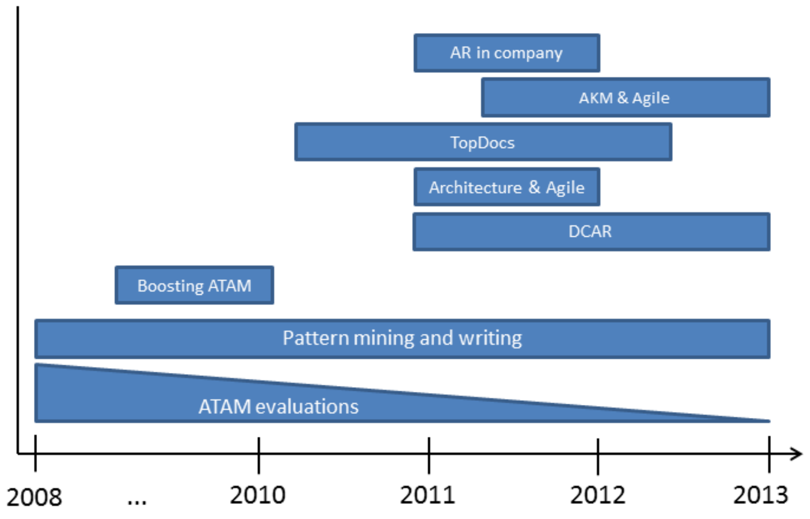


Figure 2 - Timeline of the research conducted in this thesis.

The research leading to this thesis started in 2008 in Tekes funded project Sulake. In project Sulake multiple software architecture evaluations were carried out in the Finnish machine industry by using ATAM [22] evaluation method. ATAM evaluation also exposed the software architectures of the evaluated systems for pattern mining. Patterns for distributed machine control systems were identified during these software architecture evaluations. The patterns were refined and final versions written during the whole research period. The work resulted in a book on design patterns for distributed control systems published separately [1] and to a pattern mining method which is one of the contributions of this thesis.

The experience gained from software architecture evaluation sessions led to a method for boosting ATAM by finding out potential scenario candidates prior to the actual evaluation session. In this way, time can be saved during the evaluation session as shorter time for scenario elicitation is required. The method for boosting ATAM also includes usage of patterns. As ATAM evaluations were carried out in the machine control system domain, it was natural choice to use already identified patterns as examples. Patterns can help the evaluators to find out potential risks in the analysis of the scenarios as patterns describe benefits and liabilities of the solution. Furthermore, by mapping recurring scenarios and patterns together, evaluators can find out which patterns should have been used in the system's design. If those patterns have not been applied in the design, the architect needs to argue why they have not been used in the design. This helps the evaluators to find potential risks in the architecture design. On the other hand, if the patterns have been applied, the consequences section of the pattern might contain some unaddressed negative side effects of the application of the pattern.

In software architecture evaluation a lot of application-specific and domain-specific architecture knowledge is exposed to the stakeholders and to the reviewers. During the evaluations some of that knowledge was captured in the pattern candidates and evaluation reports. However, it was obvious that with proper tooling even more of that exposed information could be captured.

The idea to capture more information during the evaluation led to the development of TopDocs architecture knowledge repository (Chapter 8). In TopDocs, architecture knowledge is stored as small items or granules which conform to the metamodel of the AKR. Input tools can be implemented on the top of the AKR. In this way, most of the architectural knowledge exposed in the software architecture evaluation can be input to the AKR. The input tools make sure that the data input conforms to the metamodel. Conformance on its behalf makes it possible to generate documents for certain stakeholder needs. For example, an architecture evaluation report can be generated from the input architecture knowledge.

As already mentioned, the author worked in a company one day a week studying problems related to software architecture documentation. This action research effort revealed typical problems related to the documentation: documents are not up-to-date; nobody seems to know which is the latest document version or the document does not contain the information stakeholder is looking for, etc. These observations partially contributed to the development of TopDocs. Wider support for different kind of architectural documents was implemented. In addition, an extension mechanism for domain-specific needs, e.g. view for functional safety related issues, was implemented.

Architecture work and agile software development are claimed to be orthogonal [17]. It is said that agile method evangelists see low value in architectural design and the design might be leading to extensive documentation and set of features which are referred to as YAGNI (You Ain't Gonna Need It) [17]. On the other hand, architecture community sees agile practices amateurish, unproven, and limited to very small web-based systems [43]. However, during the action research period the researcher witnessed co-existence of these two worlds as the company was practicing Scrum and had rigor architecture practices. This raised the interest to study how other software companies combine agile software development and architecture work.

The aforementioned question was studied using survey research and structured interviews. The study revealed four different ways of carrying out architecture work in Scrum. In the agile and architecture study we also found out that companies rarely use software architecture evaluation methods although they saw value in the evaluation. The same observation is made by other researchers, too [44]. There have also been some indications that more industry friendly evaluation methods are required [45].

We also wanted to explore ways to use the concept of architecture decisions as a first class entity in architecture evaluation. The candidate collaborated with researchers from University of Groningen as they had a lot of experience with this concept. This work resulted in a new incremental and lightweight architecture evaluation method: Decision-Centric Architecture Reviews or in short DCAR.

Rationale for having a certain way of carrying out architecture work varied from company to company. There was no clear correlation between the domain, project type or agile experience and the way of working with architecture. Thus, it became quite clear that there is no single answer to which way the architecture design should be carried out while using Scrum, but the most suitable way has to be found out through experimenting case by case. However, certain base models can be identified. This work led to the proposal on how to align agile software development with lightweight architecture knowledge management (AKM). This work took architecture knowledge repository such as TopDocs as basis and introduced models when to use architecture evaluations methods such as ATAM or DCAR as a part of Scrum. These models could help developers to store, use and refine architectural information in a transparent and non-disruptive way.

2.4 Contributions

This thesis makes multiple contributions in the field of software architecture research. These contributions are described in the following.

1. A technique to generate software architecture documents for a certain need and stakeholder automatically from architecture knowledge repository.

> This study demonstrates how an architecture knowledge database can be used to generate architecture documents which are tailored for a specific need and a specific stakeholder, automatically. The pre-requisite for such document generation is that the data in the architecture knowledge repository conforms to a meta-model which is also presented in this study. The results show that a generated document is essentially smaller in size than traditional architecture document. This is quite natural as the document does not contain all architectural knowledge. Furthermore, the study revealed that smaller the document was, the more it contained useful material for the stakeholder. This contribution is presented in publication **I** in Chapter 8.

2. A method for finding scenario candidates for scenario-based architecture evaluation method in advance shortening the actual evaluation session.

> This contribution describes a method to utilize domain knowledge, i.e. domain model and patterns to boost up scenario-based software architecture evaluation. The method requires that a few evaluations on the same domain have already been carried out. Thus, it is most beneficial when consecutive evaluations on the same domain or for the same system are carried out. The method is also beneficial if application-generic architecture knowledge is available in pattern format as patterns help the reviewers to find out potential risks in the system. This method is presented in publication **II** in Chapter 5.

3. A method for pattern mining as a side effect of software architecture evaluation and an example set of domain-specific patterns mined from the software architecture evaluations.

> The domain-specific patterns can be utilized in software architecture document generation (contribution 1) or to boost software architecture evaluation (contribution 2). Furthermore, patterns can be used to codify architectural knowledge. The method to mine domain-specific patterns is presented in publication **III**. An example set of domain-specific patterns discovered using the method is presented in publication **IV** in Chapter 4. Comprehensive and more refined set of these patterns can be found from [1].

4. An account of software architecture practices in companies using agile software development methods.

> This study reports how software architecture work is carried out in companies while using agile software development methods such as Scrum. The adoption of these work practices are mapped to the characteristics of the teams and project types. In addition, the compatibility of these practices and Scrum patterns is discussed. The study identifies four different ways how companies have combined Scrum and architecture work practices. This contribution is presented in publication **V** in Chapter 7.

5. Lightweight method to evaluate software architecture based on architecture decisions.

> This contribution introduces a software architecture evaluation method called DCAR. The method uses architecture decisions as a central element to evaluate the design of the system. When compared to existing methods, the benefits of DCAR method are light-weightness and possibility to estimate the coverage of the evaluation. Furthermore, the method is designed to create no waste in terms of lean philosophy, so all artifacts created are utilized in evaluation and can be reused as a part of architecture documentation. The method can be carried out in relatively short time, so it is suitable for iterative and incremental use. The study also reports the experiences from industrial use. This contribution is presented in publication **VI** in Chapter 6.

6. Models for aligning architecture knowledge management with agile software development by using the techniques presented in this thesis.

> This study summarizes how architecture knowledge management and the techniques presented in other contributions of this thesis can be aligned with agile software development methods. The study proposes two different models depending on how the architecture practices are carried in Scrum framework. The contribution describes guidelines for agile architecture knowledge management in from of agile AKM manifesto. This study is presented in publication **VII** in Chapter 9.

In addition to the publications included in the thesis, the topics of this thesis are discussed by the author also in publications , [1], [2], [46], [47] and [48].

PART II - SETTING THE LANDSCAPE

This part will present the background for this study. Software architecture work including software architecture evaluation and documentation is described. Software architecture knowledge management and the concept of software architecture decisions are introduced. In addition, agile software development methods and their relationship to software architecture work are described. Detailed background information is given in the context of each study in Chapters 4-9.

3 BACKGROUND

This chapter summarizes the work related to software architecture work, software architecture knowledge management and agile software development. The chapter provides background information to the research in this thesis. First, Section 3.1 describes architecture work in general. Section 3.2 focuses on software architecture evaluation. While it is part of the architecture work, it has its own section as it is central part of the work in this thesis. Section 3.3 discusses software architecture documentation and Section 3.4 presents architecture knowledge management. Section 3.5 focuses on software architecture decisions and how they are usually documented. Section 3.6 describes agile software development methods. Finally, in Section 3.7 the friction between software architecture work and agile software development is discussed.

3.1 Architecture Work

Software architecture is one of the first descriptions of the software during the development of the software. Often software architecture is thought to satisfy the quality requirements set for the system. For example, software architecture should support the system scalability, extendibility or performance depending on what is considered to be important.

ISO/IEC 42010 architecture description standard [49] defines software architecture as follows: *Software architecture is the fundamental conception of a system in its environment embodied in elements, their relationships to each other and to the environment, and the principles guiding its design and evolution*. In other words, this quite traditional definition states that the software architecture is the design of the system, its association to the environment and the guidelines made for the design.

Traditionally, software architecture design has been documented using different views on the architecture [50]. For example, the well-known 4+1 model [51] focuses on these views. However, during the last 15 years the software architecture community has also recognized the importance of the design rationale and focused more on accompanying the traditional software architecture documentation with the documentation of architecture decisions (see e.g. [52]. In this way, the rationale behind the design would be codified, too. This has also moved the focus of software architecture community to architecture knowledge management. Kruchten [53], for example, defines architecture knowledge as a sum of architecture decisions and the resulting design. We will take a closer look to the field of software architecture knowledge management in Section 3.3.

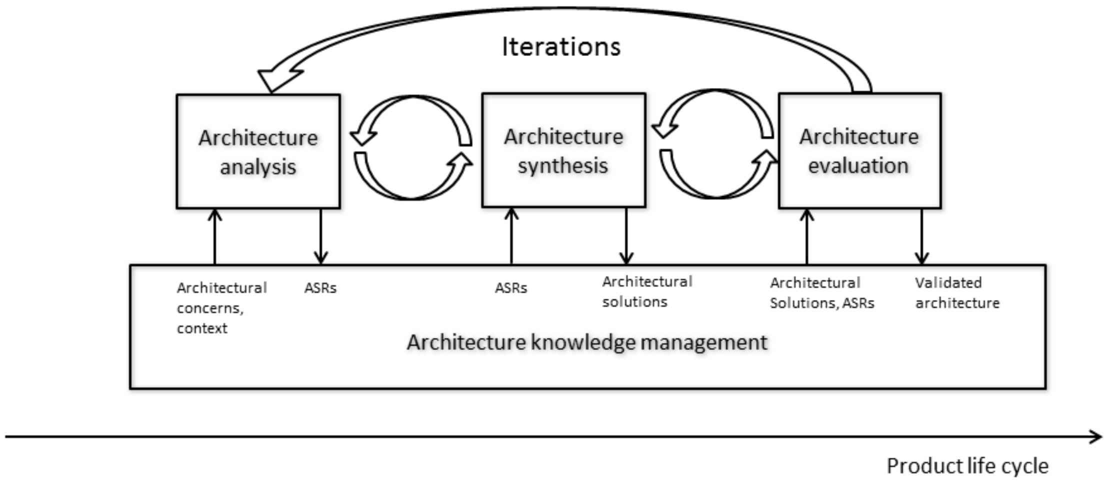


Figure 3 - Architecture work flow and knowledge management

According to Hofmeister [54] architecture work consists of three distinct activities: analysis, synthesis and evaluation. First, requirements for the system and for the software architecture being built are analyzed. Second, the software architecture satisfying these requirements is designed, i.e. synthesized. Third activity is architecture evaluation where the architecture is measured against the architecturally significant requirements. These steps are not sequential but architects move from one phase to another and back [54]. In Figure 3 the arrows between the steps illustrate this iterative and incremental nature of the work. Architecture knowledge management in turn is more covering activity which is carried out in parallel with all other architecture work activities. In Figure 3 the relationship of the architecture knowledge management and the software architecture work is illustrated. In analysis activity, architectural concerns and context are examples of possible inputs and architecturally significant requirements (ASRs) are the output of this activity. ASRs are inputs for architecture synthesis and output of this activity is actual architecture design which is input for architecture evaluation along with ASRs.

3.2 Software Architecture Evaluation

As stated by Hofmeister [54], one of the three basic activities of software architecture work is software architecture evaluation. Software architecture evaluation is the analysis of a system's capability to satisfy the most important stakeholder concerns, based on its large-scale design, or architecture [22]. Software architecture evaluation is an important activity: if the architecture is quickly cobbled together and of poor quality, the software project is more likely to fail [50]. Furthermore, the cost of an architectural change in the early stages of the development is negligible compared to the cost of an architectural change in the later stages of the development [55].

Software architecture evaluations should not be confused with code reviews. In fact, in software architecture evaluation the code is rarely viewed. The goal of architecture evaluation is to find out if made architecture decisions support the quality requirements set for the system and to find out signs of technical debt. In addition, areas of further development can be identified: are there some decisions blocking the development of features in the roadmap.

The most well-known approaches to architecture evaluation are based on scenarios, e.g. SAAM [56], ATAM [16], ALMA (Architecture-level Modifiability Analysis) [57], FAAM (Family-architecture Assessment Method) [58] and ARID (Active Review of Intermediate Designs) [59]. These methods are considered mature: they have been validated in the industry [60] and they have been in use for a long time.

However, even when there are a plethora of different methods available and the benefits of software architecture evaluation are quite clear, many software companies do not regularly conduct architecture reviews [60]. This is partially due to the fact that architecture evaluation is often perceived as complicated and expensive [61]. Another reason for this, concerning especially agile teams, is that many of the architecture evaluation methods are time consuming and require presence of multiple stakeholders. For example, SAAM report [62] shows that SAAM evaluations of 10 different systems ranging from 5 to 100 KLOC required 14 days on average. AT&T reported the cost of 70 staff days per SAAM evaluation [62]. Similarly, medium-sized ATAM evaluation might take up to 70 person-days [22]. Often because of the work load, the evaluation is a one shot activity during the project. In addition, agile teams might find it hard to decide the proper moment for this kind of one-shot evaluation.

While Hofmeister et al. [54] point out that the phases of the software architecture work are iterative, they also mention that there hasn't been much focus on research of iterative architecture evaluation. Even though there is a plethora of different methods for software architecture evaluation, most of the proposed methods are ill-suited for iterative and incremental evaluation. Often, the unfortunate result is that the architecture is not evaluated at all, although there would be benefits that could be gained from the evaluation.

One exception to single-shot approach of software architecture evaluation is a technique called Architecture Software Quality Assurance (aSQA) [63]. aSQA is iterative and incremental and is claimed to support agile software projects. The method fundamentally relies on usage of metrics, but could be carried out also using scenarios or expert judgment. However, the latter option is not validated in industrial context. So, basically aSQA focuses on following metrics that have been prioritized by the stakeholders of the architecture. In addition, levels for metrics are created basing on the stakeholder opinions. This means that once the value of the metrics change the level might be changed to e.g. one of the following: unacceptable, acceptable, and excellent. When the metric is at unacceptable level, actions need to be taken to make it go back to at least acceptable level. While this approach is rather lightweight and continuous, it also minimizes human participation in the evaluation. On the other hand, one of the most important goals of software architecture – facilitating communication between stakeholders – does not necessarily take place.

Documented rationale is recognized as important aspect in ensuring architectural quality [62] as it provides a reasoned explanation why particular architecture decisions have been made. However, often software architecture evaluation methods focus only on quality attributes and the methods do not explicitly encourage capturing other factors, e.g. financial resources or opinions of the developers, that affect architecture decisions. There are still some exceptions. For example, CBAM (Cost Benefit Analysis Method) [64] explicitly takes financial factors and schedule implications into consideration during the analysis of the architecture. However, there are only a few industrial reports [65], [66] on using CBAM evaluation and they all are from the same source.

During software architecture evaluation, architecture knowledge is made explicit as stakeholders argue in favor of their decisions. The body of knowledge that is exposed during software architecture evaluation consists of much more than only quality attributes and their prioritization. Therefore software architecture evaluation is a perfect opportunity to document software architecture efficiently. Capturing the architectural knowledge is extremely important in domains where the life cycles of the systems are long. Of course, some documentation needs to exist in any case also before the evaluation.

To summarize, agile software architecture evaluation should be quick to carry out, allow incremental evaluation, and support efficient documentation of architectural knowledge.

3.3 Software Architecture Documentation

One of the most important cornerstones for software architecture documentation is concept of *views*. Often the architecture is multi-faceted entity that cannot be captured using just a single view. The need for different views comes from various stakeholders that the software architecture has. The developer implementing the system probably requires different information than a person carrying out performance analysis. Thus, often the software architecture documentation is seen as act of documenting the relevant views for the system [67].

One of the most popular frameworks for documenting the software architecture is 4+1 architectural view model which consists of logical, development, process and physical views and of set of scenarios [51]. The views of course can be tailored to better correspond to the system at hand and thus different views can be selected. Using 4+1 model often results in heavy-weight software architecture documentation by definition: there are multiple views to the same subject. Problem is that if and when the documentation gets outdated, there are multiple views to update. This has given documentation a bad rap.

The documentation is written for two purposes: to *remember* and to *communicate* [18], [67]. When documents are written to remember things, it has to be decided for whom the document is for: was it for someone who was there when the documented design and decisions were made or for someone who wasn't there [68]. In the latter case the purpose of documentation is to educate the reader and thus needs to be more comprehensive. This kind of documentation is often the one that would need to be updated time to time.

However, not all architecture documentation is like that. A photo of the design drawn on the whiteboard could serve as a documentation as well as sketch of the architecture on the back of a napkin. Often this kind of documentation serves the purpose of communication: an architect is sketching the architecture to a napkin or whiteboard while explaining it to co-workers and developers. Cockburn [68] argues that this kind of communication is often the most efficient way to transfer knowledge.

The problem with ad hoc light-weight documentation is that it is not usually stored or organized. It is fit for *communication* but not for *remembering*. The knowledge is scattered in various Powerpoint slides, notes, photos and sketches. The knowledge is not organized, searchable or even stored in a single location. Furthermore, while the rationale of design decisions is discussed between the stakeholders when making the decision, the rationale is not made explicit by documenting it and thus in most cases it evaporates [69].

This thesis approaches this problem by introducing architecture knowledge repository, TopDocs, which is described in Chapter 8. TopDocs tries to capture the architecture knowledge as a side effect of other software architecture related activities and aims at producing architecture documents that do not differ radically from ones written manually. While this approach can be applied in any domain, it is especially beneficial in domains where the life cycle of the system is rather long and the value of documentation created for *remembering* is high.

3.4 Software Architecture Knowledge Management

Designing software architecture is a knowledge intensive task: it requires a lot of knowledge on domain, requirements, available technologies, solution models, etc. Furthermore, the information needs to be shared amongst multiple stakeholders. During the last decade the focus area in the software architecture research has expanded to cover software architecture knowledge management, too.

Definitions for software architecture knowledge vary a lot depending on the study and there is no one clear definition for it [70]. According to Boer & Farenhorst [70] the most prominent aspect of various definitions is that software architecture knowledge covers all aspects from the problem definition through design decision to the resulting solution. Basically, this means that in addition to the actual architectural design, software architecture knowledge consists of the problem and of the rationale behind the end result. It answers the question why the architecture design is the way it is. Design in this context means the high-level structure of components and connectors and interactions between the components. Design is traditionally documented using different views, e.g. 4+1 model [51]. So, in addition to that, software architecture knowledge is often thought to contain the design rationale behind the designs. This is also reflected in the definition of software architecture knowledge by [71]: *"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."*

So architectural knowledge consists of the design and all factors affecting the decisions that have been made to create that design. These factors include, but are not limited to: architecturally significant requirements, budget constraints, business goals, human resources, expertise of developers, opinions, etc. Remarkable is that some of the factors such as opinions of the architect and expertise of developers, constituting the architectural knowledge, are not very explicit. Nonaka & Takeuchi [20] draw distinction between tacit and explicit knowledge. The same distinction is also applicable to software architecture knowledge – some types of the architectural knowledge such as requirements are often explicitly stated while architect's bias towards open source software might be unconscious.

Knowledge management is the process of capturing, developing, sharing, protecting and effectively using knowledge [72]. Similarly architecture knowledge management is capturing, developing, using and sharing architecture knowledge [73]. For knowledge management there are two main strategies [74]: *Codification* and *Personalization*.

The codification strategy bases on systematically storing the knowledge and making it available for people in the company. Personalization strategy instead relies on supporting the flow of information between individuals, e.g. by offering a source of information on who knows about which technology within a company. The work in this thesis focuses mostly on codification strategy as this research offers ways to capture, store and share architecture knowledge. However, TopDocs approach offers also support for personalization strategy to some extent as it keeps track of all contributions to the AKR and other users can find out from AKR who has edited the information granules. In this way, one can find out from whom to ask questions related to the task at hand.

Earl [75] categorizes the knowledge management work on high level to three different schools: *Technocratic, Economic,* and *Behavioral*.

Technocratic school focuses on information technology or management to support knowledge-intensive work. Economic school focuses on creating revenue from exploitation of knowledge. Behavioral school focuses on orchestrating knowledge sharing in organizations.

The knowledge management approach used in this thesis can be seen as part of systems school which is a subcategory of technocratic school [75]. The key principle in systems school is to capture the knowledge in knowledge bases or repositories from where other people can access the knowledge. The TopDocs approach (Chapter 8) can be seen as an architecture knowledge repository where architectural knowledge is captured and shared between developers. When using TopDocs, sharing can be carried out using generated architecture documents or by searching the repository.

The idea to store architectural knowledge in a knowledge base is not novel. There have been multiple earlier efforts, for example ADDSS [76] and PAKME [77]. For comparison of different architecture knowledge management tools, one should refer to e.g. [78].

Novelty of the TopDocs approach is provided for the user as an architecture document that is created on-demand from information granules that conform to the metamodel of the architecture knowledge repository. As the granules conform to the metamodel, it is also possible to target the views to the information for various stakeholders. These views can be exported also as traditional architecture and design documents, so the end user can basically decide how she wants to consume the information.

Architecture knowledge repository solves the problems of storing architectural information, maintaining it in a single up-to-date storage, and sharing the information easily. It doesn’t answer the question of how to efficiently capture the information as a side effect of other activities. On the contrary, it requires that developers use AKR and separately input the information to this system. In fact, it has been said that the problem with AKR approach often is that the repositories are not used in practice [79]. They are used for a while and data is input to the system, but the information is not managed resulting in information junkyards [79]. When carrying out the research leading to this thesis (see Chapter 9 and [46]), we noticed that especially capturing the information has to be as easy and as invisible to the developer as possible or the system will not be used at all. If the capturing of architecture knowledge would disrupt with the other work, the knowledge base would not be used.

Traditionally architecture design knowledge has been captured in software architecture design documents and UML diagrams. One of the major problems of this approach is that the documents tend to get outdated while the newest information might be available only in meeting memos, Powerpoint presentations and e-mails written to colleagues. One cause for this is that the architecture knowledge is naturally produced granularly.

For example, a decision to use a two-staged boot loader could be made in a design meeting and meeting notes are sent to the participants. Now it would be an extra effort to update the architecture documentation and it feels like an extra burden to update the document because people involved in the decision already know the outcome of the decision. Thus, the document gets outdated, although pros and cons of the decision are discussed in the meeting. Even if somebody updates the document to correspond the current design, the rationale behind the design is rarely documented.

Tyree and Akerman [80] have proposed a template to capture architecture design decisions. Other templates have been proposed as well; see e.g. [81]. These templates could be used in the meetings like in the previous example to capture the design rationale. However, these templates are rarely used in practice [82] and the documentation of the architectural decision is seen as a retrospective activity. The problem of such retrospective approach is that the design rationale is not available to the stakeholders [82].

Therefore, the granule-based approach to capture the architecture knowledge would mitigate the problem as the knowledge is captured in small chunks while it is being created. However, as it will be discussed in Chapter 9 more thoroughly, the capturing of architecture knowledge must be as invisible as possible to enable developers to do it. The best case would be that the knowledge is captured as a side effect of other day-to-day activities as design or writing meeting memos. One of the approaches that are proposed in this thesis is to capture architecture knowledge as a side effect of architecture evaluation. With proper tools (see Chapter 8), architecture knowledge can be input to the AKR base without any extra effort.

As mentioned earlier, Nonaka & Takeuchi [20] distinct between tacit and explicit knowledge and this division is also applicable to architecture knowledge. To refine that, Lago and Avgeriou [19] suggest distinction between application-generic and application-specific knowledge. Application-generic knowledge is knowledge that can be applied to multiple systems or applications while application-specific knowledge includes architecture decisions, solutions, etc. that concern a particular system.

The two distinctions introduced above are orthogonal and combination of these results in four main categories of architectural knowledge [70]. Tacit application-generic architecture knowledge includes the experience of the software architect and her tendencies to favor certain kind of solutions, etc. Application-specific tacit knowledge includes business goals, stakeholder concerns, etc. Many times this knowledge category affects the outcome of architecture decisions a lot. The DCAR method tries to facilitate codification of this type of architecture knowledge. In other words, DCAR tries to capture application-specific tacit knowledge and make it more tangible by transforming it to application-specific explicit knowledge. Application-specific explicit knowledge includes architecture views, models architecturally significant requirements (ASR) and codified decisions and their rationales.

Application-generic explicit knowledge is design knowledge that is made explicit in books, standards, etc. It includes reusable solutions such as patterns and architecture styles and tactics [70]. Patterns in Chapter 4 can be seen as an example of application-generic explicit knowledge. The pattern mining method introduced in Section 4.1 uses application-specific explicit (and to some extent tacit) knowledge and transforms it to application-generic explicit knowledge.

If DCAR and pattern mining methods are used together, they can be seen as a way to transform tacit application-specific knowledge to explicit application-specific knowledge and further to explicit application-generic knowledge. This is beneficial as design patterns can be used to boost software architecture related activities such as design and architecture evaluation (see Chapter 5).

According to Babar [83] researchers and practitioners agree that if one does not take care and codify architecture knowledge, it will be lost over time. Even if the architecture knowledge is codified and found in documents, it will get outdated as software architecture is not static, but evolves during the development. More knowledge of the surrounding world is discovered during the development of software system, and architects and other stakeholders gain more experience on technologies. Better design alternatives are discovered and thus the software architecture is dynamic and evolving during the development.

To answer the problem of information getting outdated, architecture knowledge management is needed. Tools, such as TopDocs, are needed to support the AKM process, so that all stakeholders have the newest knowledge in their use and no knowledge is lost. However, there is no single tool that would fit all domains and all needs in the industry. Thus, tools used to support AKM processes must be flexible and extensible. For example, in the development of safety-critical systems, the system needs to be verified that it fulfills the safety requirements set for it. Vepsäläinen et al. [47] demonstrate how TopDocs can be extended with new views and functions to support development of safety-critical systems.

3.5 Software Architecture Decisions

As discussed in earlier sections software architecture decisions are nowadays recognized as first class entities in describing and documenting software architectures ([5], [80] and [24]). Unfortunately, software architecture decisions are not yet widely adopted among practitioners [82].

While Bosch [5] lays out the concept of using decisions to capture software architecture, there are no instructions on how to document the decisions. Tyree and Akerman [80] present a format to capture the decisions. This format is shown in Table 1. Kruchten proposes to extend the format by adding an author, timestamp and revision history of the decision to the documentation template. Kruchten [53] continues by presenting a state model for architectural design decision. The state model presents the following states for the status of the architecture decision: idea, tentative, decided, approved, challenged, rejected, and obsolesced.

For recalling the rationale of a decision, it is naturally beneficial to document all the previously mentioned aspects of an architecture decision. In practice, however, it is a huge task to document all of them and thus it might be too laborious and time consuming to do in practice. Especially as decisions are constantly revisited and new arguments in favor and against the decisions are discovered, requiring that the documentation needs to be revisited and documented again. Architecture knowledge repositories with proper tools could reduce the required work as many things could be automatized.

Kruchten [53] presents also an ontology of the relationships between architectural decisions. A relationship between decisions can indicate that a decision constrains, forbids, enables, subsumes, conflicts with, overrides, comprises, is bound to, is an alternative to or is related to another decision. In addition, Kruchten [53] presents relationships between architectural decisions and external artifacts: traces from, traces to and does not comply with. In later studies, it has been claimed that the relationships between the decisions are as important as the decisions themselves [69].

Table 1 - Architecture decision description template (adapted from [80]).

Item	Description
Issue	Description of the issue that the decision addresses.
Decision	The direction taken. Outcome of the decision.
Status	Decision’s current status. E.g. pending, decided, approved.
Group	Group that the decision belongs to. Helps to organize decisions.
Assumptions	Describes made assumptions, e.g. on costs, schedule, etc.
Constraints	Captures the constraints the chosen alternative might cause.
Positions	List of alternatives to the made decision.
Argument	Rationale why this alternative was selected.
Implications	Consequences of the decision made.
Related decisions	List of related decisions.
Related requirements	Requirements driving the decision.
Related artifacts	Architecture design or documents that are affected by the decision.
Related principles	Principles used in the enterprise that the decision needs to be aligned with.
Notes	Section to record notes made during the decision making process.

When the relationships between decisions have been documented, it is trivial to see which decisions affect other decisions. For example, if changes to an architecture decision emerge, other affected decisions can be seen from the relationships of that decision. van Heesch et al. [81] present a documentation framework for architecture decisions. As a part of that framework they present the relationship view of architecture decisions which visualizes the relationships between the decisions. Using the relationship view, one can trace the affected other decisions when a change to a decision is made. Often it might be a good idea to revisit these decisions, too and see if they need to be changed as the world around them has changed.

Other approaches have been taken, too. Dueñas and Capilla [84] propose to use an additional decision view along with the traditional 4+1 model [51]. In addition, Dueñas and Capilla [84] present an approach to associate architecture decisions to the resulting architecture design and its elements. Also guidance for software architects on how to capture and present the architecture decisions has been presented in [85].

Tang et al. [86] present a study showing that practitioners recognize the importance of documenting design rationale and they frequently use it to reason about their design choices. Almost 90 % of the respondents of that study said that they use design rationale frequently to justify their designs. On the other hand only roughly 35 % of the respondents indicated that they have never or only rarely documented discarded decisions. Tang's study also found out that the main reasons for not documenting design rationales are lack of time, budget, standards, and proper tools.

Agile software development might on its behalf reduce the eagerness to do heavyweight documentation of architecture decisions. Tooling and methods could help to some extent, but the most dominating factor is time and budget. Exhaustive documentation requires a lot of time, even if it would be written once at a late stage in a project. The cost of the documentation of architecture decisions is high, especially when compared to the value the documentation has at the time of writing. Often at the time the document is written, people remember the decisions and their rationale rather well. However, the most value of documenting architecture decisions is gained years later, when making changes to the system. This kind of approach would suit perfectly systems like distributed machine control systems and medical devices which have long life cycles and changes to the system might need to be implemented years after it was first launched.

DCAR uses architectural decisions as a first class entity to carry out software architecture evaluation. In fact, using architectural knowledge for iterative software architecture evaluation has been recommended by Kruchten et al. [71]. DCAR employs an architecture decision relationship view basing on the decisions identified in evaluation session. The decision relationship view can be used after the evaluation to see which other, possibly unevaluated decisions, also need to be revisited after making changes to the evaluated decisions.

3.6 Agile Software Development Methods

Since the publication of agile manifesto [7], the agile software development methods have been widely adopted in the industry. Agile manifesto lists the following values of agile development:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

While there are many agile methods such as Kanban [87], extreme programming (XP) [88] and lean software development [89], Scrum [90] is one of the most popular ones [8]. Furthermore, many of the other agile methods can be combined with Scrum. Principles of lean software development match quite well with the Scrum framework [18]. Kanban does not even describe a specific set of roles or process steps, but states that one should start with the existing process and incremental evolutionary changes should be made to the process. In addition, practices of XP such as pair programming or continuous integration can be used with Scrum. In fact, one might have hard time deciding which method the team is using as the methods are so similar. In the following, the Scrum framework is briefly explained to give a context for the work carried out in this thesis.

Scrum is an iterative and incremental framework for agile software development [91]. As it is a framework it is not ready-made method but should be adapted for the current need. Thus, one can easily adopt practices from XP and lean software development or even use Kanban with Scrum.

Figure 4 shows the workflow in Scrum. In Scrum there are three roles: *Product owner, Scrum Master* and *Development team* [92]. Sometimes these three roles are referred to as the Scrum team. The product owner is responsible for maximizing the value of the product being built. The product owner does this by creating backlog items which describes the features needed in the product. These backlog items are listed in *product backlog*. Product owner orders the backlog so that on the top of the backlog there are the most valuable items for the customer. Product owner needs to take care of the possible dependencies between backlog items when ordering the backlog. When the product owner is deciding which the most valuable items that need to be built next are, the product owner must take the needs of other stakeholders and the needs of development team into consideration. Product owner should also ensure that the development team understands the items in the product backlog and clarify them if needed.

The development team is a cross-functional and co-located team of 6 (+/-3) people [92]. The development team has all necessary skills to build the product and the team does the work to produce potentially shippable product increment in each sprint. Sprint is a short time-boxed period of 2 to 4 weeks which is used to create the product increment. During the sprint, no changes to the goals of the sprint are made.

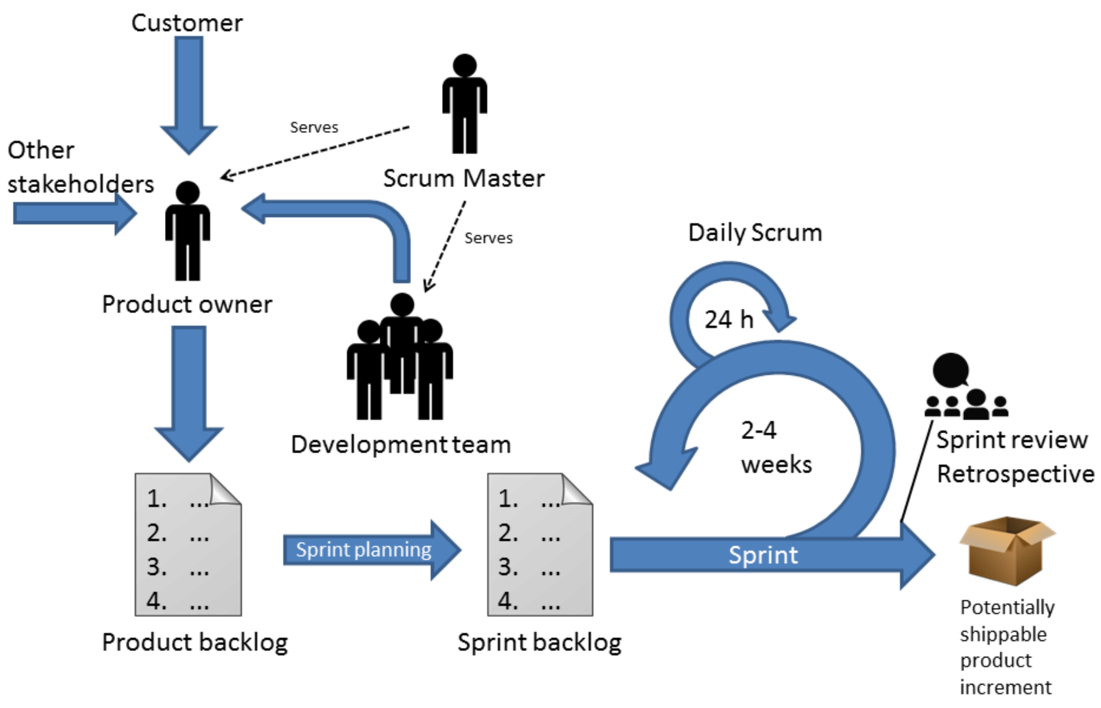


Figure 4 - Workflow in Scrum framework

The development team is self-organized and the members of the team will decide by themselves how to turn the product backlog items into shippable product increments.

The third role in Scrum is Scrum Master who is responsible for ensuring Scrum is understood and that the development team and the product owner are continuously improving their way of working. Scrum Master rather serves the product owner and the development team than gives them tasks. Scrum Master helps people outside Scrum to understand which of their interactions with the development team and the product owner are helpful and which are not. Scrum Master coaches the team to improve their way of working. Additionally, when requested or needed Scrum Master organizes Scrum events.

There are four meetings in Scrum: sprint planning meeting, sprint review, retrospective and daily Scrum (see Figure 4). Sprint planning meeting takes place in the beginning of each sprint. In sprint planning meeting the development team will plan the work for the sprint by selecting items from the top of the product backlog. Product owner is consulted if some items need more clarification. When the goal for the sprint is set and items from the product backlog are selected, the development team decides how the functionality is built. The functionality is split to smaller chunks that can be implemented in a day or two. These tasks are put to sprint backlog which is the list of work to be done within the next sprint.

When the sprint is finished, a sprint review meeting is organized. In this meeting, the development team presents the results of the sprint to product owner who reviews the work. If corrections are needed, they will be made during the following sprints depending on the value of the changes requested. After the sprint, retrospective meeting is also held. In retrospective meeting, the development team evaluates their performance during the last sprint and with the guidance of Scrum Master the team creates a plan for improvements in the way of working [92]. These improvements will be put to the product backlog and carried out in the following sprints.

Daily Scrum is a 15-minute event where development team members synchronize activities done since the last daily Scrum and they plan the activities until the next daily Scrum [92]. Daily Scrum takes place every work day at the same time and only development team members are allowed to speak in the meeting. The purpose of this meeting is to iterate the sprint planning every day and to communicate the latest development.

In Scrum development team also must agree on *Definition of Done* (DoD). It is not always intuitively clear when a task is done. Thus, the development team must have a shared understanding what it means for work to be complete [92]. For example, the implementation of a feature might not be enough for a task to be done. To finish the task, the developer might need to write and run unit tests, document and release the functionality to say that the task is done. DoD needs to be specified depending on the context where the development is carried out. DoD can also be refined by the development team if needed.

While using Scrum also other agile practices can and should be used. For example, as new functionality is developed during sprints, new code should be easy to integrate with the existing code base. Teams often use Continuous Integration (CI) as it aims to improve the software quality [93]. When CI is used, developers are encouraged to integrate their changes regularly and with short intervals in order to avoid too many changes at once. Often automatic testing tools are also integrated in CI, so that when the developer submits a change, the CI system automatically runs tests to the code. In this way, it is rather easy to track if a change causes tests to fail.

In addition, the development team might use code reviews to improve the quality of the code even before it is integrated. Code reviews also support distributing the knowledge concerning the changes made to the code base [94]. Another technique aiming for better code quality is refactoring [95]. Refactoring means improving the structure and design of the current code making it easier to read, understand, and work with. Refactoring only changes the internal structure of the code, but does not change the behaviour. Refactoring might be suggested by another developer as a result of code review. If a feature is implemented in a quick and dirty fashion due to time constraints, refactoring may be required later on to meet the quality standards of the team. In this way, the team pays back technical debt [96] which was taken when the feature was developed. Then at the later time, the code is refactored to meet the quality standards of the team.

Continuous integration and refactoring are eXtreme Programming practices, but many Scrum teams use them. Similarly, pair programming [88] or even mob programming [97] are adopted by many Scrum teams. This is natural as Scrum encourages teams to find new practices that work for them. There is no ready set of practices that would work for all teams. Selection of practices depends heavily on the context where the software is written.

3.7 Agile Software Development and Software Architecture

It has been said that software architecture and agile software development is a clash of two cultures [9]. People using agile methods tend to think that software architecture has a smell of waterfall in it: If you plan and design architecture up-front, you end up with loads of documents and YAGNI (You Ain't Gonna Need It) features [17]. Furthermore, by planning in advance, one creates a separate step before implementation that starts to remind waterfall-like approach to software development. Still, the original description of Scrum [90] has a separate planning and architecture step before actual sprints. In later descriptions, e.g. [92] , it has been removed.

At the same time, the annual state of the agile survey lists year after year that the greatest concern about (further) adopting agile is lack of up-front planning [98] [8]. Similarly, McHugh et al. [99] claim that barriers to adopt agile in the domain of medical devices include lack of up front planning and lack of documentation.

At the heart of this problem is the agile way to embrace change and adapt to the current situation against the traditional software architecture way to anticipate changes and preparing for them. Agile and lean want to postpone the decision making to the last responsible moment [18] whereas software architecture has traditionally been perceived as preparing for the future changes in advance. However, recently one of the originators of Scrum, Jeff Sutherland said that successful usage of Scrum requires some up-front planning and design [15]. Thus, it is clear that a balance between these two worlds needs to be found.

In agile and lean the value delivered to the customer is often emphasized. Even Agile Manifesto's values state that one should value working software over comprehensive documentation. If the first sprint or two are used to create an architecture design, there is no value delivered to the customer. However, one should recognize that the developers of the system are also stakeholders and the architecture has value for them. The architecture might enable the developers to deliver value to the end user with increased velocity. In addition, time spent planning in the beginning might result in higher quality of the product. While architecture is invisible to the end user, it is a way to provide more value to the end user in the future. Similarly, other aspects related to internal quality of the product reflect in the external quality of the product [1].

Cockburn [100] describes a strategy for software architecture, where one builds a walking skeleton first and then evolves it iteratively. In this thesis, a study on different ways of carrying out architecture work revealed that architecture is either planned up-front or done in sprints, just like described by Cockburn.

Whatever the approach is, iterative or (mostly) up-front, it is clear that not all decisions are architecturally significant. There are smaller design decisions that can be made in sprints by the developers. These decisions often concern how things are done on a low level. The most important architecture decisions usually need to be done on day one in the project [17]: Will the system be distributed or centralized? What technology stack will be used? And so on. Therefore, there needs to be at least some architecture work done in the beginning of the project. Unfortunately, in the beginning of the project, when the most important decisions need to be made, there is least information available. Thus, it is good to involve everybody concerned in the decision making to get the widest view possible on the problem. Still, the decisions may need to be revisited later on.

Kent Beck advises that using XP does not take away the importance of software architecture – it is as important as in any software project [88]. The same applies to other agile methods, too. However, one shouldn't get carried away with planning and design, but do just as much as is necessary. Fairbanks [101] proposes a risk-driven approach to decide how much architecture work is required. In general, this is the way most agile teams work. For example, in simple web projects, there is not much architecture involved. Most of the architecture is dictated by the selected technology stack. Similarly, if a developer wants to create an iPhone application, Apple enforces the application to use the Model-View-Controller (MVC) architecture [10]. There is not much architecture left to design especially if the mobile application does not require backend. Still, larger and more complex projects like the control system of a large work machine or medical device software require more architecture work [17].

The amount of architecture work needs to be selected basing on the context of the project, i.e. what is the environment, project type, domain, etc. This is also reflected on the study of software architecture practices: different organizations use different approaches to align agile and software architecture. However, there was no clear evidence that any of the mentioned factors alone would determine which way to work is the most suitable one.

In practice, there is no fundamental clash of cultures between agile methods and software architecture. However, agile often fails to value up-front design [18]. This together with the often misinterpreted instruction to value working software over comprehensive documentation in agile manifesto has given software architecture a bad reputation in agile. One of the key problems is that while there is no contradiction between agile and software architecture, agile does not provide tools to carry out software architecture work. Agile does not say how to identify architecturally significant requirements or carry out iterative design. There is no guidance on software architecture evaluation in agile and Scrum.

There is a plethora of methods and tools for software architecture work, but most of them are attuned for waterfall development. For example, ATAM evaluation [16] takes two to three days excluding preparation and reporting. This makes it costly. Because of the limitations, ATAM is mostly single-shot approach to software architecture evaluation and in agile it is hard to decide when it is the right moment to carry out the evaluation. Therefore, we need guidance on how to align software architecture practices to agile and we also need iterative ways to evaluate architecture. While there has been a lot of studies and methods for identifying promising candidate architectures and evaluating the completed architectures, there has been far less work on incremental or continuous evaluation of architecture [54]. This is the void that DCAR tries to fill, at least to some extent.

Similar problems concern software architecture knowledge management. Jansen & Bosch [24] state that problems of software architectures such as high cost for change, complexity and erosion, are partially due to knowledge vaporization. In agile software development, the software evolves constantly and the high cost for change is not acceptable. On the contrary, the cost of change must be made as cheap as possible. Thus, the importance of good knowledge management is emphasized in agile. Software architecture decisions should be captured and revisited once in a while to see if they are still valid or has the made assumptions behind the decisions changed. To keep the cost of such tracking low, architecture knowledge management must be lightweight and as invisible to the developer as possible.

PART III - DOMAIN KNOWLEDGE AND SOFTWARE ARCHITECTURE EVALUATION

This part will discuss usage of domain knowledge in software architecture work. First, patterns serve as an example of domain knowledge and a process for capturing this knowledge is described. Second, a method to utilize domain knowledge to boost software architecture evaluation is introduced. Finally, a decision-centric architecture review method is presented.

4 PATTERNS FOR DISTRIBUTED MACHINE CONTROL SYSTEMS

4.1 Mining Software Architecture Patterns

This section is a verbatim copy of the following book section:

Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen and Ville Reijonen- 2014. Section 4.3: The story of the Patterns in this Book. In: Designing Distributed Control Systems – A Pattern Language approach. John Wiley & Sons, ISBN: 978-1-118-69415-2, pages 86-90.

Published here with kind permission of John Wiley & Sons.

The patterns you will find in this book have come a long way from encountering to ending up on these pages. Each of the patterns has been encountered at least three times independently in real world systems. All the patterns are deeply rooted in the industrial practice. In other words, patterns are not invented; they emerge in practice and are discovered. In this section we will explain how the patterns were identified and refined. The process is also partially described by [102]

The pattern collection started in 2008 in project called Sulake which was funded by Tekes[1]. During the years 2008 to 2010 the authors carried out approximately twenty software architecture evaluations using a method resembling Architecture Trade-off Analysis Method, ATAM [16]. These evaluations were carried out in companies manufacturing mobile work machines. In these software architecture evaluations, we got an opportunity to examine the architectures of various machine control systems. The companies agreed that we can collect architecture solutions used in the systems during the evaluation sessions and document them as patterns. However, this meant that we had to promise not to give away the solutions as such as they might be trade secrets. Thus, code examples from the actual products are not presented in this book. The examples presented in this book are still similar to real world systems, but they are not excerpts from the code of the actual systems, but written by the authors to demonstrate the design. The two and half year's work resulted in a report [103] containing the first versions of the patterns and the pattern language.

Soon after the releasing the report it was clear that the work is not finished yet. The work continued in another Tekes project called Sulava. One of the authors worked in the domain for a while and saw many solutions that had eluded our first attempt to create the pattern language. As the authors got feedback on the report from the practition-

[1] Tekes is Finnish funding agency for technology and innovation.

ers, colleagues, researchers and pattern enthusiasts, it seemed that radical changes were still needed. Thus, we continued our pattern collection efforts until 2013. In this stage, for example patterns concerning HUMAN-MACHINE INTERFACE were added to the language. Furthermore, in the technical report [103] we used so called Canonical form [104] to document the patterns. We were not very happy with the format as separate sections were disrupting the flow of a story. We decided to move on to an adaptation of Alexandrian form [105] to make the text flow more fluently (see Section *The Pattern Format Used in This Book*).

In the beginning we used *how to* styled problems with the Alexandrian form. However, we soon realized that problem statements in the patterns were like "How to do the solution?" and then the pattern explained the solution. The problem statement wasn't very informative if you didn't know the solution. Thus, as the next step, we removed the *how to* part from the problem statement. The result wasn't very satisfactory either as now the problem statements sounded more like requirements: "Because of X, there should be the solution" (where X was the key driver for the solution). We noticed that it is surprisingly hard to come up with a good problem statement which really tackles with the root cause for applying the pattern. Furthermore, the root cause should also be common to all instances where the pattern has been identified in the industry. Currently, the problem statement tries to describe the root cause for applying the solution. Still, we have sometimes failed to find the root cause. While the problem statements of some of the patterns deal with the root cause, there still are some patterns where it remains undiscovered. Star rating of the patterns reflects this issue.

As already mentioned, the patterns in this book are grounded to the practice and to industry. The initial set of patterns was collected during architecture evaluation sessions. In an architecture review, the most significant architecture decisions are identified from the system and written down. Harrison states in [106] that applying a pattern is essentially an architecture decision. Other way around, it means that each architecture decision identified from the design is potentially a pattern. So, the list of architecture decisions, elicited during the software architecture evaluation, provided us the initial set of pattern candidates. Of course, during the evaluation sessions all kind of solutions were discussed and new solutions alternatives were elicited during the discussions. Sometimes these alternative solutions were also written down as pattern candidates. To record the pattern candidates in software architecture evaluation, we had a person whose only task was to write down the pattern candidates in a wiki page. Once the pattern candidate had been identified three times, it was promoted to a pattern idea. At this stage an initial patlet was written and a dedicated wiki page for the pattern was created. All notes relating to the pattern idea was moved to this wiki page.

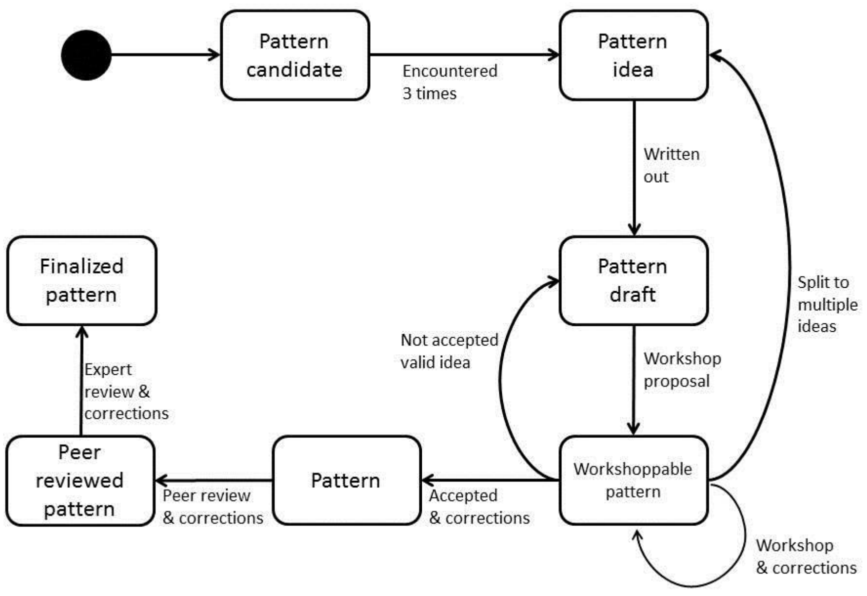


Figure 5 - Pattern writing process.

Figure 5 illustrates the life cycle of a pattern during the pattern mining process. The pattern writing process has evolved during the years and in the beginning it was different from the one presented in Figure 5. The figure shows the process for writing the final versions of the patterns that are presented in this book. First the pattern candidate is identified. The pattern candidate can be identified in three ways: in a software architecture evaluation, from literature or by discussing with an expert in the domain. Once the candidate has been identified from three independent sources, it is promoted to a pattern idea and the first version of the patlet - problem solution pair - is written. Then the first draft of the pattern is written. Each of the pattern drafts has been workshopped at least once in a writer's workshop either in one of the PLoP conferences, see [107] or in an internal workshop. See [108] for more information on writer’s workshops.

After the first workshop four things can happen. If the quality of the pattern is high enough, it might be accepted for peer review after the first workshop. Even in this case, the author of the pattern has to do some minor corrections to the pattern basing on the feedback from the workshop. Pattern may also be rejected by the workshop if the idea was not elaborated correctly in the pattern draft. In this case, the author makes a new draft for the subsequent workshop - probably from the scratch. In the third case, the idea is captured correctly in the pattern, but the quality of the first draft is so low, that the workshop cannot accept the pattern yet. In this case, the author makes corrections to the pattern and the subsequent workshops are arranged. Fourth option is that the workshop discovers that the pattern idea was not focused enough and there were multiple patterns written in one pattern. In this case the pattern is split to multiple ideas and these are rewritten as separate pattern drafts. For example, ONE TO MANY and HIGH LEVEL PROTOCOL patterns were originally written as a single pattern called MESSAGE BUS [109]. However, the pattern was very long and confusing for the reader. Thus, after a workshop at VikingPLoP 2013, it was split into two patterns.

After the pattern has been accepted by the workshop and corrected, it was peer-reviewed by another author. In this stage, another author still suggested improvements to the pattern. As the authors work in academic world, they wanted to make sure that the pattern really has industrial relevance. Thus, after peer-review the pattern was sent to software architect working in the industry for review. He made sure that the pattern was captured correctly. Sometimes description of the implementation techniques were also refined in the pattern basing on the architect's comments. During this process the patterns were written so that implementation details could not be traced back to any specific company and solution. For example, known usages come from the real systems, but may not be from the system mentioned in the example. Overall, the process the patterns were collected and refined resembles the grounded theory based process described in [39].

The process of collecting and writing this pattern language took almost six years. During this period intermediate versions of some of the patterns presented in this book have been published in the proceedings of various PLoP conferences. The progress we have made is visible in those conference proceedings. Simultaneously with the pattern writing process we have been collecting more knowledge on the domain and on patterns. We have visited dozens of companies in Finland and in Europe and interviewed experts in this domain. Software architects, developers, managers, etc. gave their invaluable feedback to the patterns. These interviews have provided us insights on many of the solutions on presented in the patterns and sometimes resulted in new patterns as more and more pattern candidates were seen in practice. Basing on the interviews, also the names of some of the patterns were changed and aliases were identified.

The approach that the pattern needs to be identified three independent times in practice, of course, means that all the newest solutions are ruled out as they have not yet widely spread. Even though, this might sound alarming, but it also has at least one benefit. The newest solutions used on the domain may not have yet matured enough to be widely used, so the patterns are really solutions that are approved in practice. One might also see that there are patterns presented in this book that have been documented already. For example, HEARTBEAT pattern has been documented multiple times by several authors (e.g. [110]). However, we wanted to document these patterns from the perspective of control system domain as they are central solutions in this domain. We wanted also to ensure that the pattern language in this book contains most of the essential solutions for the machine control systems.

To summarize, the patterns have come a long way to this book and to you. We hope that you enjoy reading and using these patterns as much as we enjoyed writing them. However, a pattern language is never complete. There are still many open issues and undocumented patterns as new innovations emerge. We expect this work to continue and the pattern language to start living the life of its own. We hope that you will take these patterns in daily use and will modify and update them. If you encounter new patterns, hopefully you will document them and share with the community, e.g. by submitting them to one of the PLoP conferences. In this way, the pattern language could gradually grow as new patterns will complete this pattern language. Furthermore, we hope that people will share their experiences with the patterns and their completions to the language in one way or another.

4.2 Example Set of Patterns for Distributed Machine Control Systems

This section is a verbatim copy of the following publication:
Veli-Pekka Eloranta and Johannes Koskinen. 2011. Messaging patterns for distributed machine control systems. In: *Proceedings of the 16th European Conference on Pattern Languages of Programs* (EuroPLoP '11). ACM, New York, NY, USA, Article 12, 18 pages.

Published here with kind permission of Association for Computing Machinery.

4.2.1 Introduction

A distributed machine control system consists of multiple embedded controllers which communicate with each other over bus. These embedded controllers are devised to control, monitor or assist the operation of equipment, machinery or plant [111]. Typically communication over bus introduces certain kind of challenges that architects need to deal with. Communication bus throughput may be a problem or sending interval of messages. In addition, sometimes there is a need to limit amount of sent messages to avoid situation where one node is flooding the bus. In this paper, we will present six design patterns which solutions are related to messaging in work machine control system architectures. Patterns were mainly collected during the years 2008 and 2009 while carrying out architectural evaluations in Finnish machine industry. However, refinements to the patterns have been made according to observations and comments from industry in 2010. Patterns originate from companies which are global manufacturers of large work machines (e.g. mining drills) or process automation systems. The collection process of pattern language is documented in more detail in [102]. Currently our pattern language consists of 51 patterns in total.

4.2.2 Setting the Landscape

Typically mobile working machines consists of machine body, engine, wheels or crawlers, cabin, a boom and a boom head, which does the actual work. The engine is used to generate hydraulic pressure that is used to operate the boom and other components of the system. Basic principle in design of these systems is that control of functionality is located near the device itself, e.g. boom head has its own controller located in boom head. This design decision is typically made in order to reduce wiring costs and to increase fault tolerance. From the software architecture point of view, this means distribution. Typically, these systems have a bus such as CAN (ControllerArea Network) [112] or Ethernet that connects different controllers.

Bus technology offers increased fault tolerance and cost effectiveness over separate wiring. However, it introduces new kind of complexity that needs to be dealt with. In addition, bus capacity may become a problem if there are a lot of nodes attached to the bus. Sometimes bus length may become too long. Then the bus can be segmented and possibly different kind of buses used for different bus segments. Furthermore, in some cases it might be necessary to use two buses in parallel. For example, CAN bus for control messages and Ethernet bus for large data transfer such as videos or updates.

Mobile working machines usually have one central node (usually a PC) that is used to run high end services such as graphical user interface, remote diagnostics and other performance greedy applications. In addition, mobile work machine has sensors that can be attached to the controllers directly or they can be so called intelligent sensors that are connected directly to the bus.

4.2.3 Patterns

In this section, we present some messaging patterns for distributed machine control system. The presented patterns are grayed in Figure 6. The relationship used between patterns in Figure 6 is refines meaning that the pattern following some other pattern refines the solution of previous pattern. Of course the applicability order in Figure 6 is just a suggestion.

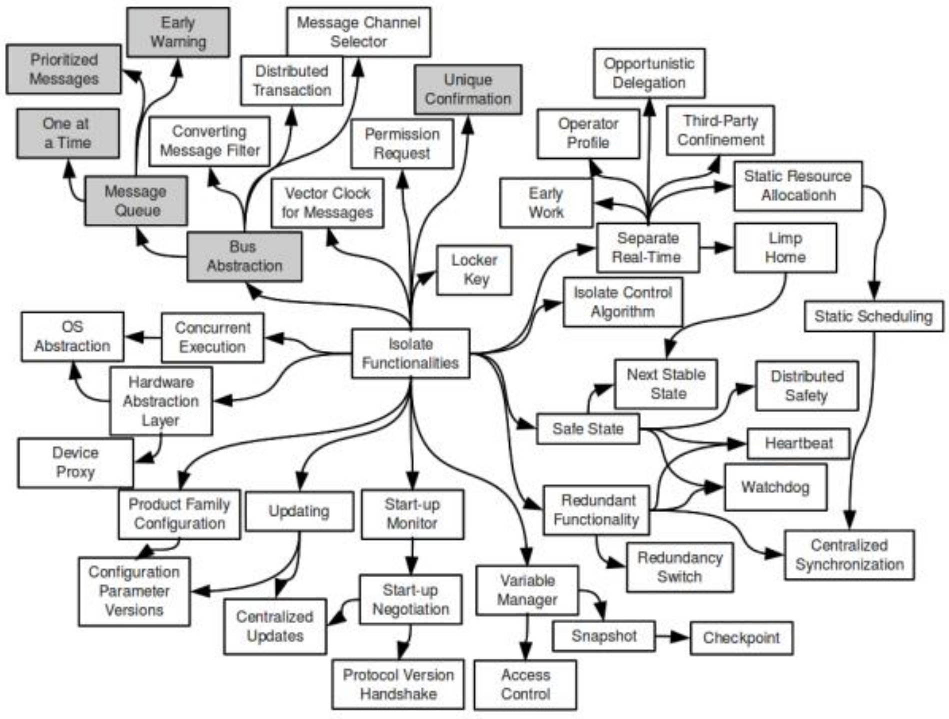


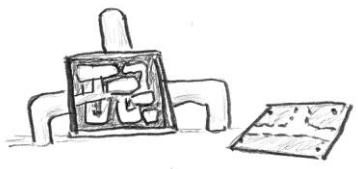
Figure 6 - Pattern language for distributed machine control systems. Grayed patterns are presented in this paper.

We use the following template to describe our patterns. First, we present *Context* where the pattern can be applied. The next section is the description of *Problem* that the pattern will solve in the given context. In *Forces* section we describe the motivation or goals that one may want to solve by applying this pattern. Furthermore, we give *Solution*, argument it and present positive (+) and negative (-) implications in *Consequences* section and give *Resulting Context* that will be the new context after applying the pattern. Finally, *Related Patterns* are presented and in *Known Usage* one known case of usage is given. In addition pattern names are written in SMALL CAPS. Problem statement and the core of the solution are typed in bold face font to make it possible to browse patterns quickly to find the right pattern.

4.2.3.1 Bus Abstraction

Context

A control system has been distributed using ISOLATE FUNCTIONALITIES pattern. The system consists of multiple nodes. Nodes are controllers or intelligent sensors. There is a bus, e.g. CAN, connecting the nodes and nodes communicate with each other using messages. Typically lifecycle of work machine is rather long, i.e. it might be over 20 years. During the life cycle of the control system software, it might be possible that chosen bus technology or protocol, e.g. CANopen, needs to be changed. This might happen as the technology is not available on market anymore or availability of spare parts weakens. Changing the used bus technology would mean that changes have to be made to multiple nodes and multiple places in code. Basically, as the bus specific calls are used developers are locked to the selected bus. Additionally, bus standard complexities have to be addressed in multiple nodes. In all, the situation will create unnecessary redundant code.



Problem

How to change the selected bus technology or protocol so that the application code does not need to be changed?

Forces

- *Scalability*: The system may incorporate many nodes and a lot of communication may emerge between them.
- *Modifiability*: Used protocol should be hidden from the developers.
- *Flexibility*: Developers should not need to know where certain services physically reside.

- *Scalability*: The system may incorporate multiple buses which use different technologies and protocols. Main bus might be used for controlling the machine whereas the second bus is used to transfer large files, e.g. videos.
- *Reusability*: It should be possible to use the same code in all nodes, thus reducing the amount of code that needs to be written.
- *Portability*: During the long life cycle of the product, the physical bus may change, e.g. because the technology becomes obsolete.

Solution

In order to make the system bus technology (and protocol) independent, a common application programming interface (API) is constructed to provide uniform messaging functionality. The main task for API is to hide, how the bus actually functions from the applications and application developers. Therefore, as API hides the actual bus implementation, the bus technology or protocol used can be changed while API will stay the same. A change may require modifications to the bus abstraction, but not to the application itself. When the bus standard and used protocol is abstracted, the addresses and locations of different services cannot be known. Therefore, it is necessary name the services so they could be used. Developers will use these names to point to required services. This implies that there has to be a mechanism to configure the system in generic way. Solution in general is illustrated in Figure 7.

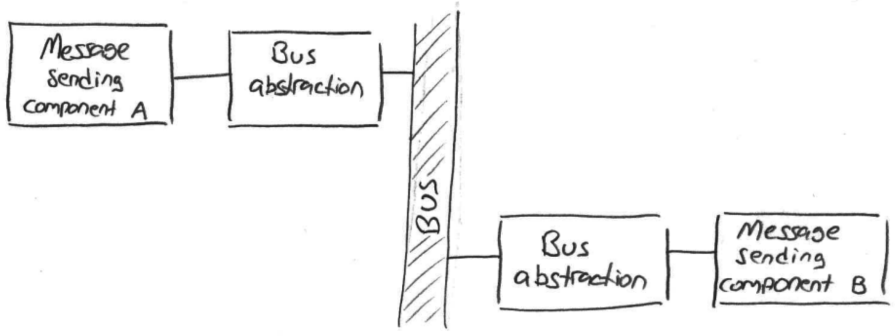


Figure 7 - Structure of a typical bus abstraction.

API should contain basic methods that allow the developer to use the bus. These methods typically are: sending a message, receiving a message, periodic sending of message, periodic receiving and monitoring other nodes sending periodic message to the node. Typically, implementation of receive message method, needs the application to provide a hook function for bus abstraction. Abstraction will use the given function to give received message to the application. Another option is that the application polls if there are messages received ready for processing. In this case, it is sometimes necessary to implement the polling as a separate task. Monitoring other node also requires a function to call when the monitoring fails, i.e. when the other node is failing.

Listing 1 - Pseudocode listing of typical bus abstraction functions.

```
bool i n i t i a l i z e ( function_to_receive_messages ) ;
bool send_message ( message , receiver , type ) ;
Message* read_message ( ) ;
bool send_message_periodic ( message, receiver, type,
error_handling_function ) ;
void read_message_periodic ( method_to_call ) ;
void monitor_node ( node_name , interval, function_to_call_ice);
```

As names are used instead of addresses, it may be difficult to know for the developer if service is local and therefore fast in responding or remote and therefore slower with responses. Remote node may also fail to respond due to malfunction or some other error. In this location question hides design issue with responsibility division: is it API's task to retry and announce failure or applications? In this case what should be done with new messages? API developer should remember that there is no reply or receipt for a sent message on all bus types or for all message types.

The names may be fixed inside the API if the system nodes consist from a standard set. Alternatively API may use a naming service to resolve corresponding node and its address. Additional bus traffic should be avoided, so the API implementation should not query names over the bus too frequently. The reason for avoiding bus traffic is twofold. First, sending a query, waiting and finally receiving an answer will delay sending the message to the recipient. For second, extra bus traffic may add congestion to the bus and delay other messages. Therefore, available names could be requested during start up to minimize bus load during normal activity or the query results should be cached.

Consequences

+ Nodes do not depend statically on each other, but only on the names, thus the system is easy to expand and modify, even at run-time.

+ The bus presents an abstraction of the physical world, understandable to software developers.

+ The same bus abstraction can be used in all nodes. Furthermore, the same bus abstraction can be used in multiple products.

+ When the bus technology is changed, only minimal changes to application code are required.

– Abstraction adds latency to the messaging.

– Might require separate task for polling messages.

– It may be difficult to know if requested service is local or remote.

– If bus technology is changed to another one with different functionality, the implementation of bus abstraction needs to be changed.

Resulting Context

The result is a distributed control system where the bus standard or topology can be changed and the change will not affect the main application code. Application communicates with other nodes through the bus abstraction, which hides the physical location and addresses of other nodes. Bus abstraction adds latency to messaging.

Related Patterns

The pattern can be viewed as a MESSAGE DISPATCHER [113], applied to an embedded machine control system with a physical bus or connectivity. The abstraction can be enhanced with MESSAGE QUEUE which will address the problem of multiple simultaneous messages. MESSAGE CHANNEL SELECTOR discusses the case where multiple routes are available for message delivery. CONVERTING MESSAGE FILTER suggests a method to filter unnecessary traffic.

Known Usage

A forest harvester uses CAN bus with J1939 for communication. The bus might be changed to use some other bus standard in the future and therefore the bus is abstracted behind an interface. All possible device names are listed with destination addresses in the interface code, during start-up the current list of devices is received and is used to mark which destinations are available on the system. When an application needs to send message to some other service, it is done through the messaging interface. To send a message recipient name and message is required. The interface will deliver the message asynchronously. When message is received in the other end, the hookup function is called giving the sender and the message as parameters.

4.2.3.2 Message Queue

Context

There is a distributed embedded control system using BUS ABSTRACTION to communicate between the nodes. The nodes should be able to send and receive messages regardless of the load on other nodes, i.e. nodes should be able to process messages when their load allows it and sender should not need to wait for sending the message. In other words, it should be possible to use asynchronous messaging scheme.

Problem

How to enable nodes to communicate asynchronously with each other?

Forces

- *Consistency*: All messages should be processed, because data should not be lost.
- *Efficiency*: System should be efficient and nodes should not need to wait for each other in communication.
- *Predictability*: The amount of messages sent or received is not known beforehand. At times, the number of sent or received messages can be high.
- *Correctness*: Usually the processing order of messages is important. The correct chronological receiving order should be preserved.
- *Resource utilization*: All messages should be processed as soon as possible to avoid congestion.

Solution

Add a queue for receiving and sending messages to each node. Implement a mechanism to put messages in the queue and that will send messages from the queue. The same mechanism can read messages from the bus and add them to the received messages queue (see Figure 8). The queues are containers which hold the messages in the order they are to be received or sent. The actual implementation may vary, but arrays or linked lists are the most common solutions.

A mechanism to process the buffer with the messages must be implemented in addition to the buffers. Its duty is to handle the queuing of the messages. The component must have some kind of event or interruption mechanism to notice that there are messages ready to be served. The message queue works in "first in - first out" fashion (FIFO), so that the oldest message in the queue gets processed first. In this way, the order of received messages is not changed. If stack would be used, the order would be different.

In the sending end the processing component fills up the buffer with the messages. The message queue processor takes the oldest message in the queue and outputs it to the bus whenever possible, e.g. node has its time slot. The receiving component reads the messages from the bus in the order they were received and puts them to the buffer as soon as the message is received. The application on the node reads received messages from the buffer as soon as it has time for it. When the message is read from the queue it is also removed.

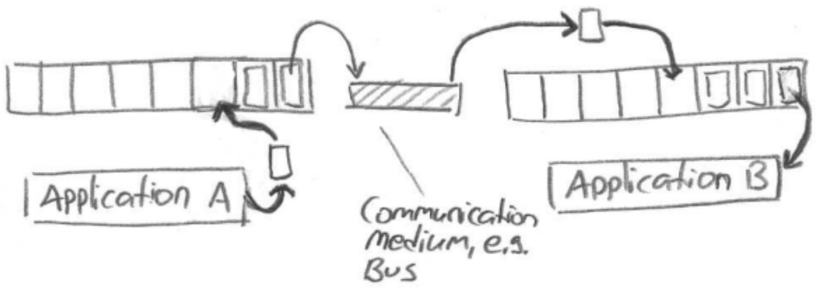


Figure 8 - Message queues in the sending and receiving end of the communication channel.

One might consider encapsulating this kind of message queue in its own component and designing an interface for it. Basically, the operations required from the interface consist of the following methods: read message, send message, queue size. In more complex cases one might consider more advanced methods such as remove certain message, clear queue, etc. It may be helpful to add an override system to the queue, if an emergency situation requires immediate response. The emergency messages can be processed as soon as they are received as the contents of the message queue usually have no use in the emergency situation. However, this causes problems if the system should return to normal operation as soon as the emergency situation is over. This could be remedied using PRIORITIZED MESSAGES pattern.

Consequences

+ The pattern gives the messaging components time to process the messages independently from the physical bus rate.

+ The resource utilization of the bus is improved as congestion peaks can be leveled during the quieter times.

+ The message order is not changed as the queue holds the messages in the order they were put in it.

– There is no easy way to prioritize the messages in FIFO.

– Message queue adds latency if the messaging rate is slower than the processing rate.

– As messages are queued, they may become obsolete while waiting for processing.

– Makes the message sending and receiving more complex.

– It may be difficult to calculate the correct message buffer size, resulting in either resource waste or buffer overflows.

Resulting Context

The pattern results in a system that has ability to communicate asynchronously and the sending not constrained by the message bus rate as the data is queued.

Related Patterns

EARLY WARNING can be used to decrease the chance of possible buffer overflows or to determine suitable buffer size. The MESSAGE QUEUE pattern also resembles closely PRODUCER/CONSUMER pattern. If the order of received messages is not important, one might consider using FRESH WORK BEFORE STALE [110] pattern to determine the processing order of messages.

Known Usage

MESSAGE QUEUE is used in a forest harvester, where nodes communicate via CAN message bus. The CAN hardware holds a buffer that queues the messages that are to be sent or received from the bus. When the application code wants to send a message, it gives it to the driver component which places the message into the queue. The application code does not have to care about the details of the sending. In the receiving end, the driver notifies the application with an interrupt that there are messages in the queue and some action should be taken to process the messages before the queue overruns.

4.2.3.3 Early Warning

Context

There is a distributed embedded CONTROL SYSTEM that uses NOTIFICATIONS or different NOTIFICATION LEVELS to inform the machine operator about different kinds of situations in the system. There has to be a certain amount of resources, e.g. memory, CPU time or slots in the MESSAGE QUEUE, available to safely run critical services, e.g. steering of the machine. If the resource needed by critical service runs out, it might cause a dangerous situation, where the safe usage of the machine cannot be guaranteed. If such situation is about to occur, it should be detected, so remedying action can be started. Furthermore, in testing phase, there should be means to find out what is the worst case utilization rate of a resource, so that in production use, there is just slightly more resources available than are needed in the worst-case situation

Problem

How to detect a situation that the resource availability (e.g. CPU load, memory, oil, etc.) is soon to reach the critical level?

Forces

- *Fault tolerance*: There should be a way to detect if resources are running out. Otherwise, the insufficient resources may crash the whole system. E.g. message buffer overflow could crash the system.
- *Resource utilization*: Resource usage can not be known beforehand. It is discovered during the runtime. There should be a way to test that the runtime resources are sufficient for the system.
- *Testability*: Running out of a resource during the system testing on hardware is not acceptable as it may cause harm to the environment, to the machine tester or to machine itself. There should be a way to detect emerging out run of resources.

- *Cost effectiveness*: Available resources should be sufficient but not excessive. Finding out the right amount of resources can be challenging without testing the runtime behavior of the system.
- *Correctness*: No dynamic resource allocations can be used since it is too easy to produce programming errors using them.
- *Performance*: No dynamic resource allocations can be used as they decrease performance.

Solution

The sufficient amount of resources cannot be known before the system is run. However, when running the software with real hardware, there should not occur any resource outruns, e.g. buffer overflows, as they may compromise safety. **Therefore, add a mechanism that monitors the availability level of a resource. Before the resource is exhausted, the mechanism warns the user or a tester.** Basically, this means that a trigger is added for certain consumption rate of a resource. Once this consumption level is met, an action is triggered. Typically this action is to send a notification (see NOTIFICATIONS). If NOTIFICATION LEVELS pattern is used, different kind of notification can be sent depending on the criticality of the resource.

Let's consider a message buffer or MESSAGE QUEUE for example. During development, we cannot know the amount of messages that are sent and received during runtime. However, we must reserve large enough buffer, to make sure that the node receiving the messages has enough time to process them. Even if the messages pile up, the buffer should never overflow. Overflow on real hardware, however, could cause system to crash and therefore is not acceptable. Therefore, early warning mechanism should be added to the message buffer. If the message buffer size is 100 messages, the early warning trigger is set to 70 % usage. So, when there are 70 messages in the buffer, early warning is triggered. In this way, the buffer still has 30 free slots before the overflow would occur. This gives enough time for the system to react to the situation, for example, by flushing the buffer or shutting the machine down. After the early warning is given, messages can still pile up in the buffer. Figure 9 illustrates the early warning mechanism while using message queue.

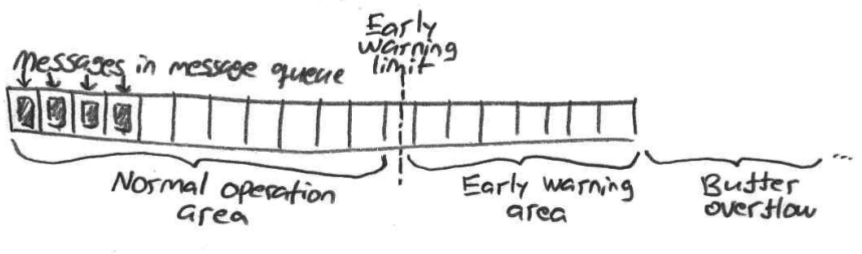


Figure 9 - Early warning system used with Message Queues.

Once the early warning system is in place, the system can be safely tested. If it is found out during testing, that early warning is constantly triggered by resource consumption level, the amount of resources should be increased. For instance, if message buffer is giving constant early warning, the overall size of the buffer should be increased.

The correct triggering level has to be determined case by case. It might work for message buffer to have triggering level at two thirds usage rate. However, for CPU load or memory consumption, the same triggering level may not be optimal. One determining factor is how rapidly the remaining resources are taken in use and how much time the system needs to react to the situation.

Once the early warning is given, the remedying actions depend on the situation. Sometimes it might be good option to shut down the system. On the other hand, in some cases it might be enough to free some unused resources. During the testing phase, it might be just enough to inform the tester that resource availability is at risk. Then tester can decide what should be the remedying action and make adjustments to the system if necessary.

One should also consider using NOTIFICATION LOGGING pattern to record the notifications. In this way, once the early warning limit has been reached, the developer can access the logs and see what caused the situation. Furthermore, from the logs, it can be deducted what the utilization level of the resource that caused the early was warning.

Consequences

+ The pattern helps to verify that there are sufficient amount of resources available. This information can be utilized during the development time to build the system more stable and fault tolerant.

+ Early warning mechanism can be used to detect the situation where resources are running out. Once the situation is detected, some remedying action can be taken.

+ Early warning makes it possible to test the system on actual hardware since possible buffer overflow situations (or other resource outruns) can be recognized beforehand. In this way, dangerous situations which could be caused by unavailability of resources, are prevented even when testing with actual hardware.

– The warning mechanisms may decrease system performance as the limiting value has to be checked. E.g. if early warning system is applied to message buffer, every time a new message is received, it has to be checked if the early warning limit is reached.

– The solution causes that there are more resources in the system than actually is used in the worst-case scenario. For example, message queue will be one third larger than is actually needed.

Resulting Context

The result is a system that has buffers that should be large enough even in the expected worst case situation. Even during the worst case situation the system should have empty slots in the buffer, so it will not crash if the expected worst case situation is exceeded. In the production use the extra space in buffer is used if the expected worst case situation is exceeded and when the early warning limit is met, the system is shut down gracefully. The resulting system can also notify the machine operator or a tester that message queue overflow is about to happen and can shut down the system gracefully.

Related Patterns

Pattern can also be applied when PRIORITIZED MESSAGES has been applied. In this case, each message queue will have its own early warning limit. The ONE MESSAGE AT A TIME pattern can also be used to limit the number of simultaneous messages on bus. NOTIFICATION LOGGING can be used together with this pattern to see what caused the situation that triggered early warning.

Known Usage

EARLY WARNING can be used for example in buffers which store events that are received from the CAN bus. In the development phase, the amount of periodic messages is known, but the amount of sporadic messages cannot be defined exactly. The architect assesses that system has 30 periodic messages and in the worst case there could be 15 sporadic messages simultaneously. The early warning level of CAN bus message queue is set 50 and the buffer size is set to 75. During the testing phase, stress tests results in the situation where the early warning level is met. It is found out from the system logs that the amount of the messages was actually 55 at maximum. The early warning level is set to 60 and the buffer size is increased to 90. Now the tests pass without any warnings from the early warning mechanism.

EARLY WARNING is applied so that in the production use the system will inform the machine operator when the early warning level of message queue used to stored received messages from the bus, is met and the system will automatically shut down the system gracefully. Now, for example if the boom controller node crashes and starts to babble to the CAN bus with sporadic messages, the system will not crash. The extra space in the MESSAGE QUEUE is used store the messages that the node sends and when the early warning level is met the system starts to shut itself down automatically. In this way, the machine malfunction will not cause danger to the operator or environment.

4.2.3.4 One Message at a Time

Context

There is a distributed system where BUS ABSTRACTION has been applied. Nodes communicate over bus using some mechanism, for example provided by MESSAGE QUEUE. Nodes may malfunction and start to send a large amount of (unnecessary) messages to bus, i.e. to babble. This kind of malfunction can be hard to detect with e.g. HEARTBEAT. The unnecessary messages cause extra bus load and may cause delays in delivery of important messages. The babbled messages can also contain erroneous data that may propagate unnecessary actions in other nodes, e.g. transition between two states.

Problem

How to ensure that malfunctioning device does not fill up the bus by repeating same messages? The problem is also known as babbling idiot's problem [114]**.**

Forces

- *Throughput*: Bus load should not have high peaks as they may compromise priority message throughput. In other words, if there is a peak in bus load, important messages may not get through.
- *Fault tolerance*: Malfunction of one node should not disable the whole bus.
- *Safety*: Critical messages should get through bus in all situations.
- *Fault tolerance*: If a node is babbling to the bus, the situation should be contained.

Solution

Add a bookkeeping mechanism to all components that communicate over bus. This mechanism keeps track of all messages that are sent to bus. It makes sure that only a certain number of messages of a certain type can be sent to the bus at a time. Number of messages that can be sent to bus can be configurable, so that larger data sets spreading over multiple messages can be sent to the bus. Nevertheless, the number of messages that can be sent should small enough so that the bus will not choke on the amount of messages.

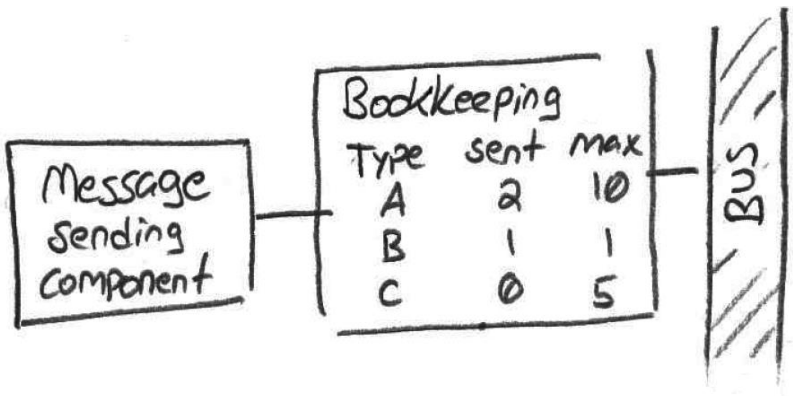


Figure 10 - Illustration of bookkeeping mechanism.

Figure 10 illustrates the solution. There is a bookkeeping component between the message sending node and the bus. Each message is passed through this component and there is bookkeeping of how many messages of each type have been passed to the bus. There is also maximum number of messages of each type that can be sent to the bus. This limit cannot be exceeded and it can vary from node to node. If there is new message of type that has exceeded the limit, the message has to wait.

The bookkeeping mechanism must keep track of Time To Live (TTL) of each message and make sure that no more than certain number of messages are sent to the bus. In addition, nodes can give acknowledgement to sender node, so that the bookkeeping mechanism can allow sending of next message of the same type. Sending acknowledgement to broadcast messages can results in problems in form of acknowledge message flood, so it should be avoided. Furthermore, state of the bus should be monitored in some way, for example using HEARTBEAT, so that the bookkeeping can be sure that no acknowledgement messages are lost. If a node is connected to multiple buses, there should be separate bookkeeping for each bus, i.e. the message sent to one bus does not affect the number of messages that can be sent to another bus.

Consequences

+ Because only limited amount of messages per type at any given time can exists on bus, the bus cannot be flooded by babbling nonsense to it.

+ Malfunctioning unit cannot send conflicting information continuously, amount of erroneous state changes in other nodes are reduced. Nevertheless, the pattern does not solve this problem completely.

+ Critical messages get through even if there is a node malfunctioning and trying to flood the bus.

– The solution reduces bus throughput, as messages are not sent even if there are unused bus capacity, if the maximum number of simultaneous messages are met. In addition, replying to a message is needed or messages must have TTL which have to be waited before sending next message.

– The bookkeeping may add delay to throughput if multiple messages of certain type are sent in a row. As the limiting number of messages may be reached before all the messages of certain type has been sent.

– The solution adds complexity as bookkeeping mechanism is required.

– The bookkeeping forms a single point of failure. If the bookkeeping mechanism does not decrease the message count for some reason, it may prevent the node from sending any messages.

Resulting Context

The system results in a distributed embedded control system where the possibility of babbling node to prevent normal operation is reduced. The bookkeeping mechanism itself cannot babble as it does not send any messages.

Related Patterns

BUS-GUARDIAN [115] solves babbling idiot's problem by introducing time slots for each node where they can send their messages. If some node sends message out of this time frame other nodes notices that the babbling node is malfunctioning.

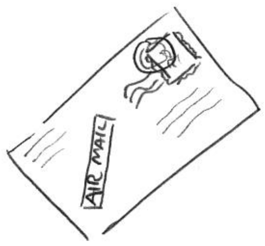
Known Usage

A mining drill consists of multiple nodes that use a CAN bus with CANopen protocol to communicate with other nodes. The controllers in the drilling unit are more likely to malfunction as rocks flying from the drilling process may physically damage controllers. Sometimes damaged nodes may start to send emergency-specific EMCY messages to the CAN bus repeatedly. The pace of sending the messages can be so high that if the number of messages sent is not limited with ONE MESSAGE AT A TIME, the receiving end will choke on the messages. This may result in situation where the receiving node does not get its own messages sent to the bus as it has to handle all messages received. This may even result in situation which makes the machine to stop operating. When this pattern is applied the problem is solved.

4.2.3.5 Prioritized Messages

Context

A distributed embedded control system uses BUS ABSTRACTION to communicate between the nodes. The MESSAGE QUEUE pattern has been applied to enable asynchronous communication. There are some messages that are more important than others.



Problem

How to ensure that important messages get handled before other (less important) messages?

Forces

- *Data integrity*: Regardless of the importance, all messages should be processed because no data should be lost. Thus a message that needs immediate action may not discard other messages.
- *Response time*: All messages should be processed as soon as possible, but taking into account their relative importance, e.g. emergency messages, should be processed immediately.
- *Safety*: Emergency situations should be taken care of as fast as possible, but the system should also be able to return to normal operation as soon as the exceptional situation is over.

Solution

The nodes communicate with each other using a message queuing service. This consists of message queues and some kind of message dispatcher that serves the messages in the queue. **The message types are prioritized according to their importance and separate MESSAGE QUEUEs are implemented for each priority.** The importance is a relative concept so the designer must decide which message types are considered more important than the others. Usually error messages and other exceptional communication are given a higher priority. In most simple case, there are only two levels of priority: messages of high importance and normal messages (see Figure 11). The message dispatcher places messages in different work queues based on their priority level.

When the message processor begins to serve the queue, it begins with the highest priority queue and after it is empty, it starts to serve the lower priority level. After every processed message it must check whether a higher priority queue has new messages to process. The designer must be careful that no starvation on any message priority level shall occur. Starvation means that high priority messaging is so frequent that the lower priority messages cannot be served at all. Basically this means that the designer should not give higher priority to messages which really don’t need it. This might be hard task to define. Another approach to tackle with this problem is to give each priority certain amount of messages that are processed until moving to the lower priority. For example, first process 5 high priority messages, then 3 medium priority messages and finally 1 low priority message. If there is only one high priority message, then after processing that, the medium priority messages should processed and so on.

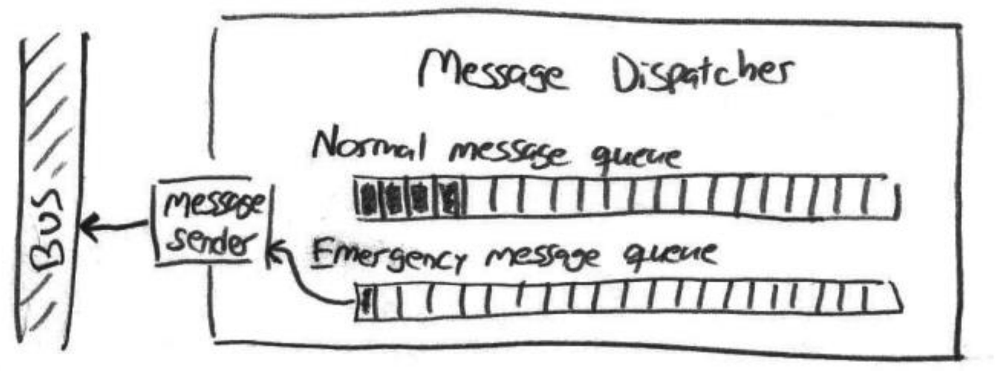


Figure 11 - Prioritized messages with two priorities and two Message Queues for these priorities: normal and emergency.

The message priority should be clearly stated in the message. It may be explicit from the message type, for example all emergency messages have high priority, or the message itself may carry additional meta-information about its priority. In the latter case, the sender must set the priority information correctly to the message and make sure that high priority messages are used only when needed. Otherwise, it may congest the message channel. In some cases, the priority may be deduced by the sender or from the context. For example, some node in the network sends always important messages.

Consequences

+ Most important messages can be processed regardless of the amount of the normal messaging.

+ The order of same priority messages is not changed as the queue holds the messages in the order they were inserted.

– The correct chronological order of messages is lost, because high priority messages are served first.

– Due to chronological discrepancies, priority inversion may occur. This means that a high priority message starts some action, which needs information from a lower priority message that is not served yet.

– Congestion of high priority messages can cause starvation of the normal messages. However, if this is the case, the system is probably overloaded.

Resulting Context

The pattern results in a system where messages are handled in the order of importance.

Related Patterns

ONE MESSAGE AT A TIME solves message prioritization problems using another approach. STL design patterns II [116] describes a similar messaging concept. EARLY WARNING helps in designing proper message queue lengths. VECTOR CLOCK FOR MESSAGES can be used to keep track of the chronological order of the messages.

Known Usage

A drilling machine has controllers that communicate via CAN bus. Basic situations are handled with normal messages, but if an emergency occurs, for example when a controller crashes, a special kind of message is used. These emergency messages have higher priority as the normal messages, so other controllers can react to the situation quickly. The message queues are presented in Fig. 6. The message dispatcher serves the high priority EMCY messages first.

4.2.3.6 Unique Confirmation

Context

There is a distributed embedded control system where ISOLATE FUNCTIONALITIES pattern is applied. The system has some actions that need confirmations from other entities. There might be multiple confirmations waiting for processing. The confirmations must not be confused with each other in order to avoid false positives.

Problem

How to handle situation where similar messages need confirmation responses and the responses may be received in wrong order?

Forces

- *Correctness*: There should be no false confirmations as they may have severe consequences.
- *Resource utilization*: Complicated confirmation protocols may be cumbersome.
- *Performance*: Communication faults or latency may cause confirmations to be sent several times or in incorrect order. This may lead to confusion.

Solution

A situation may rise, where there are multiple pending actions waiting for a confirmation. A simple "OK" message may be interpreted by the acting unit as confirmation for any of the pending actions**. Therefore, design the messaging logic so, that there are matching unique identifiers both in sent and received messages. Now the receiver always knows for which messages the response is for.** For example, the request messages may be identified with unique identification numbers (e.g. "‘Request #1"’ and "‘Request #2"’). The confirmation messages can use this number to identify the correct request (e.g. "‘Confirm #1"’ and "‘Reject #2"’). In this way, there is no risk of confusing the replies. In simpler cases this may be achieved by just using different confirmation actions in different phases of confirmation sequences (e.g. OK1, OK2). This simpler mechanism is illustrated in Figure 12.

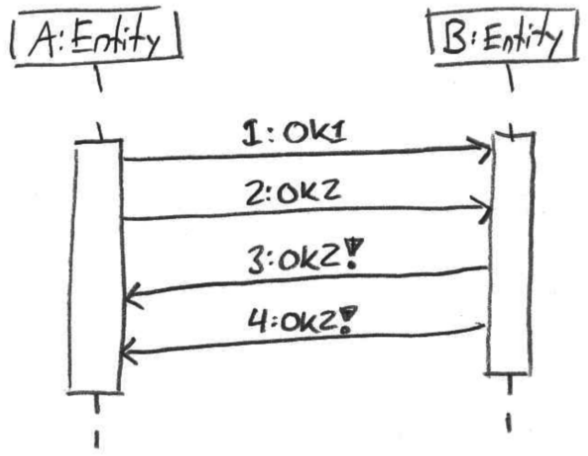


Figure 12 - The unique confirmations for request messages.

The sender should assign the message with unique identifier (UID). There are libraries available that can be utilized in generating the unique identifier. Furthermore, the sender can add their own ID to the UID to make sure that the probability to have two same identifiers is decreased. Other approach would be to use combination of time and sender’s ID. However, if this is the case, there should be outside time source.

Consequences

+ The messages can be sent asynchronously, as the confirmations are always unique.

+ The system debugging is easier as the message confirmations are easily linked to their corresponding requests.

+ The pattern allows confirmations be duplicated without fearing false positives.

– The unique confirmations require bookkeeping of the sent messages and their corresponding confirmations.

– The identifiers make the messages longer, thus wasting the bandwidth.

Resulting Context

This pattern results in a system where replies to multiple requests are not mixed as all requests have unique confirmation responses.

Related Patterns

VECTOR CLOCK FOR MESSAGES addresses the problem associated with finding out the correct order of events in a distributed system. ONE MESSAGE AT A TIME can be used to prevent a situation where multiple confirmation messages of same type need attention. ASYNCHRONOUS COMPLETION TOKEN [117] describes similar mechanism for asynchronous calls.

Known Usage

UNIQUE CONFIRMATION is used in many communication protocols, where all confirmations hold the identification number of the message they are a response to. For example, TCP/IP has on TCP level a special sequence number in each of the messages that binds replies to a known message.

5 USING DOMAIN KNOWLEDGE TO BOOST SOFTWARE ARCHITECTURE EVALUATION

This chapter is a verbatim copy of the following publication:

Veli-Pekka Eloranta and Kai Koskimies. 2010. Using domain knowledge to boost software architecture evaluation. In *Proceedings of the 4th European conference on Software architecture* (ECSA'10), Muhammad Ali Babar and Ian Gorton (Eds.). Springer-Verlag, Berlin, Heidelberg, 319-326.

5.1 Introduction

The benefits of software architecture evaluation have been widely recognized (e.g. [118]), and especially scenario-based evaluation methods [16] have become popular. However, a major problem of scenario-based evaluation methods is their need of resources, often hindering their usage especially in smaller development projects. For example, a full-scale ATAM evaluation of an average-sized system may easily take from 200 up to 400 man-hours [22], which can be hard to justify in many cases. We need techniques to make software architecture evaluation more efficient without losing its benefits and accuracy. This is particularly required in agile development.

The main idea of scenario-based evaluation methods is to identify the architectural solutions of the target system, refine and concretize the quality requirements of the system as scenarios, and analyze the prioritized scenarios against the architectural solutions. The problem of efficient scenario elicitation has been studied by several authors. In particular, the concept of a general scenario was introduced in [119] for the purpose of expressing reusable scenarios in a system-independent fashion. General scenarios are abstracted from previous evaluations and specialized for a particular system. The main contribution of this paper is a technique that supports the creation and specialization of general scenarios in the context of a particular domain. In contrast to general frameworks (e.g. [120]) or scenario categories ([121], [122]), we propose the use of a conceptual model of a system category as the basis of scenario elicitation. The main benefits of this approach are that the model provides a vehicle to relate scenarios with the basic concepts of the system, making it possible to derive and specialize general scenarios in a systematic way, and that the coverage of the scenarios can be evaluated in terms of the system concepts.

On the other hand, the identification of architectural solutions of a given system during evaluation can be hard as well. Often the architecture is poorly documented at the time of evaluation, or the documentation does not explicitly identify the architectural solutions. We demonstrate that the same conceptual model can be enriched with generic solutions (patterns) as well, supporting the identification and analysis of architectural solutions during the evaluation.

The domain we have investigated is mobile working machine control systems; however, it seems plausible that the proposed techniques are applicable to other domains as well. Our study is based on four full-scale ATAM-like architectural evaluations we have carried out in two Finnish companies manufacturing large industrial working machines: mining machines and forest harvesters. Additionally, the potential benefits of this approach were studied in a fifth evaluation.

5.2 System Concept Model for Architecture Evaluation

5.2.1 System Concept Model (SCM)

A central artifact in our approach is a conceptual model of a system or system category, called a *system concept model* (SCM). Essentially, SCM describes the basic concepts required to communicate about a system during the architecture evaluation. As such, the concepts in SCM can be related to the software architecture (e.g. archetypes [123]), hardware in an embedded system, input and output devices, domain concepts, system resources and external artifacts relevant for the system etc. The scope of SCM can vary, and we deliberately leave it open: SCM should serve as a pragmatic description of the conceptual world of architecture evaluation, rather than as a formal specification. SCM differs from a traditional domain model in that it contains technical concepts as well; on the other hand it describes the logical relationships between concepts rather than system architecture. Minimally, SCM should contain the system-related concepts appearing in the scenarios and the core concepts of the system architecture, i.e. archetypes. SCM can be given for a single system, or (more usefully) for a system category. We assume here that SCM is given as a UML class diagram.

5.2.2 Process of Scenario Elicitation Using SCM

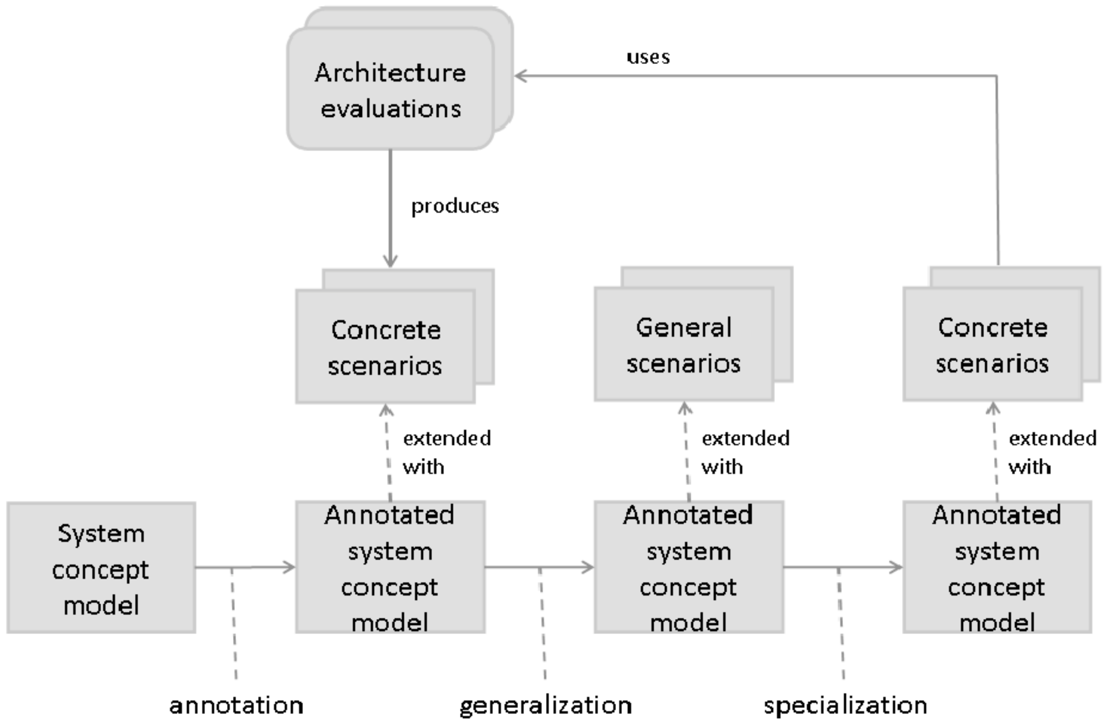


Figure 13 - Process to produce general scenarios and new concrete scenarios using SCM

Our process for exploiting general scenarios in architecture evaluation is illustrated in Figure 13. The first step in our technique is to carry out an architecture evaluation (e.g. ATAM), which produces a set of concrete scenarios. SCM is assumed either to exist before the evaluation or to be constructed after (or during) the evaluation. If SCM exists before the evaluation, it can be used to facilitate the communication between stakeholders during the evaluation, providing the evaluators and stakeholders with the same domain-specific vocabulary.

The second step is to annotate SCM with concrete scenarios. This means that keywords that can be found from SCM are searched from the scenario descriptions and the scenario is linked with the corresponding classes in SCM. Often it is useful to mark implicit relationships as well, that is, even if a concept is not mentioned in the scenario but is known to be closely related with the scenario, the corresponding link can be added for the scenario. The scenarios are presented as UML collaborations in SCM. In principle, it should be possible to attach every scenario with some concepts in SCM; in case there is a scenario with no links, it is a sign that SCM lacks concepts that are essential for understanding the system, and it should be completed with such concepts.

Once SCM is annotated with concrete scenarios, general scenarios can be derived in the third step (Figure 13). For each concrete scenario that is linked with certain specialized concepts, the generalization relationships of these concepts are followed upwards until a suitable level of abstraction is reached, and a new general scenario is created and linked with those higher-level concepts. Naturally, a scenario can be associated with arbitrary many concepts.

For example, assume that a concrete scenario is "The joystick can be operated via Bluetooth, the necessary change can be made in a month". This scenario involves two specific concepts: the joystick and Bluetooth. Both are specific kinds of more general concepts, namely a user control device and a wireless communication protocol. Thus, a general scenario could be "A user control device can be operated via a wireless communication protocol, the necessary change can be made in x months". Note that the response part cannot be generalized in the same manner, but it has to be expressed for example using variable names for specific numbers or using a template list for responses.

This process is by no means fully mechanical, since there can be several levels of more general concepts in SCM, or possibly none. In the former case, it may be possible and useful to create several general scenarios at different levels of abstraction. In the latter case, the lack of a generalization relationship may indicate that SCM needs new generalized concepts. It is also quite possible that some concrete scenarios simply cannot be generalized in a sensible manner. If a concrete scenario is linked only with the subclasses of a single concept (say, "changing CAN bus to Ethernet"), the corresponding general scenario will be linked with a single concept, and the general scenario has to be formed in a different way (e.g. "changing the communication bus type").

In the fourth step scenarios are specialized by traversing the generalization hierarchy downwards, selecting the subclasses that are relevant for the target system. For example, in the case of the general scenario above, the new target could be the control system of a machine that does not have a joystick but a touch pad. In the case of the new machine, wireless control is also interesting, but Bluetooth is not an option since the remote control is too far away. Instead, a plausible option might be a wireless internet connection. Thus, traversing downwards in the hierarchy, touch pad and WLAN are seen as relevant subclasses, and the specialized scenario becomes "The touch pad control can be operated via WLAN; the necessary change can be made in 3 months". The response part is specialized separately, assigning suitable numbers in the place of the variables. The new concrete scenario is then included in the architecture evaluation in the normal way, prioritized and possibly analyzed. The relevant quality attribute for the scenario is readily available as the general scenario is already mapped to the quality attribute through the original concrete scenario.

Besides scenario elicitation, SCM can be used also for estimating scenario coverage. SCM can be divided into major subareas, and ideally the scenarios should be fairly evenly associated with the different subareas. If the scenarios are mainly attached to only certain subareas, or if one subarea is without any scenarios, it is possible that the scenario set is somewhat misaligned and new scenarios should be elicited for the empty or scarce subareas. For example, in the domain studied in this work, SCM was divided into 9 subareas (e.g. Messaging, User interfaces, Remote access, etc.).

5.2.3 Annotating SCM with Solutions

On the solution side, the counterpart of a general scenario is a pattern: a pattern describes a solution in a system-independent manner. In the same way as a concrete scenario is an instance of a general scenario, a solution in the architecture is often (although not always) an instance of a pattern documented as part of a pattern language. Similarly to general scenarios, patterns can be and often are domain-specific.

Assuming that SCM is available, patterns can be linked to the concepts in the model in the same way as general scenarios: if a pattern refers to a concept in SCM, it is linked to the corresponding class. We have used UML collaborations also for denoting patterns in SCM. This kind of model annotation can be useful for the pattern language itself, as it provides a domain-oriented structuring for the pattern language, but it makes sense also from the viewpoint of architecture evaluation. In particular, the representation of both the problem domain (general scenarios) and the solution domain (patterns) in the same model together with the essential system concepts facilitates discussions about the relationships of problems and solutions. We will present concrete examples in the next section.

Since patterns describe general solutions in a system-independent way, they will be typically linked to the base classes in the inheritance hierarchies. For example, if a pattern describes a solution to create an abstraction for a bus, it is linked to the Bus class; if a pattern describes a communication mechanism between nodes based on a bus, it is linked with Bus and Node classes etc. Basically, the nouns in the context and solution parts of the pattern description are potential concepts to be linked with the pattern in SCM. This process can also lead to the observation that new classes need to be added to SCM.

SCM, annotated with scenarios and patterns, can be utilized in the analysis phase of ATAM in particular for more accurate solution identification. When a scenario has been proposed in such a way that it is linked with the same concepts as a pattern, an instance of the pattern may exist in the architecture. This may have been overlooked in the identification of architectural solutions: either the solution has not been recognized at all or it has been recognized but not seen as an instance of a pattern. In both cases, recognizing a solution as an instance of a pattern makes the analysis faster and more reliable, as the potential non-risks and risks are readily available as positive and negative consequences in the documentation of the pattern, respectively.

5.3 Applying the Approach

We have applied the proposed techniques in the domain of mobile working machine control systems. Based on four full-scale ATAM-like architectural evaluations carried out in Finnish machine industry, SCM was created, and the elicited scenarios were generalized and attached to SCM. During those evaluations, patterns were identified, documented and organized into a pattern language, and the patterns were attached to the SCM as well.

After that, a fifth evaluation was carried out. In this evaluation, the aim was to evaluate the potential usability of SCM enriched with general scenarios and patterns. However, since the primary purpose of the evaluation was to produce normal analysis results for the company, we could not risk the evaluation process by trying entirely new techniques in the elicitation of scenarios. Instead, scenarios were created in the conventional way during the evaluation, and compared afterwards against a set of concrete scenarios that was created before the evaluation on the basis of the general scenarios and SCM refined according to the specialized concepts of the target system.

A part of the resulting SCM annotated with patterns for mobile working machine control systems is depicted in Figure 14. A basic source of information for SCM was the set of scenarios elicited during the four evaluations. Essentially, SCM was created by gathering first a list of generally known basic concepts for the domain such as bus, node, controller etc. After that all scenarios were studied and more concepts were found in the phrasing of the scenarios. Additionally, the list of concepts was augmented with concrete examples of the existing concepts. We decided to leave attributes out from SCM

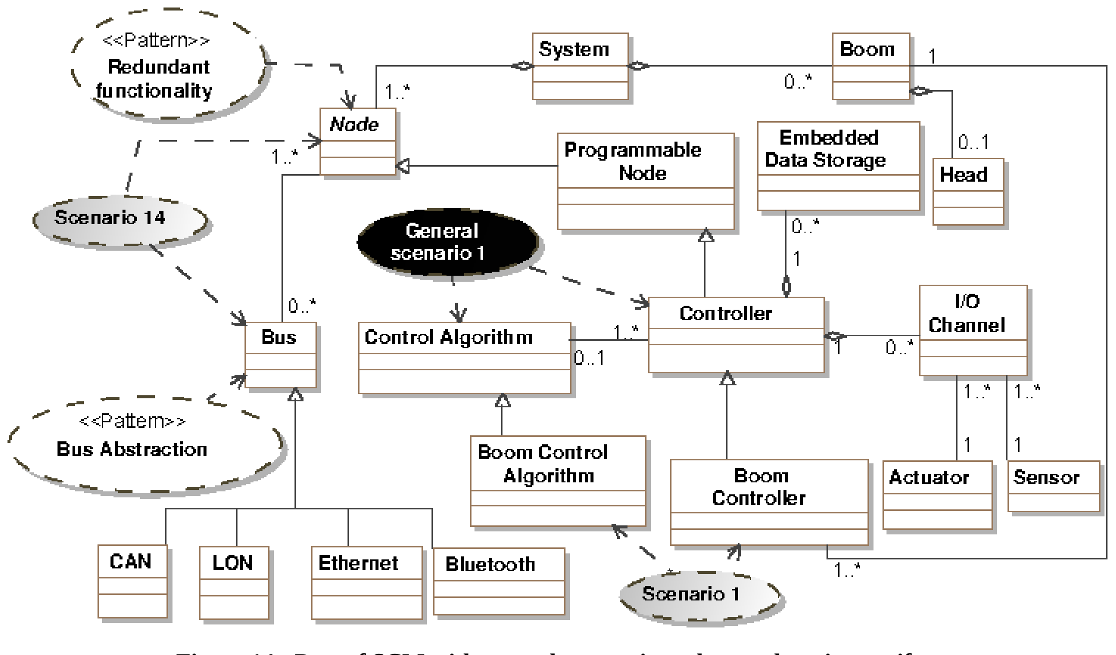


Figure 14 - Part of SCM with example scenario and some domain-specifc patterns

as they are not required in our context. The entire SCM comprises of 71 classes.

Once we had created SCM, we annotated it with concrete scenarios from previous architecture evaluations. The concrete scenario (Scenario 1) has been attached as UML collaboration to classes Boom Control Algorithm and Boom Controller in Figure 14 as the scenario is related to the control algorithm of the boom, residing in the boom controller. Scenario description is as follows:

Boom's control algorithm geometry (spherical, cartesian, etc.) can be changed

> *by changing only one parameter.*

Note that here we have used an implicit relationship between a scenario and a concept: the controller is not explicitly mentioned in the scenario, but the location of the control algorithm is considered essential from the viewpoint of the scenario.

Next, general scenarios were developed using SCM. For example, we can see from SCM that Boom Control Algorithm is inherited from Control Algorithm and Boom Controller is inherited from Controller. By following this generalization hierarchy and applying little rephrasing we can form a general scenario for Scenario 1 as follows:

> *A control algorithm in a controller can be changed to another algorithm version by changing x parameters.*

The resulting general scenario is linked with Control Algorithm and Controller, as shown in Figure 14. We created in this way 52 general scenarios from the concrete scenarios elicited in the evaluations.

The fifth evaluation was carried out as a normal ATAM-like evaluation. Scenarios were elicited by the stakeholders only, without any influence from our side. After the evaluation sessions we verified how many of the scenarios in the fifth evaluation could have been actually "guessed", at least approximately, before the evaluation sessions using the general scenarios developed earlier.

We created 57 concrete scenarios as specializations of the general scenarios before the fifth evaluation and compared these to the 22 well-formed concrete scenarios which were elicited during the evaluation. We found out that 17 of the concrete scenarios created during the evaluation were essentially the same that we have generated beforehand. This means that 77 percent of the correctly formed scenarios could have been created, in a more or less accurate form, prior to the evaluation. Bass et al. [119] reports that as much as 91 percent of created concrete scenarios could be mapped to general scenarios extracted from five evaluations, suggesting that we can reach a fairly good level of coverage of general scenarios with our technique.

During the four evaluations we gathered a pattern language of 45 patterns for the domain of distributed embedded control system [124]. These patterns were linked to the domain concepts in SCM using (general) scenarios, resulting in general scenarios and patterns linked with the classes of SCM. When annotated both with scenarios and patterns, SCM provides useful information about the interplay of scenarios and patterns during the evaluation. For example, in Figure 14 scenario 14 describes a situation where the bus is changed from CAN to Ethernet in a week. If this scenario is analyzed, evaluators can look if there is a solution described by BUS ABSTRACTION or REDUNDANT FUNCTIONALITY pattern. If the solution provided by the pattern is not used, evaluators can ask whether this was intentional or not, and if not, why. In some cases, the architect just may have ignored such a solution.

5.4 Related Work

Babar and Biffl [121] divide the techniques to steer scenario creation into two main types: top-down elicitation is guided by a predefined classification of scenario categories, while bottom-up techniques aim at extracting the scenarios directly from stakeholders using queries, interviews and brainstorming. Our technique falls clearly in the former category. In that category, the approaches that come near to ours are the idea of general scenarios discussed previously ([125], [119]), the use of various (domain-independent) matrix forms like in [120] or [126], and the use of change categories (e.g. [122]). In addition, slightly similar approach, than ours has been taken in [127] to share architectural knowledge of quantitative analysis. Another way to classify scenario elicitation techniques is the level of domain independence: domain-specific techniques build an instrument guiding the scenario elicitation on the basis of a particular domain, while domain independent techniques rely on generic guiding instruments. Our technique belongs obviously to the top-down domain-specific category. Basically, the techniques in this category require some additional work to build the domain-specific instrument, but they can provide stronger support for scenario elicitation. This has been actually empirically verified by Babar and Biffl [121] in the case of change categories: they have shown that domain specific categories of software changes can help stakeholders to generate better quality scenarios.

According to Bengtsson and Bosch [122], a classification of change categories can guide the scenario elicitation process towards high quality scenarios. The classification of change categories is derived from the target domain. Change categories also help to decide scenario coverage: when there are scenarios in each category, the amount of scenarios can be regarded sufficient [122], [128].When compared to our technique, change categories are more focused, addressing quality attributes like maintainability and modifiability. Our technique is in principle independent of quality attributes.

5.5 Conclusions

In this paper we have shown how to take advantage of domain knowledge in a scenario-based architecture evaluation process using SCM, a model that captures the conceptual world of architecture evaluation. The approach is particularly suitable if a number of evaluations are carried out for a well-understood domain. Our work leaves a number of open research questions. A practical problem is related to architectural knowledge management (e.g. [128]): how to support the collecting and management of the large amount of information pertaining to our approach, including SCM, scenarios, and patterns in a particular domain or company context? Another related problem is how to manage both architectural knowledge management and the actual architectural evaluations in an agile project context with minimal effort, without losing the benefits? These topics will be on our future research agenda.

6 DCAR – DECISION-CENTRIC ARCHITECTURE REVIEWS

This chapter is a verbatim copy of the following publication:

Uwe van Heesch, Veli-Pekka Eloranta, Paris Avgeriou, Kai Koskimies, and Neil Harrison. 2014. DCAR - Decision-Centric Architecture Reviews. *IEEE Software*, IEEE Computer Society, pages 69-76.

Published here with kind permission of IEEE Computer Society

6.1 Introduction

Software architecture that is poorly designed or carelessly cobbled together may cause an entire software project to fail. Therefore, it is important to evaluate software architecture early on in the development. Various software architecture evaluation methods have been proposed to uncover architectural problems in a systematic way [60], [129]. The most popular evaluation methods are scenario-based, e.g. ATAM [16]. In general, architecture evaluation has several benefits. Most importantly, a problem or a risk identified early in the development process can be easily fixed or mitigated, compared to a problem that is found late, e.g. in the testing or integration phase, or even during maintenance [23]. Furthermore, architecture evaluations encourage communication among the involved stakeholders that would not take place otherwise.

Despite these benefits, architecture evaluation is not widely adopted in the industry today [60], [129]; most organizations are aware of its benefits, but very few practice it. A study with software architects uncovered typical factors that influence the architecture evaluation practices of organizations [130]. According to the study, management commitment, company-wide evaluation standards, a funding model, and appropriate training for the involved staff members are prerequisites for establishing architecture evaluations in a company. Such prerequisites are often not met. Furthermore, the increasingly popular agile development approaches do not encourage the use of architecture evaluation methods, which often consume a considerable amount of time and resources.

In order to lower the threshold of industrial adoption of architecture evaluations, we developed a new evaluation method called DCAR (Decision-Centric Architecture Review). DCAR was developed bottom-up, based on our experience on performing architecture evaluations in the industry and observing what works well in practice. This lead to two high-level requirements. First, the method needs to be lightweight in terms of required time and resources. Second, the method must support the evaluation of software architecture decision-by-decision, allowing systematic analysis and recording of the rationale behind architecture decisions. The latter requirement makes the method different from scenario-based methods, which aim at testing software architecture against scenarios that refine the major quality requirements of the system.

DCAR is decision-centric in the sense that the evaluation is carried out by selecting a set of decisions which are analyzed in the context of all relevant project- and company-specific decision forces. A decision force or force in short, is any non-trivial influence on an architect who is looking for a solution to an architectural problem. The concept of forces is elaborated below. DCAR can be used for any set of architectural decisions, of any type. It is applicable for all types of software-intensive systems and domains.

We have carried out multiple DCAR evaluations in the industry. Our experience indicates that an average DCAR session takes half a day, requiring the presence of 3-5 members of the project team, including the chief architect. Thus, the total amount of company working hours is less than three person-days plus another two person-days for the review team. This makes DCAR suitable for projects that do not have the budget, schedule, or stakeholders available for full-fledged architectural evaluations. DCAR is also pertinent for projects that wish to perform architecture evaluation to justify a set of architecture decisions rather than for ensuring that a whole system satisfies its quality requirements.

In the following, we present a short profile of DCAR. The profile can be used to compare DCAR to other evaluation methods that were described using the classification used by Bass and Nord [129].

Inputs for evaluation: Informal description of requirements, business drivers and architectural design.

Output: Risks, issues, and a thorough documentation of the evaluated decisions and their decision forces.

Reviewers: Company-internal or external reviewers.

Priority setting of decisions: During the review.

Social interaction: Face to face meeting.

Resources required: 2-4 reviewers, architect, developers, and business representative.

Knowledge of evaluators: General knowledge about software architecture.

Tools or automation: Templates, Wiki, UML tool.

Evaluation objectives: Determine the soundness of architectural decisions that were made.

Scope: A set of specific architecture decisions.

Schedule: Half a day preparation and post processing, half a day review session.

Project phase: Within or after the architectural design is finalized.

6.2 Architecture Decisions

DCAR focuses on evaluating specific architecture decisions, selected by stakeholders under the assistance of the review team. Understanding architecture decisions and the rationale behind them is crucial for continuously ensuring the integrity of a system.

Architecture decisions are the fundamental choices, an architect has to make, concerning the overall structure or externally visible properties of a software system [80]. Typical examples of such decisions are the choice of an architectural pattern or style, the selection of a middleware framework, or the decision not to use open source components for licensing considerations.

Architecture decisions are not isolated; they can be seen as a web of interrelated decisions that can depend on, support, or contradict each other. Some decisions must be combined to achieve a desired property; other decisions are solely made to compensate the negative impact on a desired property, caused by a decision made to achieve a different property. As an example, an architect could decide to use an in-memory database to achieve short response times. This decision has a negative impact on reliability, which, in addition to short response times, is another desired property of the system. To compensate this negative impact, the architect could decide to use redundant power supplies, or to replicate the database and the hardware and use the replica as a hot spare. The decisions to use redundant power supply and a hot spare would then be caused by the decision to use an in-memory database.

In DCAR, the participants identify the architecture decisions made and clarify their interrelationships. This is primarily done for two reasons: First, understanding the relationships helps to identify influential decisions that have wide-ranging consequences for large parts of the architecture. Second, when a specific decision is evaluated, it is important to consider its related decisions as well (as illustrated by the *previous in-memory database* example).

6.2.1 Decision Forces

Several factors need to be taken into consideration to evaluate an architecture decision. Apart from architecturally significant functional and non-functional requirements, such factors include constraints, risks, political or organizational considerations, personal preference or experience of the architect and the development team, or business goals like quick time-to-market and low price. We call these factors decision forces [131], because of the similarities with forces in physics. Each force has a direction and a magnitude. It either pushes an architect towards a specific solution, or it pushes the architect away from that solution.

In order to evaluate an architectural solution, the related decisions also need to be contemplated and considered as decision forces. In their totality, forces reveal the entire context in which a decision is made. As some of them can be conflicting, or orthogonal to each other, an architect has to balance all forces to make the best possible decision.

Figure 15 illustrates the concept of forces using the in-memory database decision, introduced above. In this particular case, the forces in favor of the in-memory database outweigh the forces against it. DCAR explores the entire rationale behind decisions by means of forces. After identifying forces, the review participants examine if the rationale behind the evaluated decision is still valid in the current context. This is important, because forces are not immutable; not only requirements keep changing, but the tactical orientation of the company may evolve, laws and regulations may have changed, or new technologies could exist that would offer a better solution to a design problem at hand. Such changes in the design context may change the magnitude of the

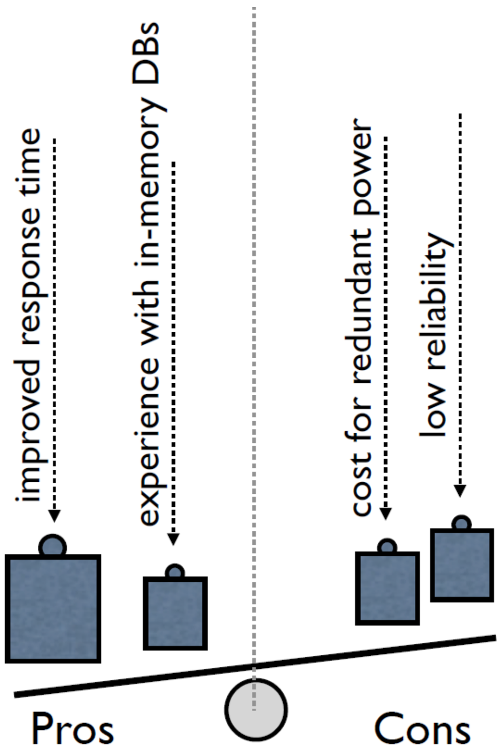


Figure 15 - Some of the forces for the in-memory database decision

forces, or even introduce new forces and make some of the old forces obsolete. In the new design context, if the negative forces outweigh the positive forces, then the reviewers recommend to reconsider the decision.

6.3 Introducing DCAR

6.3.1 Company Participants and Review Team

To achieve best results, DCAR requires the participation of the lead architect and one or two members from the development team with different roles and responsibilities in the software project. Additionally, somebody has to represent the management and customer perspective. This is important, because some decisions have to be assessed from an enterprise-wide perspective rather, than taking only project-specific forces into account.

The review can be done by external reviewers, or by an organization’s own people who are not involved in the project under review. The members of the review team need to have experience in designing software architecture; ideally, but not necessarily, in the same domain as the system under review.

6.3.2 Essential Steps

In this subsection, we briefly present how DCAR is carried out in practice[2].

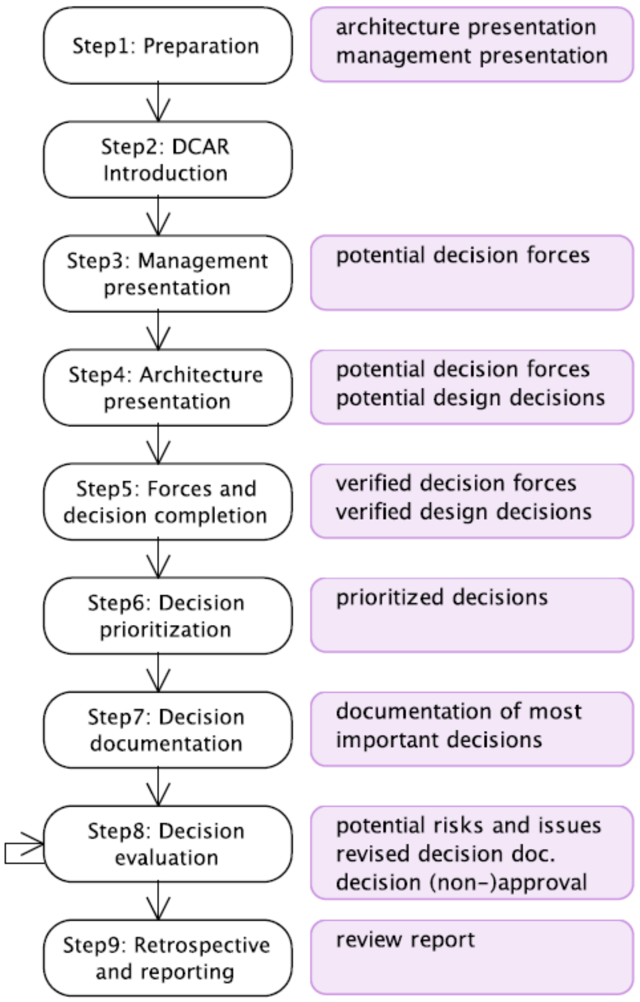

Step1: Preparation
architecture presentation
management presentation
Step2: DCAR Introduction
Step3: Management presentation
potential decision forces
Step4: Architecture presentation
potential decision forces
potential design decisions
Step5: Forces and decision completion
verified decision forces
verified design decisions
Step6: Decision prioritization
prioritized decisions
Step7: Decision documentation
documentation of most important decisions
Step8: Decision evaluation
potential risks and issues
revised decision doc.
decision (non-)approval
Step9: Retrospective and reporting
review report

Figure 16 - DCAR steps and produced artifacts during each step.

Figure 16 shows the main steps of DCAR, as well as the produced artifacts (boxes on the right). In the following, each of the steps is briefly described. Step 1 is carried out offline; all other steps are carried out during one evaluation session, in which all participants gather in one room.

Step 1) A date for the DCAR session is settled, and the stakeholders are invited to participate. The lead architect of the system[3] is asked to prepare a presentation of the archi-

[2] Additional information on the method can be found online http://www.dcar-evaluation.com

[3] DCAR cannot only be used to evaluate whole systems, but also for major and minor sub-systems. For the sake of simplicity, we refer to all of them as *systems*.

tecture. This presentation should contain the most important architectural requirements, high-level views on the architecture, used architectural approaches like patterns or styles, and the used technologies like database management systems or middleware servers. The representative of the management and customer perspective is asked to prepare a presentation describing the software product and its domain, the business environment, market differentiators, and driving business requirements and constraints. Templates for both presentations can be found on the previously mentioned DCAR website.

The slides for the presentations are sent to the review team prior to the evaluation session, so that they can prepare for the evaluation. In particular, the reviewers study the material to elicit potential architecture decisions and decision forces. Additional system documentation is not mandatory, but any additional material that can be used by the reviewers to understand the system upfront is helpful.

Step 2) The evaluation session starts with an introductory presentation of the DCAR method to all participants. This includes the schedule of the day, introduction of the DCAR steps, the scope of the evaluation, possible outcomes, and the roles and responsibilities of all participants. The DCAR website provides an example of such a presentation.

Step 3) The representative of the management and customer perspective gives a short presentation using the slides prepared in Step 1. In our experience, 15-20 minutes suffice, but more time can be used if the schedule allows it. The main purpose of this step is to allow the reviewers to elicit business-related decision forces that must be taken into consideration during the evaluation. The review team notes down potential forces during the presentations and asks questions to elicit additional forces. The representative of the management and customer perspective does not need to be present during the rest of the session; however, it is beneficial as he or she can provide additional insights during the decision analysis.

Step 4) The lead architect uses the slides prepared in Step 1 to introduce the architecture to all DCAR participants. In our own industrial DCAR sessions, we reserved between 45 and 60 minutes for this presentation. The goal is to give all participants a good mental picture of the architecture. It is supposed to be highly interactive. The review team and the other participants ask questions to complete and verify their understanding of the system. During this step, the reviewers revise and complete the list of architecture decisions they had identified as a preparation in Step 1. Identifying architecture decisions requires some experience. As a starting point, reviewers can focus on used technologies like servers, frameworks, or third-party libraries. Additionally, it has been a good practice to search for applied patterns in the architecture [132].

Apart from capturing architecture decisions, the reviewers revise and complete the list of forces they had identified in Steps 1 and 2. Forces can be documented as informal statements. Both, decisions and forces, are revisited in the next step.

Step 5) At this stage, the reviewers have assembled a preliminary list of architecture decisions and decision forces. The goal of Step 5 is two-fold: Clarify the architecture decisions and their relationships, and complete and verify the forces relevant to these decisions. To support the clarification of the decision relationships, a decisions relationship diagram [81] is created by one of the reviewers during the review session. The diagram is constantly revised during the previous steps. Figure 17 shows an excerpt of such a diagram. Each decision is represented by an ellipse that contains a short descriptive name for the decision. It is important to use the company's own vocabulary in the names, to make sure that reviewers and company stakeholders have the same understanding of the applied architectural solution. In the beginning, each of the decisions collected by the reviewers in the previous step is represented in the diagram. After all participants gained a collective understanding of the decisions, the relationships are established. In a relationship diagram, they are represented by a directed line. Although more relationship types exist [131], in an architecture review, the only important relationships are *caused by* and *depends on*. The relationships help the reviewers and stakeholders to estimate the importance of each decision. Decisions being the origin of many dependencies have a central role and must be seen critically. Relationships are also helpful to understand which decisions must be taken into consideration as decision forces for other decisions. Relationship diagrams can be created using any UML tool. A template for such a diagram can be downloaded from the DCAR website.

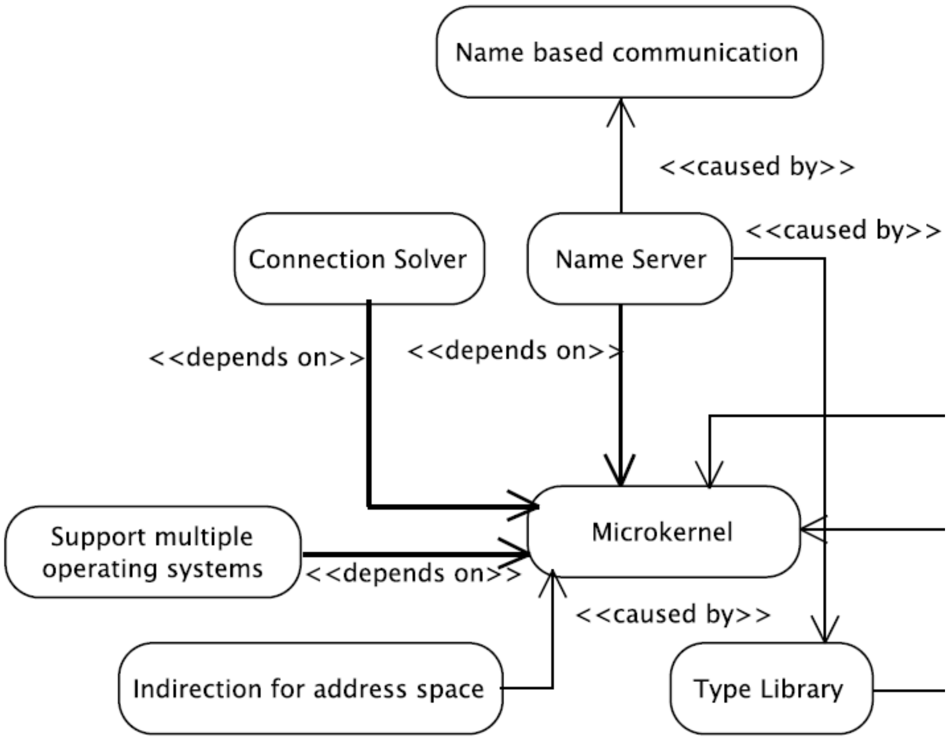


Figure 17 - Excerpt from a relationship view created in a DCAR session.

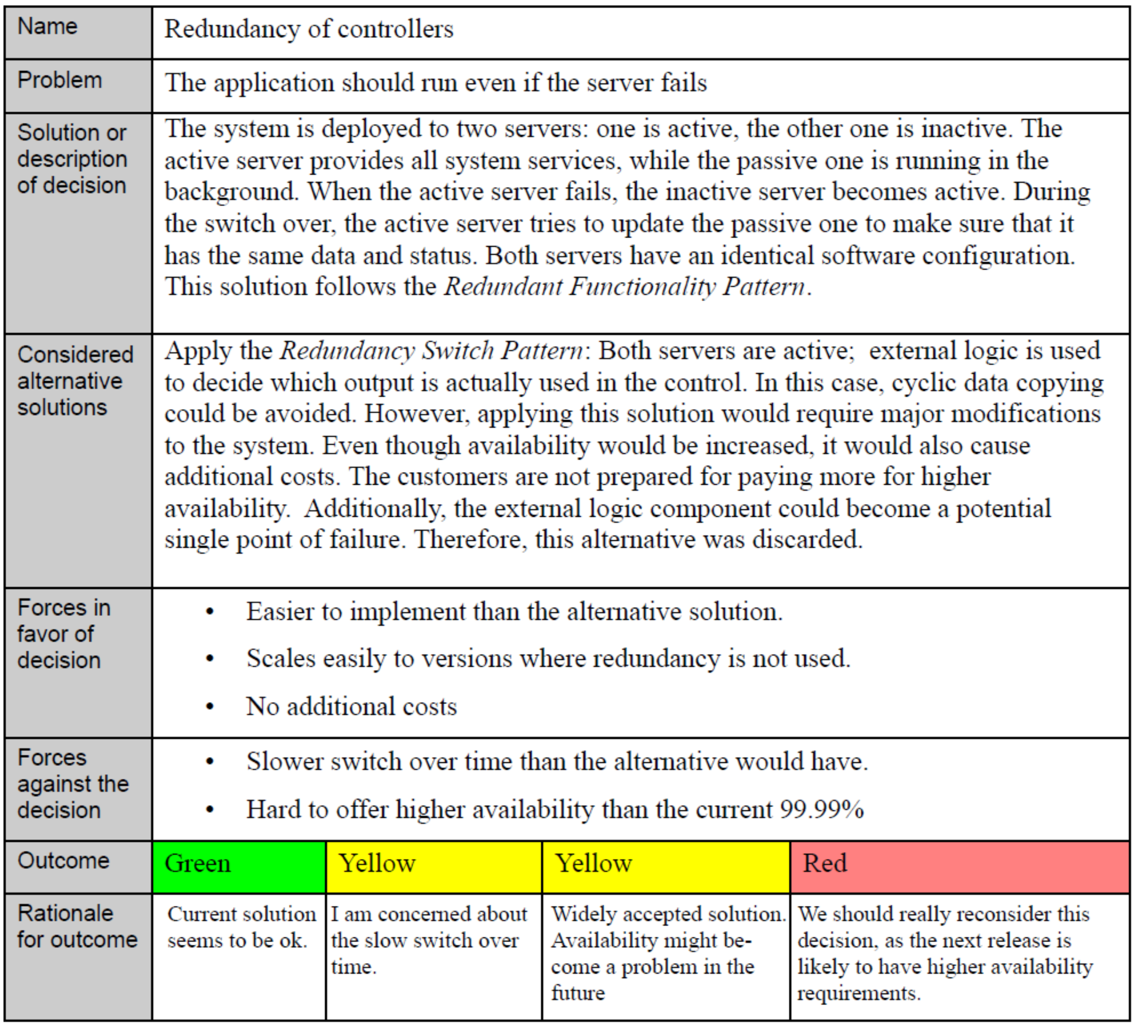
The forces are presented as a bullet list. They should be formulated unambiguously using domain specific vocabulary. Example forces from the machine control domain are *Firmware level design and implementation should be sourced out, as it is not our core business.*, or *We have a lot of in-house experience with the CANOpen protocol*. The review team discusses and completes the list of forces with the company participants.

Step 6) Usually, the number of decisions elicited in the previous steps is too large to discuss each of them during the review. Therefore, the stakeholders have to negotiate which decisions will be reviewed in the following steps. The criteria for selecting which decisions will be reviewed is context dependent and has to be negotiated between the stakeholders. The criteria could be mission-critical decisions, decisions known to bear risks, or decisions causing high costs, for instance.

We use the following procedure to prioritize decisions: Each participant gets 100 points. The points can be distributed freely over the decisions, based on the previously agreed criteria for the importance of decisions. Then the points of all participants are summed up and then the rationale behind each person's ratings is discussed. The decisions with the highest ratings are taken to the next steps. In our experience, the number of decisions that can be discussed effectively in half a day is seven to ten.

Step 7) The architect and the other company participants document the set of decisions that received the highest ratings in the previous step. Therefore, each of them selects two or three decisions he or she is knowledgeable about. The decisions are documented by describing the applied architectural solution, the problem or issue it solves, known alternative solutions, and the forces that need to be considered to evaluate the decision. The stakeholders use the list of forces assembled in the previous steps to make sure that they don't forget important ones, but they can also think of new forces. A member of the review team supports the stakeholders during this process. This is particularly important, because some of the participants may be inexperienced in decision documentation.

Figure 18 shows an example of the decision documentation template used in DCAR; other established templates could be used alternatively, e.g. the template defined by [80] or [132].



Name	Redundancy of controllers			
Problem	The application should run even if the server fails			
Solution or description of decision	The system is deployed to two servers: one is active, the other one is inactive. The active server provides all system services, while the passive one is running in the background. When the active server fails, the inactive server becomes active. During the switch over, the active server tries to update the passive one to make sure that it has the same data and status. Both servers have an identical software configuration. This solution follows the *Redundant Functionality Pattern*.			
Considered alternative solutions	Apply the *Redundancy Switch Pattern*: Both servers are active; external logic is used to decide which output is actually used in the control. In this case, cyclic data copying could be avoided. However, applying this solution would require major modifications to the system. Even though availability would be increased, it would also cause additional costs. The customers are not prepared for paying more for higher availability. Additionally, the external logic component could become a potential single point of failure. Therefore, this alternative was discarded.			
Forces in favor of decision	• Easier to implement than the alternative solution. • Scales easily to versions where redundancy is not used. • No additional costs			
Forces against the decision	• Slower switch over time than the alternative would have. • Hard to offer higher availability than the current 99.99%			
Outcome	Green	Yellow	Yellow	Red
Rationale for outcome	Current solution seems to be ok.	I am concerned about the slow switch over time.	Widely accepted solution. Availability might become a problem in the future	We should really reconsider this decision, as the next release is likely to have higher availability requirements.

Figure 18 - Example of documented and analyzed decision.

Step 8) The documented decisions (seven to ten on average) are subsequently evaluated, starting with the decision with the highest priority. The participant, who documented the current decision presents it briefly. After that, the company participants together with the reviewers challenge the decision by identifying additional forces against the chosen solution. They use the elicited decision forces and the decision relationship diagram to understand the context of the decision, i.e. related decisions and relevant forces in favor or against the applied solution. The documentation of the decisions and the decision relationship diagram are continuously updated by one of the reviewers during this step. All participants discuss whether the forces in favor of the decision outweigh the forces against it.

Finally, all participants decide in a voting procedure, whether the decision is good, acceptable, or if it has to be reconsidered. Figure 18 shows the result of an evaluated decision, created during a DCAR session. The traffic light colors indicate the ratings of all participants; green for *good*, yellow for *acceptable*, and red for *has to be reconsidered*.

Additionally, it shows justifications for the votes, given by each voter (“Rationale for outcome”).

During the whole discussion, the reviewers note down potential issues or risks that were mentioned. Each decision is discussed for approximately 15-20 minutes. In our experience, the quality of discussion diminishes at some point. If a decision requires more than 20 minutes, it can be flagged as a point for future analysis.

Step 9) After all of the selected decisions were evaluated, the review team collects all notes and artifacts created during the evaluation session. They serve as input for the evaluation report. The review team writes a report within two weeks from the review session. The report is discussed with the architect for verification and eventually refined by the review team. In our own DCAR sessions, the report was prepared by the review team on the next day. The advantage of an early report is that the review team and the architect can still vividly remember the discussions held during the evaluation session.

6.4 Experiences

DCAR has been developed in cooperation with industrial partners from the distributed control system domain. As described in Section 6.1, however, it is by no means restricted to this domain. Since the initial version, DCAR has been applied and refined in five large software projects. In this section, we report on our findings from three industrial DCAR sessions, which were conducted in different projects at Metso Automation in Tampere, Finland.

Table 2 - Descriptive statistics

Variable	**Value**
Avg. system size	600 000 SLOC
Avg. no of elicited decision after step 5	21 decisions
Avg. no of decisions documented in step 7	9 decisions
Avg. no of decisions evaluated in step 8	7 decisions
Avg. no of reviewers	4 persons
Avg. no of company stakeholders	4 persons
Avg. effort for reviewer team	50 person-hours
Avg. effort for company stakeholders	23 person-hours

Table 2 shows descriptive statistics about the DCAR executions. The systems under study in these evaluations came from the process automation domain. Each of the three DCAR sessions, summarized here, was conducted within five hours. The effort that the company stakeholders had to spent on the reviews reveals the time spent by the participants for preparation, taking part in the evaluation sessions, and reviewing the evaluation report.

To gather feedback on the participants’ perception of DCAR, we carried out interviews with a subset of the participants, including the chief architect. Apart from the chief architect, who naturally knows the architecture best, all interviewees mentioned that they got a good overview of the system’s architecture; something they were missing in their daily work, because they were only responsible for smaller sub-systems. They also stated that they liked that all important decisions, even if they were considered stable, were brought into question for the purpose of the evaluation. The prioritization procedure in DCAR step 6 made sure that bias on behalf of the decision maker or the responsible architect was reduced. Systematically discussing decisions in a group also helped them to understand different points of view that need to be considered in the context of a decision.

Generally, the stakeholders reported that the interaction between the stakeholders, and the discussion with the review team as external contributors was one of the most valuable advantages of the evaluation session. The chief architect noted that the evaluation report, produced by the review team, is a valuable supplement to the existing system documentation. The interviewees estimated that the decisions elicited during the evaluation roughly covered the most important 75% of all significant architecture decisions; this was regarded as an excellent result given the short amount of time invested in the evaluation.

The success of a DCAR depends on the stakeholders’ understanding of the concepts of architecture decisions and decision forces. Therefore, we explicitly addressed these issues in the interviews. Although all interviewees were either already familiar with both terms, or grasped the concepts quickly during the DCAR introduction in Step 2, some of them mentioned that the time given for the documentation of decisions in Step 7 was too short. This was particularly the case for stakeholders, who had never systematically documented architecture decisions before. They proposed to tackle this problem by providing examples of documented decisions prior to the evaluation.

During the evaluations, we observed that the documentation of reasoning, i.e. forces in favor or against a specific solution, was especially challenging for some of the participants. Therefore, in the later evaluations, we provided decision examples with a list of typical decision forces in the domain at hand and found out that it alleviates the problem.

These positive experiences in the past evaluations, and the continuous interest of other industrial partners to hold more evaluations in the future, show that DCAR helps organizations to adopt architectural evaluations as part of their practices. Additional empirical studies will be conducted to provide evidence concerning how far DCAR indeed lowers the threshold for industrial adoption of architecture evaluations.

PART IV – AGILE AND SOFTWARE ARCHITECTURE KNOWLEDGE MAN-AGEMENT

This part presents software architecture practices in agile software development. TopDocs architecture knowledge base is introduced to manage software architecture and domain knowledge. Furthermore, models how to align lightweight software architecture knowledge management to agile software development are described.

7 SOFTWARE ARCHITECTURE PRACTICES IN AGILE ENTERPRISES

This chapter is a verbatim copy of the following publication:

Veli-Pekka Eloranta and Kai Koskimies. 2013. Software Architecture Practices in Agile Enterprises. Book chapter in *Aligning Enterprise, System, and Software Architectures,* ed. Ivan Mistrik, Antony Tang, Rami Bahsoon and Judith A. Stafford, ISBN 978-1466-62199-2, pages 230-249.

Published here with kind permission IGI Global.

7.1 Introduction

During the last decade, software industry has widely embraced agile approaches in the development of software systems. Today, the most popular agile approach applied in industrial software development projects is Scrum [133] according to a recent survey [134] Scrum has 78% market share of agile methods in software industry. Even though Scrum is not particularly intended for software development, it has proved to be very suitable for this kind of production work, improving the quality of the products as well (e.g. [135] and [136]). Scrum combines agile principles [7] with lean manufacturing philosophy [89]. Essentially, Scrum defines how the work is decomposed into tasks, how the tasks are ordered, managed, and carried out, which are the roles of the project members, and how persons in these roles interact during the development process. A central concept of Scrum is a sprint, a working period during which the team produces a potentially shippable product increment.

In software engineering, agile approaches are often contrasted with plan-driven approaches. This is sometimes misinterpreted to imply that planning is not required in agile software development. Obviously, careful planning is required in most industrial software development projects involving tens, hundreds, or even thousands of persons. However, instead of a rigid, detailed pre-existing work plan enforced during the entire process, in agile development there is typically only a fairly loose up-front project plan, and the plan is allowed to evolve and become more precise during the process.

Software architecture is traditionally defined as the high-level organization of a system (e.g. [50] and [49]). However, from the viewpoint of a software development project, software architecture is a plan of a new software system, in the same sense as a blueprint is a plan of a house. This creates certain tension between software architecture and non-plan-driven agile philosophy. Indeed, the role of software architecture design was questioned by the early advocates of agile approaches, claiming that software architecture emerges without up-front design efforts. Later, the role of software architecture in agile projects has been studied by several researchers [17], [9] and [133] but there is still considerable variation among practitioners concerning the ways architecture is involved in agile development. This is particularly visible in Scrum, which itself does not give any advice specifically regarding software related activities, like software architecture work. Lack of up-front planning is one of the main concerns about adopting agile and barriers for further adoption [137].

In the following, we will limit the discussion to Scrum. The main aim for this work is to identify the ways of working that architects have taken while using Scrum. Furthermore, we want to investigate to what extent architecture work practices are in conflict with Scrum. In addition, we wanted to find out if there are correlations between these practices and the characteristics of the companies and teams running the projects. To analyze the Scrum conformance of these practices in more detail we used a set of core Scrum patterns [138] as a reference and analyzed the potential problems arising for each practice/pattern pair. Presenting Scrum as an extendable pattern language has been recently adopted as a preferred policy in the Scrum community [139].

In this study, the term architecture work refers mainly to architecture design, which is the most time-consuming and critical phase related to software architectures. Some practitioners consider architecture documentation as part of architecture design, and therefore documentation activities may be counted as architecture work in this study as well. However, architecture evaluation activities are explicitly excluded.

This work is based on an analysis of the results of a survey concerning the use of Scrum and the role of software architecture related work in Scrum, carried out in Finnish software industry in fall 2010. The target companies (11) ranged from small to large global companies, developing both embedded systems and more traditional business applications. The survey was carried out by conducting on-site interviews with project managers, product managers and software architects.

The chapter is structured according to the research setup given above. We will first briefly discuss existing work on the role of software architecture in agile development. We continue with an account of the survey and its results concerning the Scrum maturity in the target companies. The results related to the architecture work practices are discussed next, leading to the identification of four basic practices. The correlations of the Scrum maturity and the practices are also discussed in this context. Next, we review existing core Scrum patterns from the perspective of software architecture work and especially map out the potential problems arising from the conflicts between these patterns and the practices identified earlier. Finally, the chapter is concluded with a summary of the lessons learned.

7.2 Scrum

Scrum is an iterative, incremental framework for project management in agile software development. Scrum can be described in sets of three [92]. There are three roles within a Scrum team: Product Owner, Scrum Master and the development team. There are three meetings: sprint planning meeting, daily Scrum and sprint review (and retrospective) meeting. Finally, there are three artefacts: product backlog, sprint backlog, and impediment list. In the following subsections, each of these concepts is described in more detail.

7.2.1 Scrum Roles

The responsibility of *Product Owner* (PO) is to establish and communicate the product vision. PO creates the product backlog containing enabling specifications for the development team. PO orders the product backlog so that items in the top maximize the return on investment (ROI). However, the dependencies between items must be taken into account. Product owner also reviews the results of the sprint and makes release readiness decisions.

Scrum Master (SM) owns the Scrum process and makes sure that everybody follows the agreed process. SM also manages the definition of done, defining when a task can be considered as finished. The main responsibility, however, is to shield and help the team to develop the product. SM keeps track of impediments and tries to remove them.

Development team estimates, selects and develop work items. Development team commits to deliver selected work items in time-boxed development iteration (sprint). Development team should be cross-functional meaning that it has enough expertise to build a potentially shippable product increment in a sprint. In addition, development team should work in self-organized fashion, meaning that team members should select themselves work items they want to work on.

7.2.2 Scrum Meetings

The goal of *sprint planning meeting* is to make sure that the development team members understand requirements well enough, so that they can implement them. On the other hand, PO needs to get sufficient understanding on the development costs to manage the risks.

During the sprints development team gathers daily to a 15 minute *daily Scrum* meeting. The purpose of this meeting is to synchronize activities and to create a plan for the next 24 hours. In the meeting each development team member explains what (s)he has been doing since the last meeting, what will be done before the next daily scrum, and what obstacles are in the way.

Sprint review meeting is organized in order to review the results of a sprint and to get feedback on the results. Meeting lasts from two to three hours and maximum of 30 minutes is used for preparation. In the meeting the development team presents potentially shippable increment of product functionality to PO who accepts (or rejects) the result. In addition, key learnings from the sprint are discussed in the meeting.

7.2.3 Scrum Artefacts

Product backlog is an ordered list of everything that might be needed in the product. PO owns product backlog and is responsible for keeping it up-to-date. Product backlog changes constantly as new information (or requirements) become available. The items in the backlog are called product backlog items (PBI), which contain description, order and work estimate for the item. The ordering of PBIs can be done according to value (Return On Investment), risk priority and necessity. Typically, the items on top of the list are well-defined and more detailed than items which are lower on the list. Development team estimates the items, and if multiple teams are working for the same product they all take items into the sprints from the same product backlog.

Sprint backlog is a set of PBIs selected for the sprint plus a plan for delivering the product increment. Sprint backlog should contain all the work the team is going to do in a sprint. Sprint backlog is created during sprint planning meeting by development team.

Impediment list is a tool for SM. It is a list of obstacles and impediments that SM needs to remove in order to enable development team to work as efficiently as possible. Some of the impediments may require actions from the team and if this is the case these impediments can be added to sprint backlog during sprint planning meeting.

7.3 Survey and Target Companies

7.3.1 Target Group

The target group of the survey constituted of 13 teams in 11 Finnish companies. Approximately half of the companies were developing embedded real time machine control systems, while the rest of the companies were more traditional IT companies working on business and desktop applications. The survey was conducted as interviews of software professionals who had attended Scrum trainings at Tampere University of Technology. This guaranteed that the participants had roughly the same knowledge and understanding of Scrum, which the interviews and questions could rely on. A typical title of a person being interviewed was development manager, but also developers, Scrum Masters and architects participated in the survey.

7.3.2 Execution of the Survey

The survey was executed with a series of semi-structured interviews [140] of the teams in the target companies. Semi-structured interviews consist of a mixture of open ended and specific questions, which allowed us on one hand to map out data concerning facts that were known beforehand (e.g. questions related to the characteristics of the company), and on the other hand to recover unforeseen information (e.g. architecting practices and issues related to them). In addition to the questions related to architecture work, another aim of the interview was to explore the advantages and problems the teams had discovered in the use of Scrum practices in general. However, the latter part of the interviews is not covered in this paper.

The questions in the interview were divided into three different sections: background questions, questions related to the level of adopting Scrum, and questions related to architecture work. Background questions were intended to clarify the company size, experience in Scrum, the domain they are working on, etc. The Scrum related questions aimed at rating the company according to the so called Nokia test [141], profiling the company with respect to the Scrum adoption level. By using the Nokia test questions, we wanted to make sure that the company is truly using Scrum, rather than just claiming to be agile. In order to be included in the study, the company has to have adopted at least the most fundamental Scrum practices. Additionally, using the Nokia test we could gain valuable input for the analysis of the interplay of Scrum patterns and architecture work practices.

The questions related to architecture work constituted the core of the interview regarding the objectives of this paper: we asked specific questions related to the practices the teams have used with respect to software architectures, and the problems the teams have experienced in exercising these practices. The interview included questions clarifying the role an architect, the contents of that role, and the way architects work. Adopted architecture work practices were identified in the answers to these questions. For a complete list of the questions, see [142].

7.3.3 Nokia Test Results of the Target Companies

The idea of Nokia test was originally introduced in 2005, when Bas Vodde was coaching teams at Nokia Networks (now Nokia-Siemens Networks) in Finland and developed the first Nokia test focused on agile practices. He had hundreds of teams and wanted a simple way to determine if each team was doing the basics [143]. Nokia test cannot be taken as a scientific method for evaluating how agile organization is or how well they are implementing Scrum. Even so, it has some value as it gives a rough estimate on how Scrum is implemented. The results of each company are comparable to other companies, as the grading for each question is carried out by the same person and with the same criteria. However, the results are not repeatable by other persons as the judging criteria may differ.

Nokia test consists of nine questions that investigate different aspects of Scrum: sprint length, concept of done, enabling specifications, Product Owner, product backlog, work estimates, sprint burndown charts, team disruption and team self-organization. Each question is given 0-10 points, and the average value of these points describes the company's agile performance in Scrum [141]. According to Jeff Sutherland (2010), at the end of a typical Scrum Certification course, in average, participants think they can get their teams to score 6 out of 10.

Scoring of each question was carried out by two persons separately. Afterwards scoring results were discussed and differences of two or more points in scoring were discussed and resolved. In this way, misinterpretations of answers could be avoided. Summarized results of the Nokia test are illustrated in Figure 19, giving the average scores of all questions for each company.

The results show that there are 4 companies which are below 5, and 7 companies above 5. The overall Scrum adoption level shown in Figure 19 is rather typical for companies that have exploited Scrum work practices for some time, but that are not committed to full adoption of Scrum.

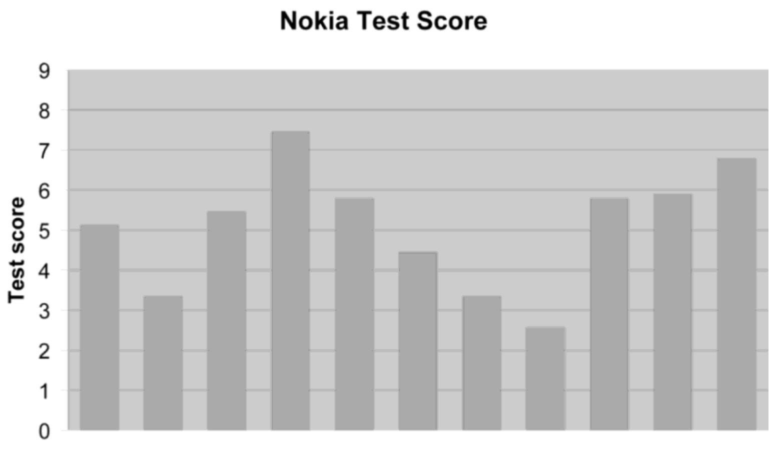


Figure 19 - Results of Nokia test conducted during the interview survey.

7.3.4 Limitations

As a method for collecting qualitative data, semi-structured interviews are widely used, but also sensitive to various sources of errors [140]. If the interview includes codifying, the results should be confirmed with the target persons to avoid misinterpretations. This was done in the context of the Nokia test. For the information regarding the architectural practices, possible misinterpretations were tried to be resolved during the interviews as much as possible, and the final report was sent to the attendees of the interviews for checking. Still, misinterpretations cannot be fully ruled out. To minimize the risk of missing or misinterpreted data we used several interviewers who were familiar both with Scrum and with software architectures.

Another threat to the validity of the results is the relatively small number of teams that were interviewed. As far as the quantitative data are concerned, the number is too small to allow statistical analysis. However, regarding the identified architectural practices in Scrum we argue that this study provides a reasonably covering picture of the ways practitioners perform architecture work in Scrum, given that the found four practices emerge consistently in the survey, and that theoretical considerations support the findings. On the other hand, more detailed observations extracted from the data may have only anecdotal evidence.

Since the companies in this survey represent different sizes and domains, the qualitative results are not expected to be biased towards certain kinds of companies. Still, the quantitative results may be affected by the relatively large number of machine manufacturers in the target companies.

7.4 Software Architecture Practices in Scrum

As stated in [17] architecture and Scrum do not go very well together, as there is no natural place for architecture work in Scrum. Yet, in the real world architecture work is carried out in one way or another. Whatever way is chosen, practitioners will probably face challenges in merging architecture work with Scrum. In this section we first analyze the topic theoretically and then briefly outline the practices for architecture work we have identified in the survey.

7.4.1 Theoretical View

In principle, software architecture work can appear in different places of the Scrum work flow. First, the architecture can be created either before the actual Scrum sprints start, or during the sprints. In the latter case the architecture can be created either within the sprints, or as a simultaneous activity outside the sprints. If the architecture is created within the sprints, there are again two options: the architecture can be either created in one or more sprints that precede the actual implementation sprints, or it can be created incrementally as needed in any sprint.

This analysis suggests that there are basically four options to place software architecture work in Scrum: before sprints, simultaneously outside sprints, in the first sprint(s), and along all the sprints. Intuitively, the last approach is closest to the Scrum philosophy, while the other approaches are less aligned with Scrum. If the architecture is designed up-front or in a separate simultaneous process, it becomes a plan which is given to the Scrum team from above, which is clearly against the self-organizing philosophy of Scrum. Furthermore, Agile Manifesto [7] suggests valuing responding to change over following a plan. If the architecture is designed in a dedicated sprint, the sprint does not produce a potentially shippable increment to the software, as required in Scrum.

In the following subsection we explore the actual practices to incorporate software architecture work in Scrum based on empirical data. The theoretical analysis above stems from that data: the analysis was carried out to provide further confidence that the four practices identified in the survey data indeed cover the major options to relate software architecture work with Scrum.

7.4.2 Observed Practices

From the interviews, four different strategies for architecture work were discovered:

1) Big up-front architecture created during the analysis phase before starting the development in sprints (BUFA).

2) Sprint zero [144] where the team created the architecture in the first planning sprint (SPR0).

3) Designing architecture when needed in normal sprints (SPR*).

4) Separate architecture process where architecture is designed outside Scrum process when necessary (SEPA).

The boundaries between different approaches are not clear cut, but we analyzed the work practices in each team and selected the approach that had the closest correspondence with the practice that was used in the team.

Figure 20 shows how many teams were using each approach. Note that the number of interviewed teams was larger than the number of companies. In the following subsections we describe each approach in more detail, based on the information obtained in the interviews.

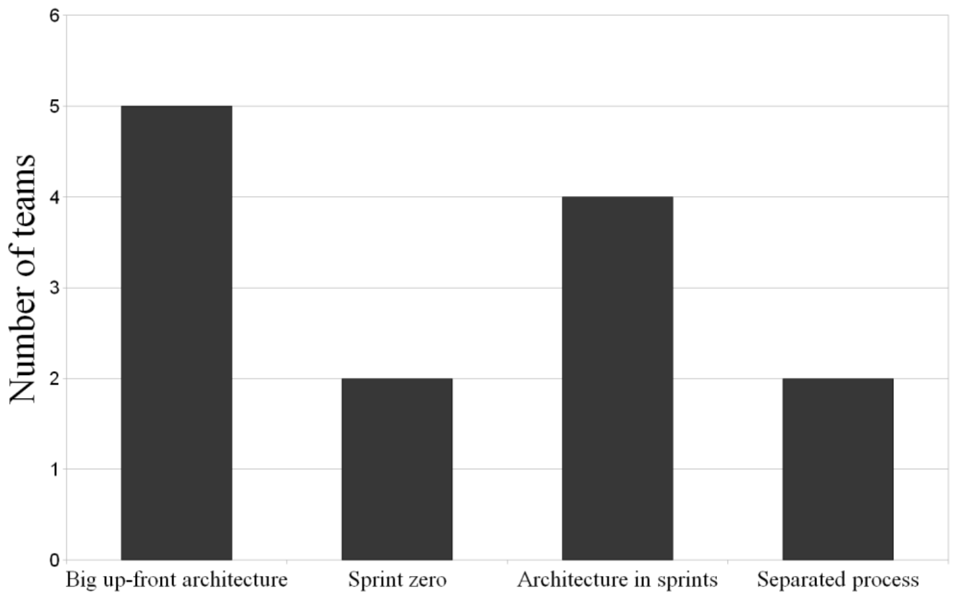


Figure 20 - Number of teams using each architecture work approach in Scrum.

7.4.3 Big Up-front Architecture (BUFA)

BUFA is typically used in companies that are building very complex real-time embedded control systems. The common factor is that the systems are huge in terms of lines of code. This kind of companies typically has a proprietary platform that is used to build specific products and working machines. However, the hardware part of the system is significant and has its specific requirements. In many cases, there is a long analysis phase where requirements are gathered and architecture is designed. Even though the platform provides the most common features, the product architecture needs serious design effort. The analysis phase might last for six months.

Once the most of the architecture is in place, the development starts. In this phase the architecture still changes when emergent requirements come in or when new information has come up in the development sprints. The platform changes may also require architectural changes. The changes are typically handled by the same architect or architecture team that designed the original architecture up-front. So the Scrum teams do not make architectural decisions at any stage.

According to interviews, this approach was taken in use as it is hard to describe features in product backlog so that they are small enough. Teams reported that typically the features are so big that it is almost impossible to implement them in one sprint, even if there were multiple teams working on them. If architecture is designed up-front; it becomes easier to divide the features into smaller items in product backlog.

Interviews also revealed that some companies did prototyping as well in the up-front design phase. They experimented with different approaches and technologies, and implemented the basic building blocks of the architecture. Especially if the architects take part in coding, this comes close to the architecture in sprints approach.

7.4.4 Sprint Zero (SPR0)

Two of the interviewed companies used SPR0. It means that they had one sprint before actual development sprints where they built product backlog and the architecture. Typically this sprint was about the same length as the other sprints, i.e. from one to four weeks. In this sprint the main designer (or several designers) worked with the development team(s) and designed the initial architecture. Many companies reported that later in the development sprints the architecture changed as the knowledge about the system increased. If this was the case, then typically the development team made the architectural changes within sprints. The most critical parts of the architecture usually persisted without changes but the rest of the design evolved during the sprints.

The main difference to BUFA is that the architecture is essentially designed by the development team itself and not by an architect who is a separate person from the team. Furthermore, the length of the sprint zero is shorter than the length of the up-front architecture design phase. When there is an up-front design phase, it might last up to six months, but sprint zero is four weeks at maximum and typically less.

7.4.5 Architecture in Sprints (SPR*)

Four teams reported that they do not have separate analysis or architecture phase: they just start doing sprints and developing the system. The goal with this approach is to produce the initial architecture during the first sprint while implementing potentially shippable features. The architecture is designed in detail only for the features that are shipped after the current sprint. For other parts, there is just a rough vision of what the architecture will look like. In every sprint, the architecture concerning potentially shippable features is refined and already implemented parts are refactored when necessary. This process goes on and the architecture is designed piece by piece during the sprints until it is finished. The approach increases the amount of required architectural refactoring as architecture is designed incrementally.

In some cases, the design was guided by one or more persons in the team that had the most experience on the domain. Other team members then built features on top of the design. One team reported that the designer is an emergent role within the team and changes from sprint to sprint. The same team also stated that anyone in the team is privileged to do architecturally significant decisions.

The common nominator in this approach is that the teams were experienced in the domain. One company reported that they have had bad experiences with this approach when the team was new to the domain. The team did not know how to proceed with the design and as there was no up-front architecture, the result was a failure. The architecture just did not correspond to requirements and it had to be refactored all the time. Finally, they had to do it again from the scratch.

7.4.6 Separate Architecture Process (SEPA)

The fourth way to carry out architecture work in Scrum was to have it as a separate process. In this approach architecture is created by a separate team that may have members from the development team(s) but is organizationally completely separated from the development team(s). Members of the architecture team might do architecture work part time, meaning that they are working also in the development team, or they can be working only for the architecture team. The architecture team has checkpoints or milestones when a certain part of the architecture must be ready. Typically there is one checkpoint for each release of the architecture team (i.e., a new version of the system's architecture that contains support for new features for the next product release). In the interviewed companies, the checkpoint cycle was three months as their release cycle was four times a year.

In the checkpoint meeting, the architecture team provides the architecture (or possibly several alternative architectures) for evaluation. Members of the development team, PO, SM, team lead, and possibly customers participate in the meeting. Once all the participants are satisfied by one of the competing proposals, they select it as a basis of the design.

It was interesting to notice that the motivation for the companies to use this approach was to make sure that "they don't mess up the Scrum process in any way". The separate architecture process is connected to the Scrum process essentially via the product backlog: the architecture process produces items to the product backlog.

7.4.7 Team Characteristics and Architecture Practices

After the interview sessions, we analyzed the ways of working and as we saw that the way of working clearly stood out from the ways we had observed earlier, we added a new approach as an observed work practice. After having four different approaches identified, it turned out that all new interviewed companies could be classified using these four categories. Since the same practices repeated themselves from one interview to another, we are confident that these are the main practices which are used to carry out software architecture work in Scrum in general, in spite of the relatively small number of interviewed teams. This conclusion is supported by the theoretical analysis as well. Interestingly, all the possible options revealed by the theoretical analysis are actually used in the companies. However, the boundaries are not always clear cut, and the practices emerge in different variations.

We compared the used architecture work approaches to team's characteristics such as agile experience, domain and project type. Table 3 shows the relationship between the used architecting practice and the experience of the team in agile methods. In the table, the numbers indicate how many teams in a particular experience category (columns) were using a particular practice (row).

Table 3 - Team's Scrum experience in years and the architecture work approach they used.

Approach used	**Less than 1 year**	**1-2 years**	**2-3 years**	**Over 3 years**
BUFA	1	2	0	2
SPR0	0	0	2	0
SPR*	0	1	3	0
SEPA	0	1	1	0

As Table 3 shows, teams that have been using Scrum for more than three years relied on the BUFA approach. Interestingly, teams that have a couple of years' experience on Scrum seem to try out different approaches without clear preferences. A possible interpretation of these numbers is that inexperienced teams start with BUFA, which is probably the easiest to understand and manage, and the most experienced teams resort back to this practice after trying out other approaches.

Next we compared the architecture work approaches to the project types the teams reported to be working on. Projects were divided into two categories: customer projects and product projects. Customer projects were identified as projects where customer has ordered something and team was creating that artefact or the team was subcontracting for a product manufacturer. Product projects are projects where the team is building a product that the company will sell to its customers.

Table 4 shows how many teams used particular architecture work practices for particular project type. As can be seen from the results, both the customer projects and product projects used all architecture work practices, with the exception that SPR0 was used only in product projects. It seems that the project type has less influence on the chosen practice. The results are not surprising as some of the interviewed teams mentioned that they used the same way of working regardless of the project.

Table 4 - Team's project type and the architecture work approach they used.

Approach used	**Product project**	**Customer project**
BUFA	2	3
SPR0	2	0
SPR*	2	2
SEPA	1	1

Finally we wanted to see if the nature of the domain of the company had any influence on the chosen practice. The interviewed companies and teams can be roughly divided into two categories: teams that work on embedded systems and teams that work on more traditional software systems. Former typically in this context are embedded control systems for work machines whereas the latter are software systems running on a web server or locally on a workstation.

Table 5 shows the results. A clear tendency can be seen in the figures: companies that are developing embedded systems use more BUFA, whereas companies developing traditional workstation systems are more willing to integrate architecture work with Scrum more tightly. A possible explanation is that in the case of embedded systems, the software architecture has to be designed in concert with the hardware architecture, and hardware (electronic, hydraulics and mechanical) designers are seldom comfortable with Scrum. Obviously, using BUFA the conflict of agile and non-agile practices can be avoided, or at least reduced. In this study we have not analyzed the impact of non-agile teams on Scrum teams in more detail, but we have identified this as an interesting topic for future research.

Table 5 - Domain the team is working on and the architecture work approach they used.

Approach used	**Embedded systems**	**Traditional software systems**
BUFA	3	2
SPR0	1	1
SPR*	1	3
SEPA	1	1

However, these figures should be taken with a grain of salt. It seems that the social value of the team is more dominant than any other factor. During the interviews it was noticed that some teams have embraced agile values and methods more deeply than others even it might not have shown in the Nokia test. This probably affects also the architecting work practice choice. The teams that appreciated agile values typically used architecture in sprints approach or let the team choose whatever approach they wanted. In companies which favored more traditional methods, the work approach was usually given to the developers from above. However, since the number of interviewed teams is relatively small, this can be regarded only as anecdotal evidence.

7.5 Scrum Patterns and Architecture Practices

In the previous section it was argued that some of the identified architecting practices are more in line with Scrum than others. To place that argumentation on a more firm basis, we have analyzed how the Scrum patterns, available at Scrumplop.org [138], resonate with the practices. Scrum patterns describe Scrum practices and experiences from practice in a pattern format. They look like design patterns but instead of creating an architectural artifact, Scrum patterns build organization structure, social value and software engineering process that uses Scrum. Scrum patterns are divided into separate pattern languages, each building a certain element of the Scrum framework. Scrum patterns are also recognized as an extension mechanism to the Scrum framework [139]. Practitioners can provide their insights and extensions to Scrum in a pattern format using this extension mechanism.

Since there are numerous Scrum patterns available at [138] we have limited our study to the most central Scrum patterns to facilitate the analysis. We selected patterns from the Scrum core pattern language and the value stream pattern language for this study, both available at Scrum Pattern Community, for this analysis. Advanced topics such as scaling Scrum and distributed Scrum were ruled out.

While Scrum patterns do not mention software architecture, or directly concern architecture work, they may still indirectly affect the way architecting work is done, and in that way resonate with the identified work practices. In the following, we discuss each pattern, and how it resonates with the four identified practices. The patterns are divided into four major groups according to the purpose of the patterns: patterns that concern the meetings, patterns that concern the artifacts, patterns that concern the Scrum roles, and finally patterns belonging to the value pattern language. The subsections follow this structure.

To give a general overview of the results of the analysis for each pattern category, we use a table in which each row corresponds to a pattern and each column corresponds to a practice. The extent to which the practice conflicts with the pattern is expressed with symbols -, 0, +, denoting ”practice potentially problematic with the pattern”, ”no resonance”, ”can be used together”, respectively. If the pattern is found problematic it means that the practices described in the Scrum pattern are conflicting with the architecture work approach, or the pattern makes it harder to use that approach (indirect conflict). No resonance means that the Scrum pattern and the architecture work approach do not share anything in common, i.e. the pattern does not suggest anything that has an effect on the architecture work while using the specific approach. If the pattern and the approach can be used together, it means that the pattern supports the way of carrying out architecture work. The argumentations why a pattern either supports or is in conflict with the architecture work approach are based on the data gathered in the interviews. The final judgment is, however, the authors' own conclusion on the topic.

7.5.1 Scrum Meeting Patterns

Scrum meeting patterns describe different kinds of meetings that are organized in Scrum. Table 6 summarizes the analysis of the patterns in this category.

Backlog grooming meeting describes a meeting where product backlog is cleaned up. This meeting is arranged approximately once a week. The whole development team usually attends these meetings. Backlog grooming meeting is optional and it is not a part of the Scrum core. It is not a problem for any of the architectural work practices, even though it is only applicable if SPR0 or SPR* is used. In two other work approaches, this pattern does not have an influence as the work is not accounted for in the product backlog.

Table 6 - Scrum meeting patterns and architecture work approaches.

Pattern	BUFA	SPR0	SPR*	SEPA
Backlog grooming meeting	0	+	+	0
Daily Scrum	-	-	+	+
Release planning meeting	+	+	+	+
Sprint planning meeting	0	+	+	0
Sprint retrospective	0	0	0	0
Sprint review	-	+	+	+

Daily Scrum describes daily Scrum meeting where developers meet and tell what they have done, what they will do and which problems they have encountered. If the architecture is created using SPR0 or SPR*, architectural issues can be discussed in this short meeting daily. If the architecture is created using SEPA, then someone from the architecture team should be present in this meeting, even though they are probably not allowed to talk as they are so-called "chickens". If BUFA is used and architecture-related issues emerge in developers' work, problems may arise. Architecture might need to be changed, but who is the person who will change it? Is it the architect or someone from the team? The architect may be already working on a new project or with the next version of the system and therefore unavailable. If the architect is not present in daily Scrum meeting, this might be a problem. The main problem here is, however, that the up-front architecture might get changed - even radically - as new information emerges. This increases the amount of wasted work. The same applies for the SPR* approach.

Release planning meeting describes a meeting where the goal and the vision of a release is established, and the initial version of product backlog is created. It affects all architecture work approaches, but does not conflict with any of them.

Sprint planning meeting describes an 8 hour meeting where the Product Owner presents the ordered product backlog to the team. The team then selects and negotiates with PO the work to be done in the sprint. This pattern can be applied to architecture work (and is not in conflict) if SPR0 or SPR* is used. The pattern is not applicable to other approaches as the work is not done in sprints.

Sprint retrospective describes a meeting after sprint review but before the next sprint planning meeting where the team discusses its development process and improves it, both to make it more effective and enjoyable. It has no direct relationship to the architecture work. However, the discussion in the meeting may concern the architecture work approach selected and if it should be changed.

Sprint review describes a meeting at the end of each sprint where the results of a sprint are reviewed and discussed. From the perspective of architecture work this pattern is in line with all approaches but BUFA. Here again the problem is that once the architecture is designed up-front there is no feedback loop. Sprint review is essentially a feedback event, and it might generate architectural tasks that have to be done. Of course, observed problems with the architecture can still be fixed, but the feedback loop becomes quite long, which is against the main idea of sprint reviews. Even if SEPA is used, those emerging tasks can be dealt with, as those tasks are input for the architecture team. However, BUFA does not support this as the architecture is already fixed. The team or the architect can make the required changes, but again this can be interpreted as extraneous work.

7.5.2 Scrum Patterns Generating Scrum Artifacts

Scrum patterns generating Scrum artifacts describe what kinds "tools" are used in Scrum process. For example, sprint or product backlog are typical tools used in the Scrum methodology. Table 7 summarizes the analysis of this pattern category.

Table 7 - Scrum artefact patterns and architecture work approaches.

Pattern	BUFA	SPR0	SPR*	SEPA
Product backlog	-	+	+	+
Release burndown	0	+	+	+
Sprint	0	+	+	0
Sprint backlog	-	+	+	-
Sprint burndown	0	+	+	0

Product backlog pattern solves the problem that everyone should agree to what needs to be done next at any given time, and that the agreement should be visible. The solution is to create a single ordered list of all deliverables, called Product Backlog Items (PBIs).

The list is ordered by delivery date. However, if the architecture is done with BUFA, the architecture is not on this single list as it is already designed when starting the sprints. This creates a problem: how is the architectural knowledge transferred to the team? Furthermore, how can the team commit to PBIs (that are taken into the sprints) as the architecture is created by someone else? Furthermore, the development team should be able to estimate PBIs, but as the architecture design is ready-made by someone else, it will hard to estimate PBIs, at least for a couple of first sprints. This problem does not exist in approaches where the architecture is created in sprints (including SPR0) as then the development team designs the architecture. If SEPA is used, the team gets the architecture given to them, but they have a chance to influence it during a checkpoint meeting. Furthermore, the architecture team can work on the same product backlog as the team, so they can collectively commit to that.

Release burndown pattern instructs to create a burndown chart for the work remaining to complete the release. This has no effect on BUFA. If the architecture is designed in sprints (SPR0 or SPR*), architecture work is also accounted for in this chart. Release burndown is especially useful in SEPA: from the release burndown the architecture team sees when the architecture for the next release has to be ready.

Sprint pattern states that the work should be time-boxed in periods called sprints. The pattern is not applicable to BUFA as the architecture is created before the sprints start. The pattern is not applicable to SEPA, either. However, it does not conflict or cause problems for any of the architecture work approaches.

Sprint backlog is a selected portion of the product backlog transformed into tasks in a sprint backlog. The pattern also instructs to let the developers choose what tasks they want to be responsible for, while making sure they are collectively capable of completing the work selected for the sprint backlog. This is not a problem from the architecture design point of view if the architecture is created in sprints (SPR0 or SPR*). In BUFA or SEPA, the developers cannot work on the architecture tasks as the architecture is given to them as a ready-made design. If SEPA is used, the developers can affect the design in checkpoints, but they cannot work on architecture tasks if they are not part of the architecture team. This reduces the benefits of the self-organization aspect of Scrum as the team is not self-organizing when it comes to tasks related to the architecture. In addition, the Scrum team has to commit to the architecture design that is out of their hands. This is against the principle that the team should be cross-functional and have all skills required to produce the system. Of course, some architectural refactoring is done in sprints (especially if the architecture is designed up-front). Usually this means that the team refactors some parts of the system and updates the architecture documentation. But the negative effect of losing cross-functionality and self-organization makes this pattern incompatible with SEPA and BUFA.

Sprint burndown pattern suggests that burndown charts should be used to provide overall status of the team in the sprint. The pattern is not applicable to BUFA and to SEPA (unless the architecture team uses burndowns to show their progress). As there is no burndown used in these approaches, it might reduce the visibility that is a central theme in Scrum. Other approaches are in line with the pattern.

7.5.3 Scrum Patterns Generating Scrum Roles

Scrum patterns generating Scrum roles describe the three Scrum roles in a pattern format. Table 8 summarizes the analysis of this pattern category.

Table 8 - Scrum role patterns and architecture work approaches.

Pattern	BUFA	SPR0	SPR*	SEPA
Product owner	+	+	+	+
Development team	-	+	+	-
Scrum master	0	+	+	0

Product Owner is responsible for the product backlog and the vision of the product. In BUFA, the architecture is created either by a separate architect or by PO (or both). Since PO should be a business oriented person, she might need help in the architecture work. SPR0 and SPR* also do not conflict with the PO role. However, it might be hard for PO to see the value of architecturally significant items in the product backlog. This might result in a situation where architectural refactoring is not getting done as PO does not see enough value in it. Consequently, this leads to increased technical debt. SEPA does not conflict with this pattern, either, especially if the architecture team works on the same product backlog as the development team(s). If the separate architecture team works on its own product backlog, it should have its own PO.

Development team pattern states that there should be a cross-functional, self-organizing team that has enough combined competencies to accomplish the project. However, if the architecture is designed up-front by the architect or as a separated process by an architecture team, then the team probably does not have enough competence to create the architecture. On the other hand, as the development team members may be working in the architecture team, it might not be a problem. If the architecture is given to the team by the architect or by the separate architecture team, it is possible that the team might not understand all the forces that have affected the design. In addition, as the architecture design is handed over from outside, it might take a lot more communication to transfer the architectural knowledge.

Scrum Master is responsible for the Scrum process. As BUFA and SEPA are outside Scrum, the pattern is not applicable to them. However, these approaches also miss the benefits of having Scrum Master removing impediments. The other two architecture work approaches are in line with the pattern.

7.5.4 Value Pattern Language

Value pattern language describes very central principles of Scrum as patterns, for example, that every sprint should produce potentially shippable increment of the product. Table 9 lists all Scrum value stream pattern language patterns.

Table 9 - Value pattern language and architecture work approaches.

Pattern	BUFA	SPR0	SPR*	SEPA
Fixed work	0	-	-	+
Regular product increment	0	-	+	0
Release plan	+	+	+	+
Visible status	-	+	+	+
Vision	-	+	+	-

All work needs to be accounted for if PO wants to use velocity of the team(s) for release planning. *Fixed work* pattern suggests to ”compartmentalize the planning work for long-running high-risk business items into periodic meetings that are outside the sprint budget”. Architecture work can be recognized as planning work for long-running high-risk business items as the architecture should address multiple quality requirements that are directly derived from the business goals. With this interpretation, *Fixed work* pattern basically suggests that architecture work should be isolated as a separate process that is not counted in the sprint work budget, thus encouraging SEPA rather than SPR0 or SPR*. The pattern does not resonate with BUFA as in that approach the amount of architecture work carried out in sprints is minimal.

Regular product increment pattern says that in every sprint the team must work together to create a regular product increment in a short time box. If every sprint should produce potentially shippable product increment, it means that SPR0 is in conflict with this pattern. In the sprint zero, development and testing environments are set up and the architecture is designed. However, this is not a shippable result as it does not provide any value to the customer. If the zero sprint produces usable features, too, the approach is not pure SPR0 anymore. SPR* does not conflict with this pattern as in that approach each sprint produces potentially shippable product increment. In other approaches this pattern is not applicable as there are no sprints involved.

Release plan also known as product roadmap describes how PO should select features for each release and basing on velocities of the team(s) calculate the release date for each release. This pattern is applicable to all four approaches to architecture work. Furthermore, it is especially suitable for a separated architecting process, as the architecture team sees the future release dates and features that the releases contain and can plan their work based on the roadmap. Without this pattern applied, it might even be impossible to use that approach. This pattern can be also used with up-front architecture for the same reasons.

Visible status pattern promotes Scrum core value, visibility, by saying that the statuses of teams should be posted on the walls in the organization. One could use sprint burn-downs, release burndowns or other approaches. In approaches where the architecture is designed in sprints, this pattern is easy to apply. Even in SEPA, visibility of the architecture team can be supported by this pattern, if the architecture work proceeds in clear increments. However, in BUFA the visibility of the architecture work has less value, and it might be hard to say reliably which percentage of the design is completed.

Vision pattern describes the vision as a description of the desired future state the team is going to create. Furthermore, unlike many corporate visions, this vision needs to be something that can be created. The product backlog items should describe how PO thinks the vision will realize. If interpreted loosely, this means that PO should have a vision of the system and its architecture and the team means basically everybody else that is involved in creating the vision - including the architect. If that is the case, this pattern is in line with all the architecture work approaches. If interpreted more strictly, the team meaning development team, this pattern is in conflict with the BUFA and SEPA.

7.5.5 Summary

To summarize the analysis on Scrum patterns and architecture work practices, it seems that the SPR* approach is compatible with all the patterns but one. Both SPR0 and SEPA have conflicts with few patterns, but both seem to be mostly in line with Scrum. On the other hand, Scrum guide [92] clearly states that each sprint should produce potentially shippable product increment. If the architecture is created in the first sprint, it might be the case that the product of that sprint is not potentially shippable. Incremental architecting required by SPR*, although well aligned with Scrum, is challenging since it requires exceptionally skillful and knowledgeable development team, both in terms of software architectures and the domain. BUFA is both most popular, and most in conflict with Scrum, with 7 Scrum pattern conflicts. Still, even this practice is aligned with Scrum patterns in 3 cases and has no resonance in 9 cases, which gives some rationale why even this practice seems to work, at least with partial Scrum.

As a conclusion, aligning architecture work and Scrum is possible but challenging. For example, it can be difficult to develop a large embedded system without big up-front design. It may be sensible for a company to apply architecting practices which are less aligned with Scrum, because the losing of some of the benefits of agile methods may be compensated by better alignment with the organization. Architecture could also be built in sprints, but then to be successful, the team must have strong domain knowledge or the project has to be small enough. SPR0 or SEPA could be a feasible choice if the company is not deeply committed to Scrum values. SEPA might work for both large and small systems, but it requires coordinated communication between the architect and the development team. Ergo, a company should choose pragmatically an approach which fits the characteristics of the project and the desired level of agility, without forgetting to continuously improve the process.

7.6 Related Work

Even though incompatibility of Scrum and software architecture is often used as an argument against Scrum, there are surprisingly few studies made on the topic. Babar [145] has studied architectural practices in agile software development, based on interviews and focus groups. The interviews were carried out in a company developing systems for financial services. This study was not specifically focusing on Scrum. Babar’s analysis was based on different architectural activities, like architectural analysis, synthesis and evaluation, rather than on overall approaches to integrate software architecture work with agile. In this sense we see our work as complementary to Babar’s work. The key challenges found in this research were incorrect prioritization of user stories, lack of time and motivation to consider design choices, and unknown domain and untried solutions. Especially the last two challenges are quite obvious if software architecture design is carried out in the most Scrum-like way, SPR*. This may be one reason why so many experienced Scrum teams had chosen BUFA in our study.

Schougaard et al. [146] have carried out field studies on software architecture and software quality in four Danish software companies. In their study they say that all architects in studied companies reported having problems in fitting their work to the Scrum framework. Schougaard et al [146] also say that it seems that there is a need for a parallel process or an addition to Scrum that takes the special characteristics of software architecture work into account. This is in line with our observation that some of the interviewed companies used a separate process for the architecture work (SEPA).

Babar et al. [147] present an industrial case study of exploiting product line architectures in agile software development. In the case study they describe software architecture process of a software company. They also present a lot of day-to-day practices such as how architecture is documented and how daily meetings are carried out. However, they also report on a practice where the company uses separate development process for the baseline architecture and new features are tried out in a research process. Both of these processes use Scrum. The research process results are integrated (if necessary) to baseline architecture which is developed in the development process. The research project approach described by Babar et al. (2009b) has some similar characteristics to what we have observed in SEPA.

Baskerville et al. [148] have conducted four studies over 10 years of time. They have observed a two-stage pattern where changes in the market cause a disruption of established engineering practices. Baskerville et al. [148] conclude that the next arising problem could be the organizational issue created by the boundaries between agile and plan-driven teams. This issue is a root cause for the problems studied in this paper, as architectural practices mainly originate from the plan-driven approaches. This issue was also observed in the cases where agile Scrum teams had to co-operate with non-agile hardware design teams.

Isham [149] describes how monolithic redesign of software architecture leads to a failed project. In this work, the architecture was then redesigned using Scrum. The success story presented by Isham [149] was accomplished by setting a vision and a roadmap in place in the beginning of the project, and changing the course as new information became available. This approach resembles the SPR* approach that we observed in our study. This experience also speaks against BUFA.

Cockburn [68] suggests starting with a simple architecture that handles the most critical cases. Then it can be developed and refactored to tackle the special cases and new requirements. However, it should not be an objective itself to get the architecture at the end of the project. This approach is similar to what we have observed in companies which use the SPR* approach.

Coplien and Bjørnvig [18] describe a lean approach to software architecture. Lean architecture means just-in-time delivery of functionality and reducing the waste. Coplien and Bjørnvig [18] also state that rework in production is waste. However, in design rework is not waste, rather it creates value as it saves more expensive work in the production phase. In order to achieve this, Coplien and Bjørnvig [18] encourages to engage all stakeholders together as early as possible. They continue that architectural decisions should be made at the responsible moment - that moment after which the foregone decision has a bearing on how subsequent decisions are made, or would have been made. Furthermore in lean architecture approach there is strong emphasis on collaboration between people during the design phase and utilization of domain knowledge. Domain knowledge is recognized as an essential part, in order to capture the end user mental model. Regarding the practices we have identified, the lean architecture concept seems to advocate the SPR* approach.

7.7 Conclusion

In this study, we have identified four approaches to integrate Scrum with software architecture work in 11 companies. Although there may be different variants of these practices, it seems reasonable to expect that they represent main Scrum practices in the industry. This belief is also grounded by an analysis of the possible (sensible) choices there are for arranging architecture work in Scrum: the identified practices cover those choices.

The practices were studied both from the viewpoint of the data revealed by the interviews, and from the viewpoint of their potential conflicts with actual Scrum practices, as concretized by Scrum patterns. The most popular practice appearing in the interviews was also found to be the most problematic in terms of the Scrum patterns. On the basis of the analysis on the interplay between Scrum patterns and architecture work practices, we conclude that BUFA (big up-front architecture) and SEPA (separate architecture process) approaches are the most conflicting approaches when using Scrum. However, in interviews it was discovered that the other two approaches are not without problems, too. SPR* (architecture in sprints) approach can lead to a failure of the project if the team does not have enough domain knowledge. In turn, SPR0 (sprint zero) approach is in conflict with one of the most fundamental concept of Scrum: delivering a potentially shippable product increment in every sprint. In general, it seems that there are strong forces which discourage practitioners to design software architecture in the ways aligned with Scrum.

We see that our study can serve practitioners in several ways. First, the study presents a covering set of practices for architecting that has been tried in industry. These practices work at least in some environments, but the study illuminates the problems associated with each of the practices. Second, the study helps to understand which practices are less problematic, given a certain set of Scrum patterns that have been adopted in the company. Third, if the company has already adopted one of the four architecting practice, and is considering increasing the level of agility by imposing new Scrum patterns, the study shows the possible management risks arising from conflicts between the new Scrum patterns and the adopted way of architecture work.

8 TOPDOCS: USING SOFTWARE ARCHITECTURE KNOWLEDGE BASE FOR GENERATING TOPICAL DOCUMENTS

This chapter is a verbatim copy of the following publication:

Veli-Pekka Eloranta, Otto Hylli, Timo Vepsäläinen and Kai Koskimies. 2012. TopDocs: Using Software Architecture Knowledge Base for Generating Topical Documents. In *Proceedings Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA) 2012*, IEEE Computer Society, pages 191-195.

8.1 Introduction

The information related to software architecture is exploited by many stakeholders including managers, architects, designers, coders, testers, and maintainers. Each stakeholder needs a particular slice of architectural information, depending on the stakeholder's viewpoint and the task at hand. Obviously, it would be highly desirable to be able to produce an information package for a stakeholder that covers exactly the task or concern at hand, rather than burdening her with the entire architecture documentation where most of the information is irrelevant for her current needs. In this paper, we call such condensed targeted information packages *topical documents*. A topical document is a specialized dynamically generated document focusing on a certain kind of need. Clearly, a major obstacle for making use of topical documents is the effort needed to produce a specific document for a casual, possibly one-shot need.

Architectural knowledge management (AKM) has been an active research area during the last few years (e.g. [128]). Recent surveys of tools and techniques for AKM ([150], [128], [78]) reveal, however, that relatively little attention has been paid to the ways the architectural information is exposed to the stakeholder, besides navigation facilities and queries. The focus has been in providing analytical, derived information (e.g. information related to traceability, change impact analysis, design maturity, etc.), rather than selecting and arranging the information for a particular stakeholder need. We propose an approach to automatically generate topical documents using the information in an architectural knowledge base (AKB). We assume that such a knowledge base has been populated during the development process with information regarding architecturally significant requirements, architectural decisions, design descriptions, and architectural evaluations. We also assume that this information is organized according to a well-defined metamodel.

There are several anticipated advantages of AKB based architecture document generation. First, producing software architecture documentation is traditionally challenging, because architectural knowledge is created in small fragments during the life-time of the system by several stakeholders. Using AKB, the pieces of information can be immediately recorded when they are created, and an up-to-date topical document can be generated when needed, by exposing the relevant information in a structured, readable form.

Second, by analyzing the AKB metamodel against the emerging needs of stakeholders, it becomes possible to write scripts for traversing the AKB and collecting all the relevant items that might be relevant for the need. In this way, the AKB metamodel facilitates the generation of specialized document types.

Third, due to the underlying knowledge base, a document can be produced not only as a traditional static document, but also as an online electronic document with links to other information in the AKB or to outside sources such as web pages.

Finally, being based on a metamodel, the approach can be easily extended for new types of needs. In our work, we needed a special safety-oriented architectural document that would highlight the safety integrity levels (SIL) [151] in a safety-critical embedded system. This was done by small extensions of the metamodel, and by writing scripts that exploited those extensions [47].

We have realized our approach on top of an Application Lifecycle Management (ALM) platform, Polarion [152], which provides built-in versioning and time-stamping support. The implemented infrastructure for topical documents, together with the scripts of existing types of topical documents and their user interfaces, is called TopDocs.

8.2 TopDocs Metamodel

The requirements for the TopDocs metamodel originated mostly from the needs of our industrial partners. We wanted to support a basic view of software architecture as components and their relationships. However, as the current view on software architecture emphasizes the rationale behind the design, rather than just the design itself [5], [153], we wanted to encourage a decision-centric way to describe the architecture: decisions complement architectural design with rationale. Furthermore, we wanted to allow freeform representation of individual design solutions with text, diagrams etc. Also, both domain and application specific knowledge [19] had to be supported. The metamodel is presented in Figure 21.

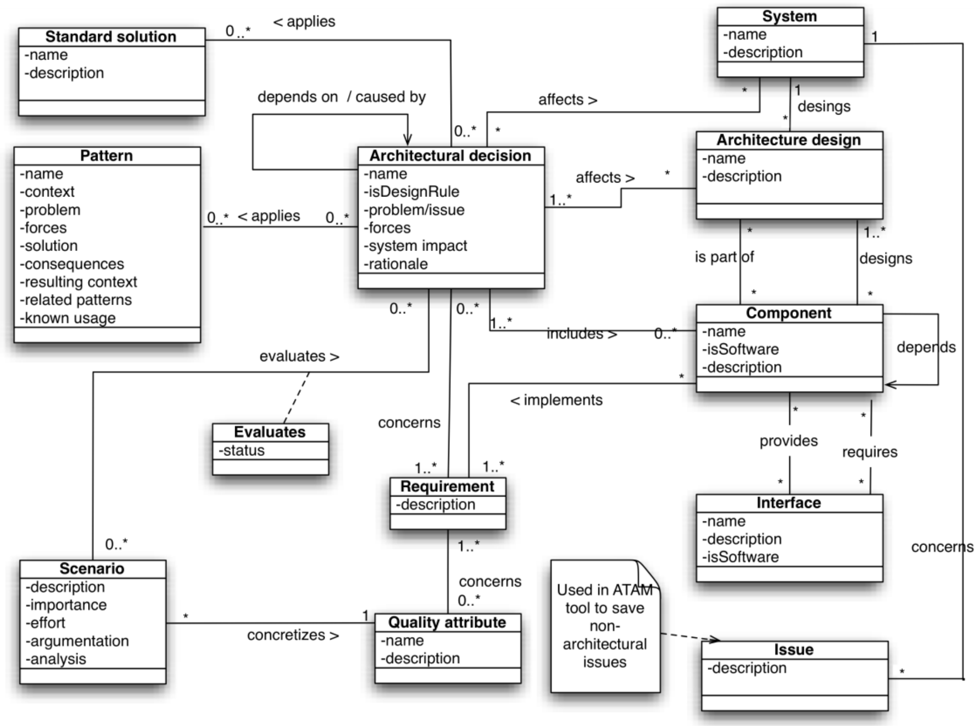


Figure 21 - Metamodel for TopDocs architecture knowledge management database.

The metamodel supports storing general architectural knowledge through two concepts: *Pattern* and *Standard solution*. Patterns follow strictly a widely used pattern format whereas a *Standard solution* can be described in free form. *Pattern* and *Standard solution* concepts are associated with *Architectural decision* concept. These relationships are “applies” relationships as an architectural decision can be the application of a pattern or standard solution [106].

Architectural decisions are presented using quite a minimalistic approach (similar to [81]) as the industrial partners indicated that lengthy documentation formats (e.g. the one presented in [80]) can be too laborious to use in practice. We added *forces* attribute to the architecture decision concept as we wanted to provide a place where different factors impacting the decision making process can be documented. An attribute for separating design rules from other decisions was added, as the nature of design rules is somewhat different from that of ordinary decisions, with more global effect on the architecture.

Architectural decisions affect *Architecture design* and may include usage of certain *Components*. Architecture design is a free-form description of the design and may include diagrams, attachments and text. The *Component* concept is used to describe an individual component of the system and may contain diagrams, attachments and text as well. A component may also provide or require an *Interface*. The *System* concept represents a general description of the system to be built. It is related to architectural decisions as some decisions might affect the whole system instead of some specific design.

The *Requirement* concept captures the requirements of the system. A decision may concern one or more requirements. Furthermore, to enable traceability, components are linked with requirements. A component may implement one or more requirements.

We wanted to include evaluation-related knowledge in the metamodel as well. We chose to support scenario-based methods such as ATAM [16] as it has been most widely adopted [60]. Thus, we added concepts *Quality attribute* [22], *Scenario,* and *Issue.* The evaluation result (risk, non-risk or trade-off) is stored in the *Evaluates* relationship concept. The *Issue* concept was added to enable recording other issues that may surface during the architecture evaluation, but are not directly related to architecture.

In addition to the relationships shown in Figure 21, the underlying ALM platform provides built-in relationships for presenting hierarchical structures (parent and child relationships) between any elements of a particular type. We exploit these for several concepts (e.g. quality attribute's refinement hierarchies and architectural design hierarchies).

8.3 Topical Documents

8.3.1 Overall Approach

A major issue in this kind of approach is the way stakeholder needs are expressed for generating an appropriate topical document. In principle, this could be done in different manners. The main purpose is to let a stakeholder express the target of her interest in such a way that the produced document matches the needs as closely as possible. A minimum requirement is that the document does not miss any architecture-related information that is relevant for the concern at hand (and that exists in the knowledge base). On the other hand, a topical document should be as small and concise as possible, to avoid burdening the stakeholder with unnecessary information.

Different consumers of architectural knowledge have been identified e.g. in [154]. In this work we will concentrate on three central stakeholders consuming architectural knowledge, namely architects, developers, and maintainers. Our general approach to generate topical documents that are useful for these stakeholders is the following. We consider the metamodel from the viewpoint of a particular stakeholder, and identify (i) which element(s) in the metamodel can serve as a key that identifies a major concern for that stakeholder, and (ii) given such key element(s), which information contents inferable from that key should be included in a topical document that satisfies the information need for that concern. Once the concern-key-contents mapping has been specified, the only remaining task is to define a suitable external representation for the contents (that is, for the topical document).

In the following, we will discuss examples of topical documents for architects, developers, and maintainers. All these topical documents can be currently generated using TopDocs. We expect that various other topical documents could be specified and implemented for these and for other stakeholders as well, using the same basic approach.

In addition to topical documents specified by key elements, it makes sense to define non-parameterized topical documents for specific purposes. For example, we have implemented a topical document which is a report of architectural (ATAM) evaluation.

Similarly, TopDocs provides a special topical document that is essentially a traditional comprehensive software architecture document.

8.3.2 Quality-centered Document for an Architect

A main concern of an architect is the satisfaction of the quality requirements of the system, as the architecture is the vehicle to realize such requirements. For example, when a so-called risk theme is exposed in ATAM [16], the architect is presumably interested to scrutinize all aspects in the architecture, which are somehow related to a compromised quality attribute.

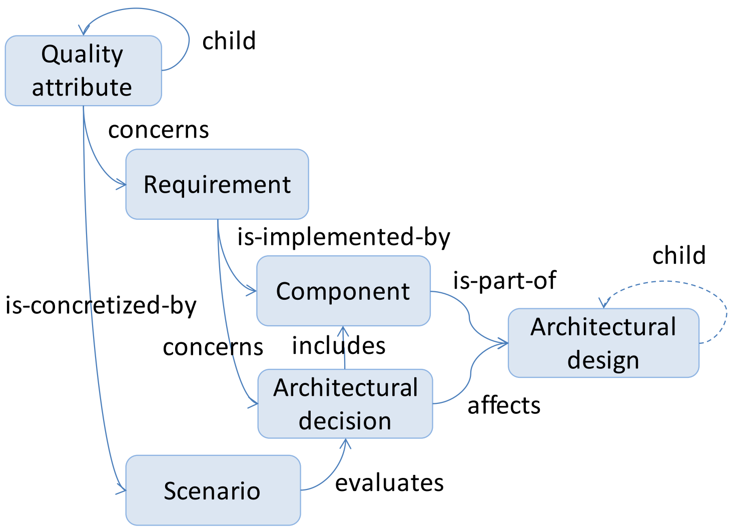


Figure 22 - Contents of quality-centered topical document.

Accordingly, we define a topical document to satisfy a quality-related concern, typically for an architect. The key element in the metamodel is obviously *Quality attribute*. The topical document should collect all information that is relevant for scrutinizing the architecture from the viewpoint of one or more quality attributes. The contents of the topical document for a quality attribute are depicted in terms of the metamodel in Figure 22.

The starting point is the chosen quality attribute and its possible child quality attributes. The topical document is built by examining the requirements and scenarios related to these quality attributes, and by further traversing the components and architectural decisions related to the requirements and scenarios. Finally, the actual architectural designs that either contain the components found, or that are affected by the decisions, are included in the document. If a detailed document is desired, the refinements of architectural designs are included as well (dashed arrow). We omit the detailed structures of the documents here.

8.3.3 Component-centered Document for a Developer

Typically, a developer uses the documentation of software architecture as a basis of the implementation of components. The documentation presumably gives some guidelines for the detailed design and for implementation of the component, as well as the usage context of the component. In particular, the documentation should present and explain the provided and required interfaces of the component.

If a developer is simply given a comprehensive, general software architecture document, finding the relevant parts of the document for the relatively specific task at hand can be challenging. Our primary aim is to generate a topical document that collects as accurately as possible the needed information for the realization of a component, so that the developer can focus on this topical document as a basis of the work. Naturally, the same topical document can be used also by other stakeholders interested in a component (or subsystem) viewpoint of the architecture.

The contents of this kind of topical document are depicted in Figure 23, again in terms of the metamodel. The key element of the metamodel is obviously Component. The topical document is built by first taking the description of that component. Next the requirements connected to this component are collected. If the component is part of a larger design, this is included in the document to give a technical context for the component. A central part of the document is also the description of the interfaces either

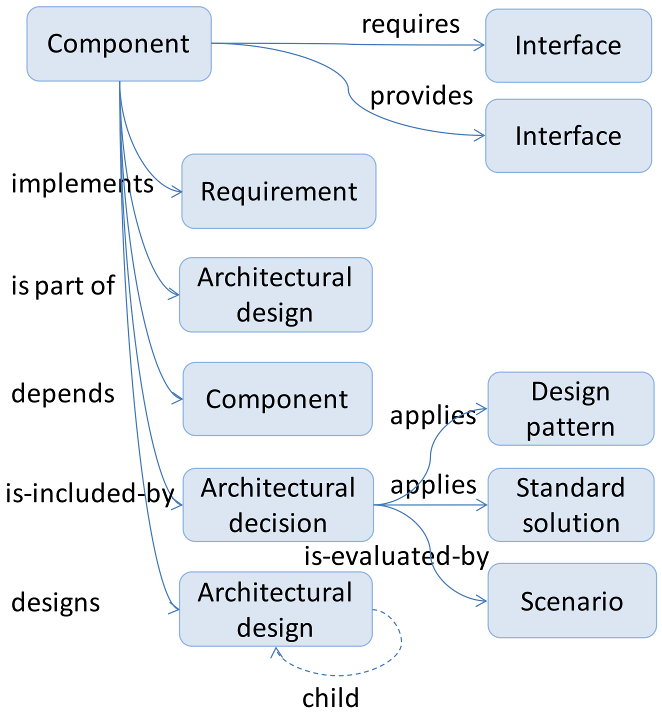


Figure 23 - Contents of component-centered topical document.

provided or required by the component. The developer should also know the other components that either depends on this component, or this component is dependent of.

The architectural decisions that deal with the chosen component are essential information constraining and guiding the developer. If the decisions apply design patterns or other standard solutions, these are included in the document to make sure that the developer understands the idea of the decided solution. We also include the possible scenarios that have been used in architectural evaluation to analyze these decisions (together with the results of the evaluation), as this may provide valuable information on the (expected) effect of the component on the quality requirements. Finally, the possible architectural design given for the component itself is included in the document, together with possible refinements for a detailed version.

8.3.4 Scenario-centered Document for a Maintainer

The third topical document has a more narrow scope. Basically, we want to support a maintainer in a task that is actually foreseen by the stakeholders. Our approach is based on the observation that in fact scenario-based architectural evaluation reveals the maintenance tasks that are anticipated during the evolution of the system. In ATAM, the architectural analysis relies on scenarios which depict concrete situations in which certain quality attribute may be compromised. If the quality attribute is related to system evolution (say modifiability, maintainability, portability, traceability), the scenario typically describes a task that is expected to become necessary in the future. Thus, it makes sense to provide a topical document that is based on the scenario and its relationships in the AKB.

The key element of the metamodel for this kind of topical document is *Scenario*. The contents of the topical document are depicted in Figure 24. The topical document is built starting from the chosen scenario. The scenario itself, together with its analysis gives the architect's explanation on the technical execution of the task depicted in the scenario. For example, if the scenario describes an extension of the system, the scenario analysis traverses all the places in the system that needs modification, and evaluates the difficulty of the modifications. The document also includes all the architectural decisions that are involved in the analysis, that is, decisions that are in some way related to the task. If these decisions in turn concern certain requirements, these are included as well to give an understanding of the requirements that may be affected by the scenario. Finally, the actual architectural designs related to the decisions are included, possibly augmented with their refining designs.

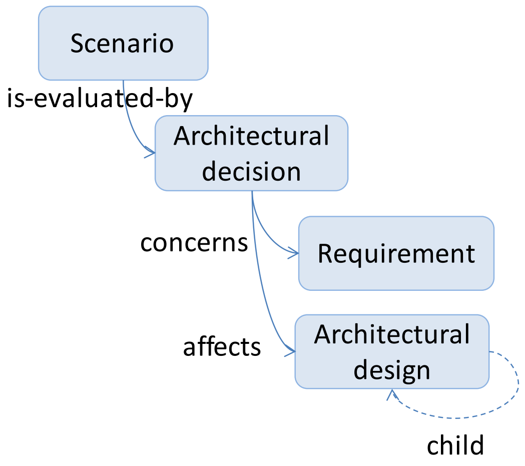


Figure 24 - Contents of scenario-centered topical document.

8.4 Case Study

We carried out a case study to explore the extent the generated documents are useful for stakeholders with different backgrounds. In particular, we wanted to find out how well a topical document succeeds to capture the relevant information for the kind of task it is intended for. The target system in the case study was IDLE [155], a learning environment where students can enrol for the courses, do exercises, and return project works.

The case study was executed in four steps. In step 1, we fed existing documented architectural knowledge to the AKB. This was carried out manually by identifying designs, components, etc. from an existing document of the system and presenting this information as work items in the AKB. In step 2, we carried out ATAM evaluation for the system using an integrated ATAM bookkeeping tool which automatically stores the architectural information emerging during ATAM sessions in the AKB. Roughly half of the items in the AKB were created in this step. After the ATAM evaluation, decisions' rationales were completed, when necessary. In step 3, we carried out an experiment where the product manager of IDLE was asked to provide three potential evolutionary scenarios that have architectural implications, and formulate related maintenance tasks each concerning a certain quality attribute relevant to the system. The product manager had no knowledge of the principles of TopDocs or the contents of the AKB. We generated a topical document for each of these tasks using the quality-centered topical document type (III B). Finally, in step 4 we analyzed the results.

Four test persons, not related to our research, were selected among the department staff to act as the stakeholders responsible of the tasks specified in the scenarios. These four persons represented different experience levels: experienced architect, architect, experienced developer and developer. The persons were given one scenario and the corresponding topical document at a time, and asked to read the document and answer the questions, if possible. In addition, the persons were asked to mark which parts of the document they found useful in respect to the task at hand. The persons used about 15 minutes in average to handle a scenario.

Summarizing the results, the sizes of the topical documents were 7-32% of the full architecture document, indicating that filtering the AKB with a particular concern significantly reduces the size of the topical document. The proportions of useful items in the three documents were 20% (security concern), 33% (consistency), and 60% (performance) in average.

8.5 Related Work

TopDocs bases on dividing architectural knowledge to small chunks, work items. A similar approach has been proposed by Su et al. in [156] in order to capture architecture document exploration paths. On the other hand, TopDocs can be viewed as a general AKM tool approach. Liang et al. [128] present a covering set of possible use cases for AKM tools. These use cases are divided in four separate groups: (i) consuming architectural knowledge, (ii) producing architectural knowledge, (iii) knowledge management, and (iv) intelligent support [128]. In this grouping, TopDocs focuses on (i) and (ii).

Farenhorst et al. [150] describe the desired properties of architectural knowledge sharing tools, and compare seven existing tools in terms of these properties. According to [150], none of the compared tools support showing stakeholder-specific content in a single view. In our approach, stakeholder-specific documents can be produced for several stakeholder groups. Codification of architectural knowledge is generally supported in AKM tools; in addition to normal adding, editing and reusing, our approach also includes tools for the codification of architectural knowledge as a side effect of other activities (architecture evaluation). As far as the expressiveness of the metamodel is concerned, our metamodel covers the four categories identified by Tang et al. [78]: context knowledge, general knowledge, reasoning knowledge and design knowledge.

8.6 Conclusions

The added value of TopDocs comes from the built-in knowledge of the relevance of the different data items in the AKB from the viewpoint of a particular type of concern, and from the organization of these data items as a readable document. We argue that the TopDocs approach could significantly help organizations by exposing codified architectural knowledge in an attractive, focused form without manual documentation effort.

9 LIGHTWEIGHT ARCHITECTURE KNOWLEDGE MANAGEMENT FOR AGILE SOFTWARE DEVELOPMENT

This chapter is a verbatim copy of the following publication:

Veli-Pekka Eloranta and Kai Koskimies. 2014. Chapter 8 – Lightweight Architecture Knowledge Management for Agile Software Development. In *Agile Software Architecture*. Editors: Muhammad Ali Babar, Alan W. Brown and Ivan Mistrik. Morgan Kaufmann, Boston, 2014. ISBN 978-012407720, pages 189-213.

9.1 Introduction

The tension between agile software development and software architecture (e.g. [17], [9] and [133]) is reflected and even emphasized in architecture knowledge management. We define architecture knowledge management (AKM) as the methods to support sharing, distributing, creating, capturing, and understanding of a company's knowledge on software architecture (see [52] and [157]).

Agile Manifesto [7] downplays the significance of activities aiming at comprehensive knowledge codification, including architectural knowledge. Still, it is generally agreed (e.g. [158] and [159]) that systematic codification of architectural knowledge is required for many kinds of systems, for example for systems that are used and maintained for decades, or have legislative viewpoints. If agile approaches are used in the context of such systems, AKM must be integrated with agile development.

On the other hand, we strongly believe that AKM is not necessarily a burden for agile development in general, but rather a practice that can support agile in the long run by improving communication between stakeholders especially in large scale, possibly distributed agile projects, and to maintain the systems developed. In particular, we argue that by integrating a carefully tuned, lightweight AKM as part of the process, the agile development paradigm can largely avoid the maintenance problems originating from scarce documentation.

To fit with agile approaches, AKM itself must be lightweight. The central property of lightweight AKM is minimal additional cost in terms of human work: neither producing nor consuming of architectural knowledge should introduce significant activities that are related to AKM itself, rather than to the direct support of product development. We aim at AKM that is next to invisible for the stakeholders, thus affecting the agile characteristics of the work as little as possible. In this chapter, we propose possible ways to make both producing and consuming architectural knowledge lightweight, while preserving the main benefits of AKM.

From the viewpoint of architectural knowledge codification, an attractive approach would be to populate the architectural information repository (AIR) as a side-effect of some activity that creates new architectural information or makes architectural knowledge explicit. This can be accomplished by integrating the tools used in those activities with the AIR. Examples of such tools are requirement engineering tools (creating architecturally significant requirements), architecture modeling tools and IDEs, architectural evaluation tools, and reverse engineering tools. In particular, we advocate here the usage of lightweight architectural evaluations for codifying architectural knowledge: during architectural evaluations, significant amount of architectural knowledge emerges and becomes explicit, like architectural decisions, high-level designs, architecturally significant requirements and business drivers etc. If a bookkeeping tool used in the evaluations is integrated with the AIR, a large body of valuable architectural knowledge can be codified virtually effortless. In particular, tacit architectural knowledge [70] is not captured in documents during the development process, but it does typically emerge during the discussions in architecture evaluation. Thus architecture evaluation is a natural context to make tacit architectural knowledge explicit.

From the viewpoint of consuming architectural knowledge in an effortless way, the main problem is that the information in the AIR is typically structured according to a conceptual metamodel rather than according to the needs of the stakeholders ([160], [161], [162]). Consequently, without more high-level need-oriented support, the stakeholders have to express their needs using general query and navigation mechanisms, which can be tedious in many cases. In this chapter we propose to augment the AIR with a need-oriented interface which is able to generate information packages that satisfy particular anticipated needs of stakeholders. In this way, the stakeholders do not have to use time and energy for screening out the relevant information from a mass of architectural knowledge. An earlier description of this kind of tool support is given in [163].

There are many ways to carry out software development in an agile way. We will explore the realization of the above proposals in the context of Scrum [90], which is by far the most popular agile approach today applied in industrial software development projects [133]: according to a recent survey [164] Scrum and Scrum hybrids have 69% market share of agile methods in software industry. Scrum combines agile principles [7] with lean manufacturing philosophy [89]. Essentially, Scrum defines how the work is decomposed into tasks, how the tasks are ordered, managed, and carried out, which are the roles of the project members, and how persons in these roles interact during the development process. A central concept of Scrum is a sprint, a working period during which the team produces a potentially shippable product increment.

On the other hand, there are also many ways to do architecture related work in Scrum. In [165], we investigated software architecture work practices used with Scrum in software industry. While some of these practices are more in line with Scrum than others, they are all motivated by reasons that emerge in real life, and they all have their merits in the companies involved. The main contribution of this chapter is a proposal for integrating existing architecture evaluation and documentation techniques with these practices to establish lightweight AKM in Scrum. Since the proposal is based on observations of agile projects in industry concerning the software architecting practices, we expect that the proposal is feasible in practice. In addition, to understand in more detail the flow of architectural information in real life agile projects, we carried out a study in which we interviewed practitioners in industry to identify the artifacts carrying architectural information and the ways they are produced and consumed. The approach draws on the experiences of the first author on software architecture documentation challenges in agile projects, to be summarized in the next section.

After the next section, we will continue as follows. In Section 9.3 we briefly introduce techniques that serve as constituent parts of our proposal for lightweight AKM for agile software development, architectural evaluation methods and automated document generation. In Section 9.4 we give a short summary of Scrum and discuss the findings of an earlier study on architectural practices in Scrum in industry. In Section 9.5 we discuss the results of an interview carried out in industry to identify the architectural information flow in real life agile projects. In Section 9.6 we propose models to integrate Scrum with lightweight AKM, based on the observed architectural practices and architectural information flow in Scrum. Finally, we conclude with a formulation of lightweight AKM principles for agile software development as we see them after this work, and with some remarks on future work.

9.2 Challenges of Agile Architecture Documentation

Agile Manifesto [7] guides to value working software over comprehensive documentation, but it also explicitly states that documentation can be valuable as well. Unfortunately, Agile Manifesto is often misread as a permission to overlook documentation. In small, short projects documentation may be less important. However, in large projects the needs for communication exceed the limits of face-to-face communication both spatially and temporally. When hundreds or even thousands of developers and other stakeholders are involved in the project, some documentation practices must usually be established to distribute the information efficiently throughout the organization. Similarly, if the life-span of a software system is several decades, documentation is needed to bridge the communication gap between several generations of architects and developers. On the other hand, there may be even compelling legislation reasons that dictate documentation. In particular, safety critical systems must often pass certifications that are largely based on reviewing documents.

It should be emphasized that documentation should not replace face-to-face communication, when the latter is more appropriate. According to Coplien and Bjørnvig (2010), all documentation (including architectural) is written for two reasons: to remember things and to communicate them. However, documentation is one-directional communication and as stated by Cockburn in [68], it is not a very efficient way of communicating between two persons. The most efficient way of communicating is, two persons talking face-to-face at the whiteboard. The main motivation to produce architectural documentation is to record the design and its rationale as a kind of collective memory.

To understand better the problems of architectural documentation in agile industrial project contexts, the first author participated actively in an industrial project for about 18 months, with the responsibility of developing the software architecture documentation practices in the project. The company was a global manufacturer of work machines with embedded control systems. From this experience, we recognized three major challenges related to software architecture documentation:

- *Size*. One of the main concerns was that the amount of required documentation is rather large. The architecture document soon became unreadable and non-maintainable as its size grew. To solve the problem, the document was split into smaller chunks. However, this led to the problem that it was hard to find the right document. Furthermore, the splitting of the document resulted in synchronization problems: how to ensure that the different documents constitute a consistent and complete description of the architecture.
- *Fragmentation*. In general, fragmentation of architectural information was found to be a major problem. A lot of existing architectural information was fragmented in different places: presentation slides, e-mails, meeting memos, etc. The reason for this was that people had to make notes in the meetings, give presentations, send e-mails, etc., but updating the documents was not on any critical path towards the completion of backlog items. This resulted in an unpleasant situation: architectural information existed but nobody was sure where that information was, or whether it was the most current.
- *Separated architecture documentation*. Architectural information was produced generally all the time during the development, but architecture documentation took place only at certain major milestones. This caused a delay in the recording of architectural information, and consequently led to loss of information. Architectural information was mostly produced in the beginning of the project in the form of rough designs and decisions, but also during Scrum sprints more detailed design information was produced and more architectural and design decisions were made. To make the codifying of architectural knowledge more efficient and precise, the codifying should take place immediately when the knowledge is available.

The conclusions drawn from the experience were the following:

- The architectural information should be exposed in small information packages that address specific needs of stakeholders, rather than in conventional all-purpose, comprehensive architecture documents. The consistency of such information packages should be guaranteed by the underlying infrastructure, rather than relying on manual updating.

- The storing of architectural information should be centralized in a common architectural repository in order to be sure where the current version of the information resides.
- The codifying of architectural knowledge should be seamlessly integrated with the agile process, so that architectural knowledge is largely codified as a side-effect of those activities that create the knowledge, without notable additional effort.

These conclusions can be viewed as (fairly ambitious) high-level requirements for lightweight AKM for agile software development. In this chapter, we propose approaches which constitute a partial solution for these requirements, but also leave many issues open. In particular, the integration of the AKM practices with an agile process is a challenging problem for which we offer a specific solution approach, namely the exploiting of (lightweight) architectural evaluation as a means to codify architectural knowledge without extra effort.

9.3 Supporting Techniques for AKM in Agile Software Development

In this section we discuss existing techniques that can be used to support lightweight AKM in agile software development, and that we will exploit in the sequel. First, we discuss the role of architecture evaluation from the viewpoint of AKM and briefly introduce two architecture evaluation methods, ATAM [16] and DCAR [166]. The former is the most widely used architecture evaluation method, while the latter is a new method especially aiming at lightweight and incremental evaluation. Both can be used to produce core architectural information in a repository as a side-effect of the evaluation. We use ATAM as an example of a holistic evaluation method; several other architecture evaluation methods such as SAAM [167], PBAR [132], or CBAM [64] could replace ATAM in the discussion of this chapter. As the second topic, we outline existing techniques to expose the architectural information in the form of focused information packages and to populate the AIR [163].

9.3.1 Architecture Evaluation Methods, Agility, and AKM

Software architecture is typically one of the first descriptions of the system to be built. It forms a basis for the design and dictates whether the most important qualities and functionalities of the system can be achieved or not. Architecture evaluation is a systematic method to expose problems and risks in the architectural design, preferably before the system is implemented.

ATAM (Architecture Trade-off Analysis Method) is a well-known scenario-based architecture evaluation method used in industry [16]. The basic idea of a scenario-based architecture evaluation method is to refine quality attributes into concrete scenarios phrased by the stakeholders (developers, architects, managers, marketing, testing etc.). In this way, the stakeholders can present their concerns related to the quality requirements. The scenarios are prioritized according to their importance and expected difficulty, and highly prioritized scenarios are eventually used in the architectural analysis. The analysis is preceded by presentations of the business drivers and of the software architecture.

Architectural evaluations not only reveal risks in the system design, but also bring up a lot of central information about software architecture. The authors have carried out approximately 20 full-scale scenario-based evaluations in the industry, and in most cases the industrial participants have expressed their need for uncovering architectural knowledge as a major motivation for the evaluation. A typical feedback comment has been that a significant benefit of the evaluation was communication about software architecture between different stakeholders, which otherwise would not have taken place. Thus, software architecture evaluation has an important facet related to AKM, not often recognized.

Essential architectural information emerging in ATAM evaluations include architecturally significant requirements (and scenarios refining them), architectural decisions, relationships between requirements and decisions, analysis and rationale for the decisions, and identified risks of the architecture. Since the issues discussed in the evaluation are based on probable and important scenarios from the viewpoint of several kinds of stakeholders, it is reasonable to argue that the information emerging in ATAM (and other evaluation methods) is actually the most relevant information about the software architecture. Furthermore, this information is likely to be actually used later on and therefore important to document.

From the viewpoint of agile development, the main drawback of ATAM (and scenario-based architecture evaluation methods in general) is heavy-weightiness: scenario-based methods are considered to be rather complicated and expensive to use ([168], [23], [61]). A medium sized ATAM evaluation can take up to 40 person-days covering the work of different stakeholders ([22, p. 41]. In our experience, even getting all the required stakeholders in the same room for two or three days is next to impossible in an agile context. Furthermore, a lot of time in the evaluation is spent on refining quality requirements into scenarios and on discussing the requirements and even the form of the scenarios. Most of the scenarios are actually not used (as they don’t get sufficient votes in the prioritization), implying notable waste in the lean sense [89]. On the other hand, although communication about requirements is beneficial, it is often time-consuming as it comes back to the question of the system’s purpose. In an agile context, the question about building the right product is a central concern that is taken care of by fast development cycles and incremental development, allowing the customers to participate actively in the process. Thus, a more lightweight evaluation method that concentrates on the soundness of the current architectural decisions rather than on the requirement analysis would serve better an agile project setup.

Techniques to boost architecture evaluation using domain knowledge have been proposed by several authors. So called general scenarios [119] can be utilized in ATAM evaluation to express patterns of scenarios that tend to reoccur in different systems in the same domain. Furthermore, if multiple ATAM evaluations are carried out in the same domain, the domain model can be utilized to find the concrete scenarios for the system [169]. In this way, some of the scenarios can be found offline before the evaluation sessions. This will speed up the elicitation process in the evaluation sessions and create significant cost savings as less time is required for the scenario elicitation. However, even with these improvements, our experience and a recent survey [60] suggest that scenario-based architecture evaluation methods are not widely used in the industry as they are too heavy-weight, especially for agile projects.

Another problem of integrating ATAM with agile processes is that it is designed for one-off evaluation rather than for continuous evaluation that would be required in agile. ATAM is based on a holistic view of the system, starting with top-level quality requirements which are refined into concrete scenarios, and architectural approaches are analyzed only against these scenarios. This works well in a one-off evaluation, but poses problems in an agile context where the architecture is developed incrementally. The unit of architectural development is an architectural decision, and an agile evaluation method should be incremental with respect to this unit, in the sense that the evaluation can be carried out by considering a subset of the decisions at a time.

The lack of incrementality in ATAM is reflected in the difficulty to decide the proper time for architectural evaluation in an agile project. If the architecture is designed upfront as in traditional waterfall development, the proper moment for evaluation is naturally when the design is mostly done. However, when using agile methods such as Scrum, the architecture is often created in sprints. Performing an ATAM-like evaluation in every sprint that creates architecture would be too time-consuming. On the other hand, if the evaluation is carried out as a post-Scrum activity, the system is already implemented and the changes become costly.

A recent survey shows that architects seldom revisit the decisions made [168]. This might be due to the fact that the implementation is ready and changing the architectural decisions would be costly. Therefore, it would be advisable to do evaluation of the decisions right after they are made. This approach can be extended to agile practices. If the architecture and architectural decisions are made in sprints, it would be advisable to revisit and review these decisions immediately after the sprint.

Partly motivated by this reasoning, a new software architecture evaluation method called DCAR (Decision-Centric Architecture Review method) has been proposed in [166]. DCAR uses architectural decisions as the basic concept in the architecture evaluation. Another central concept in DCAR is a *decision force*, that is, any fact or viewpoint that has pushed the decision to a certain direction [131]. Forces can be requirements, existing decisions, e.g. technology choices, previous experiences, political or economic considerations etc. A force is a basic unit of the rationale of a decision; essentially, a decision is made to balance the forces. In DCAR, a set of architectural decisions is analyzed by identifying the forces that have affected the decisions, and by examining whether the decision is still justified in the presence of the current forces. Besides the evaluation team, only architects and chief developers are assumed to participate in the evaluation.

DCAR is expected to be more suitable for agile projects than scenario-based methods because of its light-weightiness: a typical system-wide DCAR evaluation session can be carried out in half a day, with roughly 15-20 person hours of project resources [166]. Thus, performing a DCAR evaluation during a sprint is quite feasible, especially when developing systems with long lifespan or with special emphasis on risks (e.g. safety critical systems). On the other hand, since DCAR is structured according to decisions rather than scenarios, it can be carried out incrementally by considering a certain subset of decisions at a time. If the forces of decisions do not change between successive DCAR evaluations, the conclusions drawn in previous evaluation sessions are not invalidated by later evaluations. Thus, in Scrum decisions can be evaluated right after the sprint they are made in. If the number of decisions is relatively small (say, less than 10 decisions), such a partial evaluation requires less than two hours and can be done as part of the sprint retrospect in Scrum.

If new information emerges in the sprints, the old decisions can be revisited and evaluated again. DCAR makes this possible by documenting the relationships between decisions. If a decision needs to be changed in a sprint, it is easy to see which earlier decisions might be affected as well and (re-)evaluate them. Additionally, as the decisions drivers, i.e. forces, are documented in each decision, it is rather easy to see if the emergent information is going to affect the decision and if the decision should be re-evaluated.

From the AKM viewpoint, a particularly beneficial feature of DCAR is that the decisions are documented as part of the DCAR process, during the evaluation. This decision documentation can be reused in software architecture documentation as such. If a tool is used in DCAR to keep track of the evaluation and to record the decision documentation, this information can be immediately stored in the AIR without extra effort.

9.3.2 Advanced Techniques for Managing Architectural Repositories

Even for a medium-sized system, the AIR can grow large, containing several hundreds or even thousands of information elements. This can easily lead to an information bloat problem: a stakeholder that has specific information needs faces a mass of information with complicated structure, not directly addressing the needs of the stakeholder. Typical AKM tools (e.g. [78]) provide search and navigation mechanisms to find the relevant information, but their utilization requires understanding of the AIR structure and experience in its usage. Introducing such a tool concept in an agile development process is difficult, as the agile philosophy explicitly tries to avoid heavy tools.

In [163], this problem is addressed by providing a need-oriented high-level interface for consuming the information in the AIR, essentially hiding the AIR from the consumer. This could be seen analogical to an informaticist that has knowledge about the kinds of specialized documents that are probably needed, and that is able to produce the desired type of document on the basis of the contents of the AIR. If such a document is not sufficient, the stakeholder may use the conventional search and navigation mechanisms of the AIR, but since new document types can be introduced when needed, it is expected that eventually the predefined document types cover most of the needs of stakeholders. The tool, called TopDocs, has been implemented on top of Polarion [152], a commercial lifecycle management system.

Currently TopDocs supports the generation of specialized architectural documents on the basis of a certain quality aspect, a certain part of the system, or a certain maintenance scenario. For example, a developer can ask for a component-oriented architectural document, exposing everything she needs to know about the component when developing it, e.g. interfaces, dependencies, internal design, architectural contexts, related decisions etc. In addition, the tool can generate a conventional comprehensive software architecture document or an architecture evaluation document. The tool can also generate documents on different granularity levels, allowing for example the generation of very high-level architectural documents for managers. TopDocs differs from other repository based AKM tools such as EAGLE [150] or ArchiMind [160] in that it has the capability to generate specialized, focused architectural documents for a particular need, required in our approach.

These documents correspond closely to the view concept in ISO/IEC 42010 [49], defining a view as a work product expressing the architecture of a system from the perspective of specific system concerns. Architecture viewpoint [49], in turn, establishes the conventions for the construction, interpretation and use of architecture views to frame specific system concerns. Thus, a viewpoint is realized in TopDocs by defining and implementing a new document type for a new concern. In principle, any AKM tool that provides the capability to define new viewpoints and generate views according to the viewpoints could be used in our approach.

It is important to provide automated support for populating the AIR as well. Centralized AIR offers a common place where all architectural information can be recorded immediately when it emerges. For example, instead of writing separate memos in meetings and hoping that this information eventually finds its way to the architecture document, a stakeholder can directly write the memo into the AIR where its information immediately contributes to the reports produced from the AIR contents. Similarly, a book-keeping tool for ATAM and other evaluation methods should be integrated with the AIR, so that the tool automatically populates the AIR immediately when the architecturally relevant information emerges. TopDocs provides such a tool interface for ATAM and DCAR evaluations, making it possible to store all architectural information appearing in an evaluation session directly to the AIR without extra effort. Assuming that this kind of information captures the relevant information about the software architecture that is probably used in the future, automated evaluation-based feeding of the AIR offers an attractive solution to the problem of populating the architecture information repository.

From the agile viewpoint, hiding the complexity of the AIR both from the producing side and from the consuming side is of crucial importance. Ideally, the stakeholders should not be aware of the underlying AKM infrastructure, but just use tools which are targeted to the activities related to the actual development work. Under the surface, these tools should then populate the AIR with the new architectural information. TopDocs is a step in that direction demonstrating that especially architectural evaluation tools, which are integrated with the AIR can greatly assist in the feeding of the AIR. In a truly lightweight AKM, the architectural information is collected automatically, and the information needs of stakeholders are satisfied with automated generation of targeted information packages.

9.4 Architecture Practices in Agile Projects

In this section we lay groundwork for the later discussion on how to align lightweight AKM with Scrum. We start by briefly describing the main elements of the Scrum framework. In section 1.4.2 we will summarize the results of a recent survey on how organizations build architectures while using Scrum.

9.4.1 Scrum Framework

Scrum is an iterative and incremental framework for agile project management [90], [91]. Figure 25 illustrates the structure of the Scrum process. The process is divided into three central parts: analysis phase (sometimes called pregame), development phase and review phase.

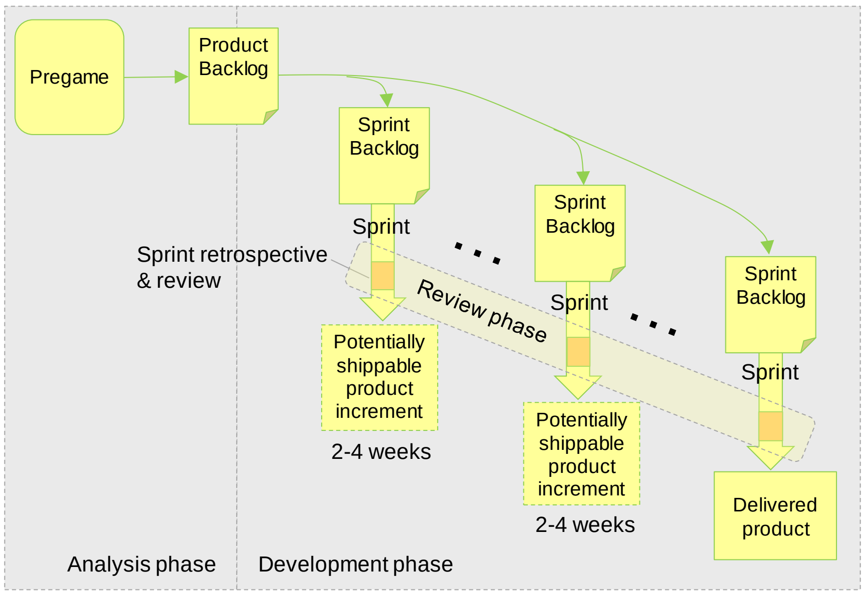


Figure 25 - Scrum framework.

In the analysis phase, the requirements are collected and analyzed. This analysis leads to product backlog which is a collection of all features and properties that the software to be built may have. For example, an item in product backlog could say “Rework the component so that it has better scalability”. Product backlog is ordered so that the most valuable items are on top of the list while taking dependencies into account. Items in the top of the list will get implemented in the next sprint.

The original Scrum paper [90] states, that the high-level architecture design is part of the pre-game. However, descriptions about software architecture and Scrum are left open in [90]. Many later descriptions of the Scrum framework do not describe the analysis phase in detail at all.

After the analysis phase, the actual implementation of the system starts. Implementation is time-boxed into time periods of 2 to 4 weeks which are called sprints. In the beginning of the sprint development teams take items from the top of the product backlog and split them into tasks which they commit to finish within a sprint.

After the sprint there are review sessions where the customer accepts the results of the sprint. It is also assessed if the goals set for the sprint were met (all tasks finished). At this stage, the software should be tested and ready to be shipped to the customer. In the review phase, the team also has a lessons-learned session where they discuss how they could improve their work practices.

9.4.2 Architecting while Using Scrum

Software architecture can be seen as a result of a set of complex system of architectural design decisions depending on each other [24], [170]. These decisions are typically made during an iterative and incremental process. The process of architecture design has been divided into three common recurring activities by Hofmeister et al. [171]: architectural analysis, synthesis and evaluation. *Architectural analysis* consists of identifying and analyzing concerns and business contexts in order to produce architecturally significant requirements (ASR). *Architectural synthesis* is an activity where solutions satisfying ASRs are found. Finally *architectural evaluation* ensures that the architectural decisions are solid. This is done by evaluating candidate solutions from the synthesis phase against ASRs.

Aligning these activities with an agile software development process has raised a lot of concerns [17], [9], [133] - it has even been said that agile and architecture is a clash of two cultures [9]. Architecture has traditionally been perceived as a plan on how the system will be built. However, in agile methods change should be embraced and there is no separate planning phase. Still, organizations have architects and they do architecture design, also in agile context. The authors have conducted an interview study in the industry to find out how organizations align architecture work and agile development (in this case Scrum) [165]. The results revealed that there are four main software architecture practices that organizations use: big up-front design, sprint zero, in sprints and separate architecture team.

Big-up-front-design practice constitutes of analysis, synthesis and evaluation of the architecture before the system is implemented in the sprints. During the implementation phase only small changes are made to the architecture design. In a sense, this is close to the waterfall approach; however, the implementation phase is carried out in sprints by the development team. The architecture is typically designed by dedicated architects and not by the development team itself. The original Scrum paper [90] actually describes that the architecture design should be done up-front. Later on, descriptions of Scrum have omitted this up-front analysis and design (see e.g. [92]).

In *sprint-zero* practice the architecture design is carried out in the first sprint, sprint zero. This sprint is from 2 to 4 weeks in length whereas big-up-front-design phase might last for 6 months. The difference of sprint zero to other sprints is that a potentially shippable product increment is not created in the sprint zero. The sprint is dedicated for design and setting up the development environment. In other words, analysis and synthesis are done in the sprint zero. Sometimes also evaluation is carried out within the sprint or right after it. The main differences to big-up-front-design approach are that the length of the design phase is radically shorter and the design is carried out by the development team itself.

In-sprints practice builds the architecture within the sprints. The architecture work is carried out by the development team. Architecture is designed and refactored within the sprints whenever the need arises. This requires very skillful development team with good domain knowledge. As stated in [165], without experienced team with good domain knowledge this approach is doomed to fail. In this approach, analysis is partially made by the product owner outside sprints and partially by team within the sprints. Architectural synthesis takes place in the sprints. This approach does not offer a natural place for architectural evaluation as the architecture is evolving all the time. Of course, at some point the architecture design stabilizes and the sprints focus on finishing the functionality of the system. At this point, it might be possible to carry out architectural evaluation, especially in a lightweight form. However, changes to the architecture at this stage of the development can already be expensive to make.

In *separate-architecture-team* approach there is a separate team that designs the architecture. The members of this team might be from different development teams. Additionally, there might be a named architect who acts as a part of this team. Typically a separate architecture team gathers up whenever necessary and analyses the next release ASRs and designs the architecture (synthesis). The actual implementation of the system is then carried out by development teams. Architectural evaluation can be carried out when the architecture team releases a new version of the design.

9.5 Architectural Information Flow in Industry

In this section we present the results of an interview survey concerning AK flow in the industry. The first section briefly describes the interview setup. In Section 9.5.2 the main results of this survey are presented. Additional general remarks of interviewees are presented and discussed in Section 9.5.3 Section 9.5.4 finally describes the limitations of this survey.

9.5.1 Interview Setup

In order to study the actual flow of architectural information in industry using Scrum, we interviewed three teams (represented by six persons) in two companies, which are global manufacturers of large machines and automation systems, exploiting Scrum in their projects. The goal was to find out what kind of architectural knowledge is produced, why and for whom it is produced, who uses it and at what stages of the project the information is produced and consumed. In this way, we expected that we can explore architectural information flow in a realistic industrial context, and ensure that our proposal for lightweight AKM is consistent with the actual architectural information flow in the company. Although the number of interviewed teams is small, this study was expected to provide at least a rough picture of architectural information flow followed in real life Scrum work.

The interview questions were the following:

- What is meant by software architecture in your company?
- How is software architecting carried out while using Scrum?

These questions revealed the architectural approach used by the teams, assumed to fall into one of the four types discussed in the previous section. The next question,

- What kind of architecture information is utilized in your company?

resulted in a list of artifacts (or types of architectural information) such as architecturally significant requirements (ASR), designs, design patterns, component descriptions, etc. For each of these artifacts mentioned by the interviewee, the following set of questions was presented:

- Who is the producer of this architectural information?
- In which stage of the project is this kind of architectural information produced?
- Why is this information produced?
- Who uses this information?
- In which stage of the project is this architectural information utilized?
- How does the producer communicate this information to the consumer?
- How often does this communication take place?

Interviewers utilized the architectural knowledge metamodel presented in [163] to make sure that interviewees did not forget some relevant type of architectural information. The interviewers did not mention any specific stakeholder groups, but just collected the stakeholders the interviewees brought up.

9.5.2 Results

The three interviewed teams used each a different approach to carry out architecture work: big-up-front-design, sprint-zero and in-sprints approach. The separate-architecture-team approach was not used by these teams. The results are presented in Figure 26 and Figure 27 . The figures are structured according to the main three phases of Scrum: Analysis phase, Development phase and Review phase [90]. Figure 26 summarizes the architectural information flow in teams using big-up-front-design and sprint-zero approaches: these approaches coincide from the viewpoint of architectural information flow, viewing the first sprint in the sprint-zero approach as part of the analysis phase. Figure 27 shows the architectural information flow in the in-sprints approach in the same way. Arrows in the figures show what information a stakeholder produces (exiting arrow) and (entering arrow) consumes in a certain phase. Arrows to both directions mean that the architectural information is both produced and consumed by the participating stakeholders.

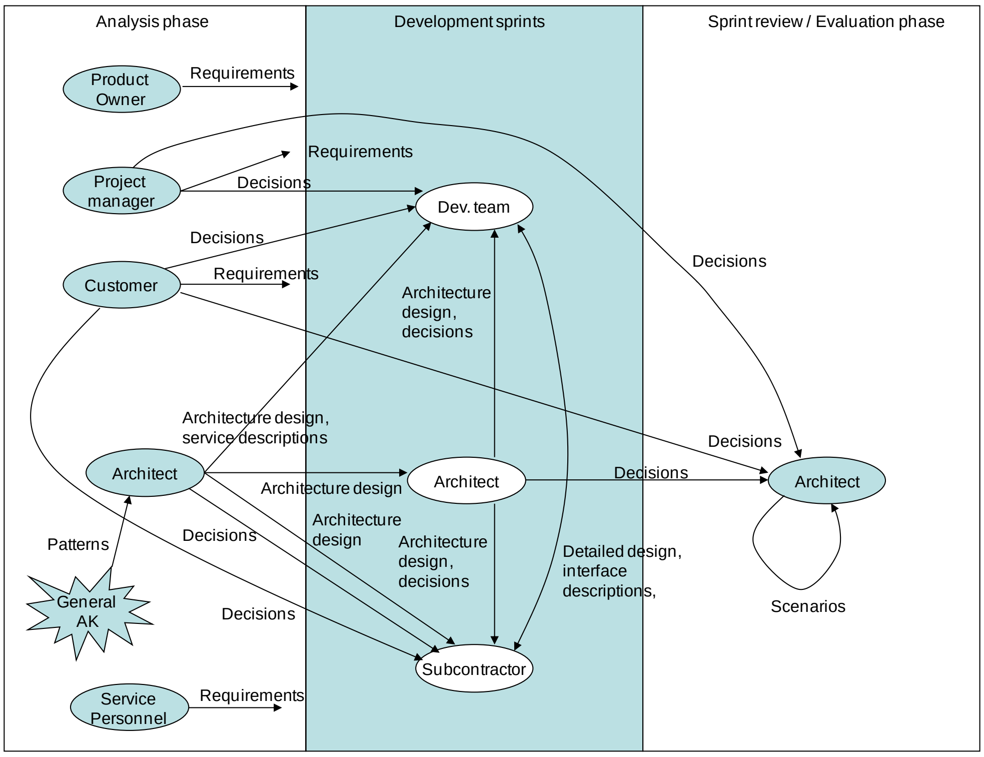


Figure 26 - Architectural information flow in big-up-front-design and sprint zero approaches

The figures exhibit fairly expected information flow. The differences in the information flow between big-up-front-design/sprint-zero and in-sprints are clearly visible: in the former, architectural decisions are produced in the analysis phase by several stakeholders and mostly consumed in sprints, while in the in-sprints approach only an initial architecture design is produced by the architect in the analysis phase, and most of the architectural decisions and designs are produced during the sprints. A noteworthy observation is that ASRs are produced in the analysis phase, but not consumed explicitly in the development phase, as one could expect (the arrows going nowhere). An explanation could be that the ASRs are reflected in the architectural decisions and designs, and consumed in this form during the development. Another possible explanation is that while using Scrum the requirements are reflected in product backlog items (in user stories, in use cases, etc.), so no one is explicitly using requirements and the stakeholders have a feeling that they are using product backlog items rather than requirements.

Patterns were used by the architect in the analysis phase (big up-front) and by the development team in the development phase (in sprints). The interviewees also reported that they have their own in-house patterns but they are rarely documented. Sometimes patterns were discussed during the review to find out which pattern really worked and

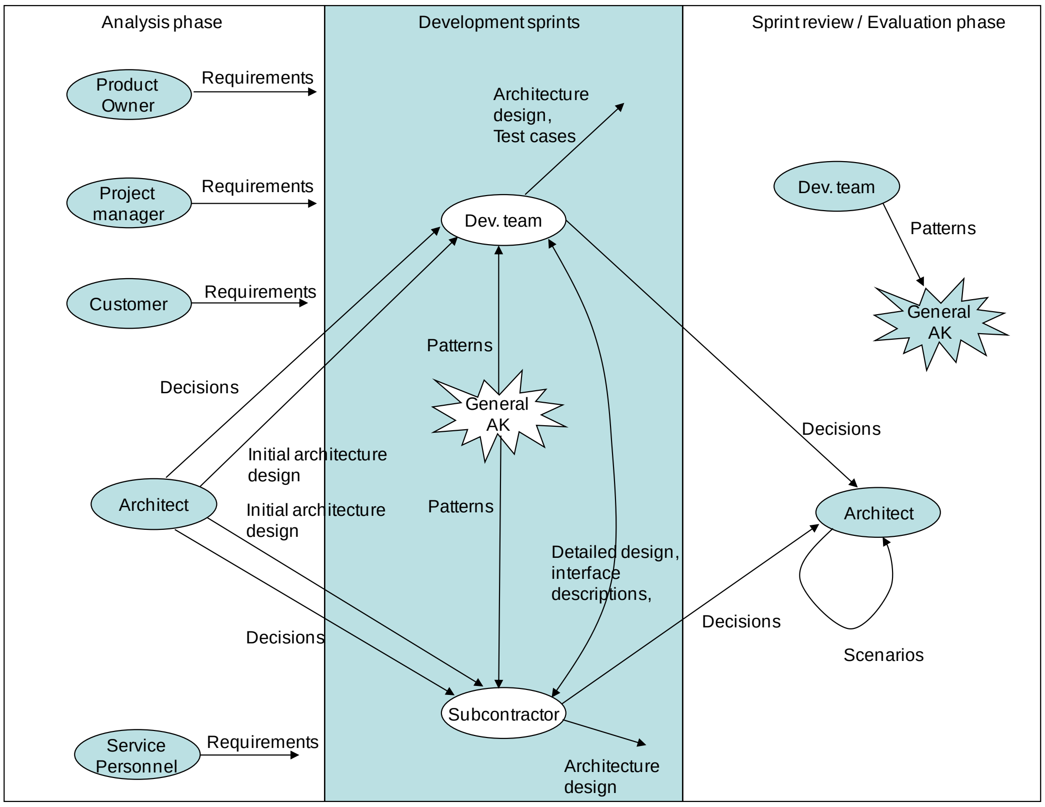


Figure 27 - Architectural information flow in in-sprints approach.

which ones caused problems. So, the team was refining the general architectural knowledge in this phase. However, this information was not written down, either. In all approaches, architects mentioned that they use decisions and scenarios in the architecture evaluations, and exploit them also in the sprint review phase. Decisions are typically discussed in the sprint review during the lessons learned sessions. One observation was that architectural designs are not used in sprint reviews as such, but the discussion focuses on the decisions. Scenarios were produced and consumed by the architect in the architecture evaluation.

Even in big-up-front-design/sprint-zero approaches, interfaces were not specified in the analysis phase but only in the development phase. In the interviews an architect mentioned that it is not reasonable to specify interfaces up-front, but instead one should just describe what services the interface should provide, as the exact interface design will always change during the implementation.

In the in-sprints approach architecture designs were produced by subcontractors and development teams, but not consumed by any stakeholder. The explanation is that the products the interviewed teams were working on were not released yet. The architecture design was produced and documented for later use, i.e. for the maintenance phase of the product. However, normally this information would probably be used by the teams themselves.

9.5.3 General Comments from Interviewees

The interviewees confirmed the difficulty to align architecture and agile development. In Scrum, for instance, there is no explicit architect role and still many organizations do up-front design and have architects. This creates the need to communicate the design to the development team, which is hard. Architectural design cannot be completely communicated using traditional architecture documents, because it is, in many cases, lacking *why* something is designed as it is. For example, the implementation might be in line with the design and functions properly when the system has order of 100 parameters. But when the number of used parameters grows significantly during the lifecycle of the product, the same implementation does not scale anymore. So it is crucially important to provide the development team with a rationale for the design and explicitly mention the main concerns in the design. E.g. in the aforementioned example, it is important to have efficient parameter handling, but it is even more important that the implementation is scalable and the efficiency is not lost when the amount of parameters increases.

In general, if the design is carried out without simultaneous implementation, there is a high possibility that the design needs to be changed. New information emerges during the implementation, making the design invalid. Often a customer may also change his/her mind, and the current design cannot handle those changes.

In interviews, the architects also mentioned that there is a lot of architectural knowledge that should be communicated. In many cases, the development teams are experienced with the domain and with similar systems, so all the knowledge does not need to be communicated from the architect to the team. However, if there is a lot of subcontracting involved or the development teams change once in a while, the architectural knowledge needs to be communicated. In this process, architectural documentation can help but not completely replace face-to-face communication. Furthermore, if the development team is implementing just one component of the large system, they may lose the touch to the architectural context and may not be able to take into account all information that concerns the component.

The interviewees reported that the communication of architectural information is most prevalent in the beginning of the sprints, e.g. in sprint planning sessions. The interviewed teams had used architecture evaluation methods like ATAM to evaluate the architecture. The experience was that evaluations are beneficial when the project size is large. If the project is small, the workload caused by the evaluation was considered to be too heavy.

9.5.4 Limitations

The number of interviewed teams and persons was small, limiting the reliability and generalizability of the results. On the other hand, an interview allowed the clarification of the work practices and more detailed answers than, for example, a questionnaire-based study would allow. Further, one of the architectural practices in Scrum, separate-architecture-team, was not addressed at all in the interview. Still, since the interviewed teams had already years of experience in using Scrum for software development projects, we believe that the results give a representative snapshot of the architectural information flow in industry, to be used as a guideline for proposing lightweight AKM practices.

9.6 Architecture Knowledge Management in Scrum

In this section we present a model to produce and consume architectural information in Scrum, exploiting the existing techniques for software architecture evaluation (ATAM, DCAR), architectural information repositories, and architectural document generation, as reviewed in Section 9.3. We discuss the architectural practices presented in Section 9.4 separately; however, we discuss separate-architecture-team case only briefly as we don’t have empirical data about the architectural information flow in that case. We also merge big-up-front-design and sprint-zero approaches, since they are similar from the AKM viewpoint. We make use of the types of architectural information identified in the study of Section 9.5, showing the Scrum phases when a particular type of information can be fed to the AIR, and when a particular information package should be generated from the AIR to support the stakeholders.

9.6.1 Big-up-front-architecture and Sprint-zero Architecting Approaches

In the big-up-front-architecture approach, most of the architectural decisions are made and a comprehensive architecture design is produced in the analysis phase (or, in the case of sprint-zero, in the first sprint), before the development sprints. Still, it is realistic to assume that during the development sprints architecture has to be modified, and some new decisions are made by the Scrum team or by the architect. The AKM process for the big-up-front-architecture (and sprint-zero) case is depicted in Figure 28. For the sprint-zero case, the only essential difference is that the analysis phase extends to the first sprint, so that most of the architectural designs and decisions are produced in that sprint.

Since a major concern in the big-up-front-architecture and sprint-zero is to ensure that the architecture satisfies all the essential quality requirements coming from different viewpoints, a natural evaluation method for this case is a scenario-based one, e.g. ATAM [16], possibly in a lightened form. With a suitable scribe tool (e.g. [163]), the information emerging in ATAM can be immediately stored in the AIR without additional effort. Since ATAM is based on the idea of discussing the most relevant architectural aspects of the system from the viewpoint of various stakeholders, it is reasonable to assume that the architectural information emerging in ATAM actually captures the most useful portion of architectural knowledge. On the other hand, an example of information type not appearing in ATAM is service descriptions, reported in the interview in Section 9.5. Service descriptions have to be manually inserted into the AIR.

For evaluating the decisions made during the development sprints, ATAM or similar evaluation methods would be inappropriate, but instead DCAR can be used for lightweight evaluation of the individual decisions possibly made during a sprint, for example in the context of the retrospective. A suitable DCAR book-keeping tool, built on top of AIR, allows the completing of the AIR contents with the new architectural information emerging in DCAR. DCAR does not directly support the documentation of design, and introducing heavy design tools would not be appropriate in Scrum. However, light-

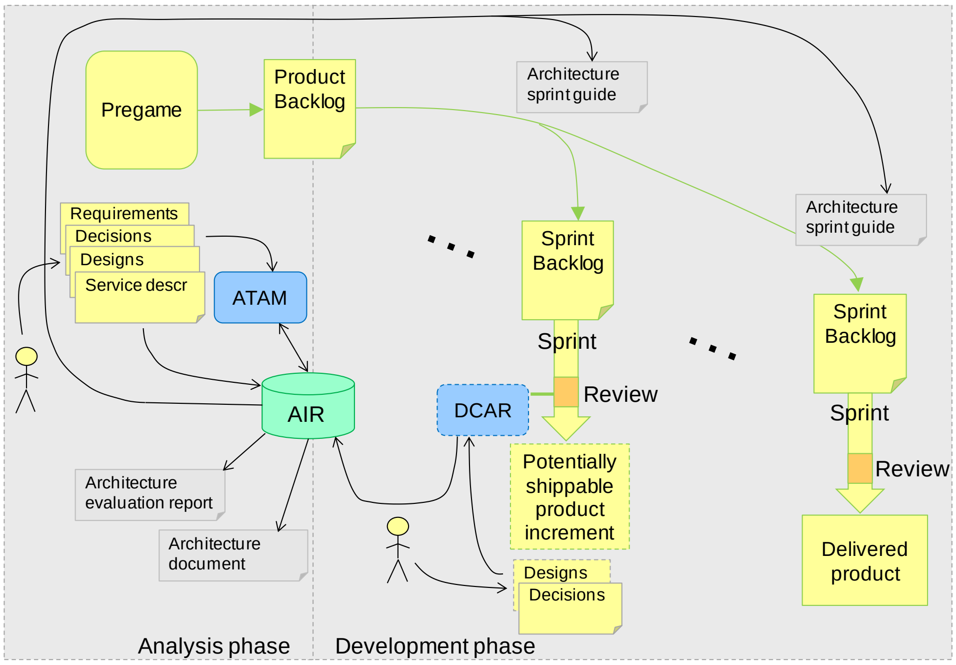


Figure 28 - AKM for big-up-front architecture (sprint zero) case

weight practices of capturing design information during team meetings (e.g. taking photos from white-board drawings) and attaching this information to the decisions in DCAR would make it possible to add this information to the AIR without notable extra effort.

As discussed in [163], it is possible to exploit the AIR to generate a (possibly on-line) software architecture document, exposing the current architectural design in a conventional form to the stakeholders. This will presumably be used in the composing of the product backlog, and also used as a guideline throughout the project, always reflecting up to-date architectural information. In addition, the information related to the ATAM evaluation can be extracted from AIR and shown in the form of an evaluation report, possible affecting some of the decisions.

The usage of the AIR enables the generation of focused architecture guides to be consumed during a sprint, collecting architectural information that is supposed to be relevant for the backlog items under work in the sprint. Assuming that backlog items (or requirements) are linked to decisions and designs, the required architectural information ("Architecture sprint guide") can be filtered and compiled automatically from AIR, in the style of the customized documents in [163].

9.6.2 In-sprints Architecting Approach

If the architecture work follows the in-sprints architecting practice, architectural decisions are made primarily during the development sprints, as part of the elaboration of the sprint backlog items. In this case there will presumably be more architecture related sprints in the beginning of the project. Regardless of the character of the sprint, all architectural decisions made during a sprint can be evaluated with DCAR in the retrospective phase of the sprint. If a DCAR scribe tool has been built on top of the AIR, the decisions can be again recorded in the AIR as a side effect of DCAR, possibly augment-

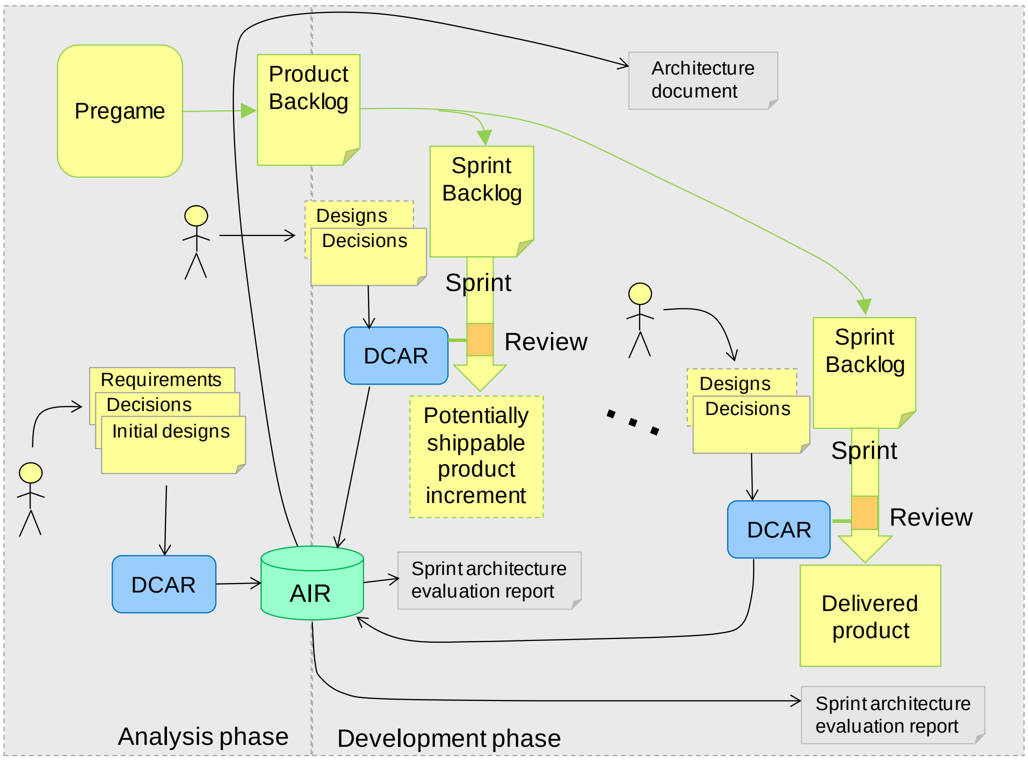


Figure 29 - AKM for in-sprints architecting case.

ed with design drawing photos from whiteboards. An evaluation report ("Sprint architecture evaluation report") can be generated from the AIR, to be reviewed in the sprint review meetings and possibly affecting the product backlog and the next sprint backlog. At any time, the contents of the AIR can be used to generate a comprehensive architecture document, for example to be used in the communication with customers or offshore teams. The AKM flow in the in-sprints architecting case is depicted in Figure 29.

As observed in the interview in Section 9.5, some initial design and basic architectural decisions can be produced already in the analysis phase. A natural way to both validate these decisions and store them to the AIR would be again to use DCAR, in principle in the same way as during the sprints. Note that architecturally significant requirements are associated with decisions as forces in DCAR, and will be stored in the AIR along with the decisions.

9.6.3 Separated-architecture-team Architecting Approach

In the case of separated-architecture-team, architectural decisions are made outside of the Scrum loop by an architecture team, consisting of members of the Scrum teams or external architects (or both). In this approach, the AIR can serve as an architectural communication interface between the architecture team and the Scrum team. Architectural decisions are recorded by the architecture team, and these decisions can be evaluated by DCAR in suitable intervals, generating evaluation reports that may affect product backlog and sprint backlog. The evaluation may also lead to reconsideration of some of the decisions.

In the separated-architecture-team case, a major issue is the communication between the architecture team and the Scrum team. For each sprint, the Scrum team should get the architectural information relevant for the sprint in a condensed form. Similarly to the big-up-front-architecture case, this can be accomplished by generating a customized sprint architecture guide containing the information in the AIR related to the backlog items under work in the sprint.

9.7 Related Work

Even though the clash between agile software development and software architecture (e.g. [17], [9], [133]) has been recognized by multiple authors, there is surprisingly little written about aligning software architecture process with agile development processes, and practically nothing about adjusting AKM for agile software development, to the best of our knowledge. For example, in a recent edited book on architecture knowledge management [52], the agile viewpoint is completely missing. From agilists there are few proposals on how to scale up Scrum and agility, e.g. [172]. However, typically these proposals focus only on big up-front architecture design, neglecting other types of practices.

Babar [83] describes software architecture process as a set of activities producing the architecture: architecture analysis, synthesis, evaluation, implementation and architectural maintenance. Basically, the same set of activities is presented by Tang et al. (2010). While this model applies as such for agile architecting as well, the faster cycles of the development essentially change the nature of architecture work. If the big-up-front-design approach is used, architecture design can be much like in traditional waterfall development [14]. However, if architecture design is carried out in sprints, architecturally significant requirements need to be analyzed, design has to be finished and documented and the architecture needs to be evaluated within a sprint. Using traditional tools and techniques this might be next to impossible to carry out.

Two strategies how organizations can manage their (architectural) knowledge have been identified: codification and personalization [158], [74]. In addition, a third approach combining these two has been proposed [173]. In the codification strategy, as much information as possible is made explicit, for example, by writing it down to the documents. In the personalization strategy, the majority of the information is tacit knowledge in persons' minds. There might be also some advice concerning who is the right person to ask from, when the need for the knowledge emerges.

Although agile approaches in many cases overlook the importance of documentation, we believe that the key to successful AKM while using Scrum is to find the sweet spot between codification and personalization. Agile approaches tend to favor personalization, but in large projects it might not work. Especially if the development is distributed or there are subcontractors involved, the software process becomes highly knowledge-intensive [173] and requires efficient ways of communication. Codification is too laborious and time-consuming in many cases, especially without proper tool.

We suggested the exploiting of an existing technique to generate targeted information packages to satisfy the information needs of agile stakeholders [163]. Somewhat similar ontology-based documentation approach has been proposed by de Graaf et al. [160].

General requirements for AKM infrastructure have been proposed by Liang et al. [128] in the form of a covering set of possible use cases for AKM tools, many of them applying to agile context as well. Furthermore, Farenhorst et al. [150] present seven desired properties of architectural knowledge sharing tools. Some of these properties are particularly important in agile context, like easy manipulation of content and sticky in nature.

9.8 Conclusions

We have demonstrated that by exploiting state-of-the-art techniques related to software architecture evaluation and automated document generation, Scrum can be augmented with lightweight AKM with reasonable cost. The most significant additional activity is architectural evaluation. When architecture evaluation is required in sprints, a decision-based lightweight evaluation technique can be used. If a simple book-keeping tool is used to assist the evaluation, the architectural information can be stored in a repository virtually without extra effort.

The proposed AKM approach, despite of its light-weightness, may not be suitable for all projects. For small projects it might still be too laborious and the benefits of the approach do not necessarily outweigh the amount of work that has to be put into codifying the architecture knowledge, unless there is definite need for covering technical documentation. However, for large and mid-sized projects in which the information sharing needs exceed the limits of personal communication, a systematic AKM approach aligned with Scrum is justified, even though there are no explicit requirements to produce technical documentation.

Research on AKM has produced a lot of advanced tools for codifying and presenting architectural knowledge. However, in our view, many of these approaches have been developed with the mindset that the software development process is expected to be adjusted for these tools, rather than the other way around. We are afraid that this may be a critical hindrance to the adoption of AKM in the industry in large scale. To be successful, an AKM approach should be possible to be introduced in a project with close to zero cost, and with clearly visible and significant benefits.

To make things more explicit, we postulate the essence of lightweight AKM for agile software development as the following manifesto.

1. *The producing and consuming of architectural information in AKM should not require extra effort.* There should be no activities that are related only to AKM itself during the software system lifecycle.
2. *AKM should be invisible from the viewpoint of information producers and consumers*. The producers and consumers should not need to be aware of AKM. AKM should be seamlessly integrated with usual knowledge-sharing activities like documentation, reviews and project meetings.
3. *Only potentially useful architectural information should be stored in AKM*. All stored information is expected to be used in the context of a probable development or evolution scenario. The burden of useless information surpasses the possible benefit of coincidental usage.

In this chapter, we have taken steps in the direction of the above principles, although many open questions still remain. In particular, although the proposed integrated collection of lightweight AKM practices are founded on findings in industrial studies, the actual application of the combination of AKM practices in Scrum calls for extensive empirical research that falls beyond the scope of this chapter.

PART V - CLOSURE

This final part will give a summary of contributions and discusses the limitations of this study. Future research directions are described and finally the part is concluded with final remarks.

10 CONCLUSIONS

10.1 Research Questions Revisited

In this section the research questions from Chapter 2 are revisited and the main results are briefly summarized.

A.1 How to carry out software architecture work in the context of agile software development methods?

> Eleven software companies were studied and four different ways to cope with software architecture in Scrum was identified. These four ways are: 1) Big Up-front architecture, 2) Sprint zero, 3) Architecture in sprints, 4) Separated process. To understand the relations of these architecture work practices to Scrum, all four ways of working were analyzed against Scrum patterns. The analysis of relations between Scrum patterns and the identified architecture practices helps to understand how software architecture work is and can be aligned with real life Scrum. All four identified practices are conflicting with Scrum practices at least to some extent. It highly depends on the project context which of the four approaches should be used. While the big up-front architecture and the separated process approach were identified as the most conflicting ways of carrying out architecture work, they were also widely used in the companies.

A.2 How could architecture knowledge repository help to provide up-to-date information for stakeholders and could architecture documentation be replaced with documents generated from the data in the architecture knowledge repository?

> The results show that TopDocs architecture knowledge repository could be used to store all kinds of granules of information produced in the software project. If these granules conform to a certain meta-model, the information can be organized automatically as software architecture documents that can be viewed from the screen or printed out. As documents are generated, they can be tailored for a certain stakeholder need. In addition, as the information in the documents originates from the repository it is always up-to-date as long as the repository is used as only means to store architecture information. The results indicate that developers and architects find these automatically generated documents useful when they have to answer questions concerning architecture.

A.3 How can general and domain-specific architecture knowledge be collected and presented in a reusable form?

> As a result a method for collecting domain-specific patterns was developed. The method utilizes architecture evaluation sessions as the main source of information. While carrying out multiple architecture evaluations on the same domain, the domain-specific architecture knowledge can be captured and refined to design patterns. The method was applied to machine control system domain and resulted in the collection of 80 domain-specific patterns. An example set of those patterns was presented as a part of this thesis.

A. How can agile software development and architecture knowledge management be aligned without losing the benefits of both worlds?

> The results suggest that there is no single approach how to align agile software development and architecture knowledge management. The way of working has to be selected basing on the context of the project. The results, however, provide models for aligning these two activities. The most suitable model can be applied depending on the context of the project.

B.1 How can software architecture evaluation methods be made more efficient to support agile development methods?

> The results show that currently used software architecture evaluation methods such as ATAM can be made more lightweight by utilizing domain knowledge. This allows software architecture evaluations to be carried out in shorter time. In this way, evaluation methods can be integrated in Scrum as the threshold to carry out multiple evaluations during the project is lowered.

B.2 How to make architecture decisions first class entities in software architecture evaluation to support iterative and incremental evaluation?

> To make the architecture decisions first class entities in software architecture evaluation a new evaluation method is required. As a result, this study introduces decision-centric architecture review (DCAR) method which documents and uses architecture decisions as a central element in the evaluation. The benefit of this kind of method is that it is lightweight and designed for iterative and incremental use. The method is lightweight enough to be carried out after each Scrum sprint if necessary. In addition, as a side effect the evaluation method produces

documentation that can be used as a part of the software architecture documentation.

B. How can software architecture evaluation be aligned with agile software development?

Software architecture evaluation is a part of the larger activity: architecture knowledge management. The results present models for aligning software architecture knowledge management and agile software development. The Scrum framework was used as an example of agile process and models on how to align software architecture evaluation and other software architecture work activities with Scrum was presented.

11.1 Limitations of the Study

The main limitations of this study are related to the limitations of the selected research methods. While the case study research is well-suited for software engineering as it allows studying a phenomena which would be hard to study in isolation [28], it has some limitations. Case study research is often criticized for not producing generalizable results [29]. Validity of the results got with survey research largely depends on the covering and wide range of questions asked. Additionally, one needs to make sure that the group of respondents is not biased. The validity of this study is examined by using a classification scheme introduced by [26]. This classification scheme is also recommended to be used in the field of software engineering by [28]. This classification scheme uses four aspects to study the validity of the study: *Construct validity*, *internal validity*, *external validity* and *reliability*.

Construct validity reflects to what extent the operational measures that are studied really represent what the researcher had in mind. Basically, this means that the researcher and the interviewed person should have a common vocabulary and share common understanding on the goal and target of the study. If the interview questions are not interpreted in the same way by the researcher and the interviewed person, construct validity is threatened. To cope with this threat, questions of interviews and questionnaires were piloted once before carrying out the real interviews. In addition, interviews included questions to clarify the concepts discussed. For example, interviews regarding the DCAR method had questions where the interviewee had to explain what is architecture decision, decision context, etc. In this way, the researcher could verify that the terms and questions are interpreted in the same way. Additionally, in interviews, the collected data was both taped and transcribed in real time to have evidence from multiple sources.

Internal validity is a concern when causal relations are examined. Internal validity might become threatened if there is a third factor affecting the causal relationship of the two factors examined and the researcher is not aware of the existence of the third factor.

In this study, the internal validity of the research is obtained by interviewing heterogenous group of companies. For example, in the study on architecture work practices, companies having varying backgrounds and domains were interviewed. In this way, the selection bias can be avoided. However, still one threat to internal validity remains: most of the research is done in Finnish software companies. This might affect the results to some extent as Finnish companies might have homogenous work practices which might differ from the practices of software companies elsewhere. However, as the results of this research are in line with the results of other researchers, there is no particular reason to believe that these limitations would danger the credibility of the results.

The results in this thesis build partly on interviews and practical work carried out in and with the software companies. This limits the sample size of surveyed companies and the number of companies where the introduced methods have been tested. This might pose a threat to internal validity. While the sample size in some interviews might be quite low (10 to 15 companies) the results are in line with the related work. Furthermore, face-to-face interviews with practitioners instead of online questionnaires offered possibilities for deeper discussions on the topic and enabled the researcher to understand the phenomena more deeply.

External validity is an aspect of validity which is concerned with to what extent the results are generalizable. The results of this thesis can be generalized and applied in multiple organizations to some extent. For example, the DCAR method can be used in different organizations to all kinds of software architectures as it is generic method which bases on the basic concepts of software architecture such as architecture decisions. Thus, it is not limited to the domain where it has been developed. Similarly, the TopDocs approach can be applied in any organization as the metamodel on which TopDocs bases on consists of fundamental concepts of software architecture. Additionally, software architecture document types generated from the repository can be configured by writing new scripts. Furthermore, TopDocs provides an extension mechanism that has been successfully applied in the domain of safety-critical systems [47]. However, while models to align architecture knowledge management and Scrum can be found from the results of this thesis, there might be a need to adapt them to the organization where they will be applied. As there are no two similar organizations, an out-of-the-box solution can not be provided. Still, the models provide support and guidelines on how to apply architecture practices in an organization depending on the case. To summarize, the results in this thesis are generalizable, but when the results are applied to different organizations some changes to proposed methods may be needed.

Reliability aspect of validity may be threatened if the collected data and the analysis depend on the specific researcher. In other words, another researcher conducting the same study later on should get the same results. To cope with this threat to validity, the specific research papers contain an explanation on how the study was carried out. For example, the pattern mining process is explained in detail in Section 4.1. However, some parts of the research deal with aspects which are highly dependent on the time of research. For example, architecture work practices in Scrum may change over time even radically, when organizations find out better ways of working. Similarly, software architecture information flows in an organization change when the organization evolves and its structures change. Thus, the results got in this study heavily depend on the practices that have been used at the time of the study. Organizations learn and are likely to change their practices as they learn. Therefore, one might get slightly different results when conducting the same studies again.

11.2 Future Work

In general, the future research directions are multifaceted. In the following the main future directions of this work are discussed.

First, pattern languages are never finished and are supposed to grow as new patterns are discovered. Thus, one of future research direction could be to extend the pattern language for distributed control systems and find out examples from neighboring domains as medical systems. Furthermore, the pattern mining process can be adapted to mine patterns for new domains. One promising direction is to mine software startup patterns [174]. As the pattern mining process is applied to new domains, it is expected to evolve.

Second, decision-centric architecture review method could also be refined and further studied. One possible setting to study DCAR could be a software architecture course where students could use DCAR in parallel with ATAM on similar systems. This approach would offer a possibility to study the strengths and weaknesses of the method in a controlled environment. Another aspect of DCAR that needs further investigation is incremental and iterative evaluations. This study needs to be carried out in an industrial setting to provide realistic environment. However, the challenge is to find a development team in the industry which is willing to try the method on their own.

Third, as social media is becoming more popular also among the enterprises, the possibility to add social media aspects to TopDocs architecture knowledge base could form an interesting future research direction. Architecture knowledge base could be extended outside a company boundary to allow communication with other architects and experts. Architectural solutions could be shared in a social network and the architect could get feedback on the design outside the company. This could also alleviate the communication problems with third parties such as subcontractors. Additionally, gamification and similar approaches could be used to encourage communication and knowledge exchange.

Fourth, architecture knowledge flows in industry should be further researched to find out what kind of AKM models would be suitable for small agile projects. In small projects, the overhead of AKM can still become too large quite easily. Thus, defining the minimum useful architecture knowledge management flow can be seen as a part of the future work.

Finally, studying how architectural information is utilized in continuous deployment of software systems can be seen as a new research direction that builds upon this work. Research on what kind of architectural approaches might slow down or boost the process of continuous delivery and continuous deployment can also be part of the future work.

11.3 Final Remarks

Agile software development has already changed the software industry. Agile has also changed the way architecture work is carried out in organizations. Companies have adopted the concept of *value* of lean software development and optimized the amount of value they produce to the customer with techniques like value stream mapping.

While it is clear for the developers and architects that well-designed software architecture has value, the customer does not see software architecture per se. Thus, in agile development the focus easily shifts away from designing solid software architecture to creating finished product which is just cobbled together. The value of software architecture is only reflected in the final product and thus the value of software architecture is often invisible to the customer. Thus, the developers need to sell the idea of investing in software architecture to the customer. The customer needs to understand why the software architecture work is valuable and worthy of the time invested in it.

On the other hand, software architecture practices must prove their value to the developers: what is the benefit of a practice and why it should be used. For example, software architecture evaluation methods need to be streamlined so that they don't produce any wasted artifacts – all produced artifacts should be usable in achieving the goal set for the project.

Similarly software architecture knowledge management needs to be non-disrupting for the developers and provide value from day one. Often architecture knowledge repositories gain their value over time when the knowledge stored in the repository helps developers to remember things. However, the value of the stored information should be stressed up-front. If developers don't see the value of the architecture knowledge repository up-front, it often leads to a situation where the repository becomes information junkyards that nobody uses. This indicates that the researchers need to find even more convincing evidence that repositories prevent knowledge vaporization and are valuable in the long run. In addition, researchers should find ways to increase the up-front value of the repository.

In this thesis, techniques and practices for software architecture work in agile software development have been presented. These techniques aim to help developers and architects to rapidly deliver software with solid software architecture to a customer, while taking care of that the most important aspects are sufficiently documented.

REFERENCES

[1] V.-P. Eloranta, J. Koskinen, M. Leppänen and V. Reijonen, Designing Distributed Control Systems: A Pattern Language Approach, John Wiley & Sons, 2014.

[2] V.-P. Eloranta, U. van Heesch, P. Avgeriou, N. Harrison and K. Koskimies, "Lightweight Evaluation of Software Architecture Decisions", in *System Quality and Software Architecture,* Elsevier Science & Technology Books, 2014.

[3] SEI, "Community Software Architecture Definitions", 2014. [Online]. Available: http://www.sei.cmu.edu/architecture/start/glossary/community.cfm. [Accessed 14. 12. 2014].

[4] IEEE, "Recommended Practice for Architectural Description of Software-Intensive Systems IEEE standard 1471-2000", IEEE, 2000.

[5] J. Bosch, "Software Architecture: The Next Step" in *Software Architecture 3047 Lecture Notes in Computer Science*, Springer, 2004, pp. 194-199.

[6] S. McConnell, Code Complete (First ed.), Microsoft Press, 1993.

[7] Agile Alliance, "Manifesto for Agile Software Development", 2001. [Online]. Available: http://agilemanifesto.org. [Accessed 9.5. 2012].

[8] VersionOne, "8th Annual State of Agile Survey", 2013. [Online]. Available: http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf. [Accessed 04. 11. 2014].

[9] P. Kruchten, "Software Architecture and Agile Software Development: A Clash of Two Cultures?" in *Proceedings of 32nd International Conference on Software Engineering (ICSE),* 2010.

[10] Apple Inc., "Model-View-Controller", 17. 09. 2013. [Online]. Available: https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html. [Accessed 23. 11. 2014].

[11] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä and T. Männistö, "The Speedways and Country Roads to Continuous", in *Accepted for publishing in IEEE Software special issue on Release engineering,* 2014.

[12] T. Chow and D.-B. Cao, "A survey study of critical success factors in agile software projects", *Journal of Systems and Software,* vol. 81, no. 6, p. 961–971, 2008.

[13] Kone Corporation, "Lifecycle Analysis", 2014. [Online]. Available: http://www.environdec.com/FileUpload/RE%20Comments%20overview%20until%207th%20Feb%202014/SuppDoc6_impacts_manufacturing_stage.pdf.

[Accessed 02. 11. 2014].

[14] W. W. Royce, "Managing the Development of Large Software Systems", in *Proceedings of IEEE WESCON*, 1970.

[15] N. Harrison and J. Sutherland, *Discussion on Value Stream in Scrum,* 2013.

[16] R. Kazman, M. Klein and P. Clements, "ATAM: Method for Architecture Evaluation", Software Engineering Institute, Carnegie Mellon University, 2000.

[17] P. Abrahamsson, M. A. Babar and P. Kruchten, "Agility and Architecture: Can they coexist?", *IEEE Software,* vol. 27, no. 2, pp. 16-22, 2010.

[18] J. O. Coplien and G. Bjørnvig, Lean Architecture: for Agile Software Development, John Wiley & Sons, 2010.

[19] P. Lago and P. Avgeriou, "First workshop on sharing and reusing architectural knowledge" in *SIGSOFT Software Engineering Notes 31, 5,* 2006.

[20] I. A. Nonaka and H. Takeuchi, The knowledge-creating company: How Japanese Companies Create the Dynamics of Innovation, Oxford University Press, 1995.

[21] R. Farenhorst and R. C. de Boer, Architectural knowledge management: Supporting Architects and Auditors, VU University, 2009.

[22] P. Clements, R. Kazman and M. Klein, Evaluating Software Architectures, Addison-Wesley, 2002.

[23] J. Maranzano, S. Rozsypal, G. Zimmerman, P. Warnken and D. Weiss, "Architecture reviews: Practice and Experience", *IEEE Software,* vol. 22, no. 2, pp. 34-43, 2005.

[24] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions", in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA),* 2005.

[25] I. Benbasat, D. K. Goldstein and M. Mead, "The Case Research Strategy in Studies of Information Systems", *Management Information Systems Quarterly,* vol. 11, no. 3, pp. 369-386, September 1987.

[26] R. K. Yin, Case Study Research: Design and Methods, 3rd ed., SAGE Publications, Inc, 2003.

[27] C. Robson, Real World Research 3rd edition, Wiley, 2011.

[28] P. Runeson and M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering", *Empirical Software Engineering,* vol. 14, no. 2, pp. 131-164, 2009.

[29] B. Flyvbjerg, ”Five Misunderstandings About Case-Study Research”, *Qualitative Inquiry,* osa/vuosik. 12, nro 2, pp. 219-245, 2006.

[30] A. R. Hevner, S. T. March, J. Park and S. Ram, "Design science in information systems research", *MIS Quarterly. Volume 28, Issue 1,* pp. 75-105, 2004.

[31] S. March and G. Smith, "Decision Support Systems", *Design and Natural Science Reseach on Information Technologies,* pp. 251-266, 1995.

[32] M. Markus, A. Majchrzak and L. Gasser, "A Design Theory for Systems that Support Emergent Knowledge Process", *MIS Quarterly,* vol. 26, no. 3, pp. 179-212, 2002.

[33] J. Walls, G. Widmeyer and O. El Sawy, "Building an Information System Design Theory for Vigilant EIS", *Information Systems Research,* vol. 3, no. 1, pp. 36-59, 1992.

[34] J. Shaughnessy, E. Zechmeister ja Z. Jeanne, Research Methods in Psychology (9th ed.), New York, NY: McGraw Hill, 2011.

[35] M. Q. Patton, Qualitative Research & Evaluation Methods, Thousand Oaks: SAGE, 1991.

[36] S. Easterbrook, J. Singer, M.-A. Storey and D. Damian, "Selecting Empirical Methods for Software Engineering Research", in *Guide to Advanced Empirical Software Engineering,* London, Springer, 2008, pp. 285-311.

[37] B. Glaser and A. L. Strauss, "The discovery of grounded theory: strategies for qualitative research", 1967.

[38] B. Glaser, Emergence vs. Forcing: Basics of Grounded Theory Analysis, Mill Valley, Ca: Sociology Press, 1992, pp. , Mill Valley, Ca.: Sociology Press.

[39] C. Hentrich, U. Zdun, V. Hlupic and F. Dotsika, "An Approach for Supporting Pattern Mining and Validation through Grounded Theory and its Applications to Process-Driven SOA Patterns", 2013.

[40] D. Avison, F. Lau, M. D. Myers and P. A. Nielsen, "Action Research", *Communications of the ACM,* vol. 42, no. 1, pp. 94-97, January 1999.

[41] P. S. Medeiros dos Santos and G. H. Travassos, "Action research use in software engineering: An initial survey" in *In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09),* Washington, DC, 2009.

[42] G. I. Susman and R. D. Evered, "An Assessment of the Scientific Merits of Action Research", *Administrative Science Quarterly, Volume 23, No. 4,* pp. 582-603, 1978.

[43] P. Kruchten, "Voyage in the Agile Memeplex: Agility, Agilese, Agilitis, Agilology", *ACM Queue,* pp. 38-44, 2007.

[44] L. Bass and R. L. Nord, "Understanding the Context of Architecture Evaluation Methods" in *Proceedings of Joint Working IEEE/IFIP Conference on Software*

Architecture (WICSA) and European Conference on Software Architecture (ECSA), Helsinki, 2012.

[45] Z. Li and J. Zheng, "Toward Industry Friendly Software Architecture Evaluation" in *Proceedings of 7th European Conference on Software Architecture (ECSA)*, Montpellier, 2013.

[46] V.-P. Eloranta and K. Koskimies, "Aligning Architecture Knowledge Management with Scrum" in *WICSA/ECSA '12 Proceedings of the WICSA/ECSA 2012 Companion Volume*, New York,. USA, 2012.

[47] T. Vepsäläinen, S. Kuikka and V.-P. Eloranta, "Software Architecture Knowledge Management for Safety Systems" in *Proceedings of IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2012.

[48] V.-P. Eloranta, K. Koskimies, T. Mikkonen ja J. Vuorinen, ”Scrum Anti-Patterns - An Empirical Study” in *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC' 2013)*, Bangkok, Thailand, 2013.

[49] ISO/IEC, "42010:2011 - Systems and Software Engineering - Architecture description", International Organization for Standardization, 2011.

[50] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, 2nd ed., Addison-Wesley Professional, 2003.

[51] P. Kruchten, "Architectural Blueprints - The "4+1" View Model of Software Architecture", *IEEE Software*, vol. 12, no. 6, pp. 42-50, 1995.

[52] M. A. Babar, T. Dingsoyr, P. Lago and H. van Vliet, Software Architecture Knowledge Management, Berlin Heidelberg: Springer, 2009.

[53] P. Kruchten, "An ontology of architectural design decisions in software intesive systems" in *Proceedings of 2nd Groningen Workshop on Software Variability*, Groningen, 2004.

[54] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran and P. America, "Generalizing a Model of Software Architecture Design from Five Industrial Approaches" in *Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture*, 2005.

[55] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions" in *Proceedings of WICSA 2005*, 2005.

[56] R. Kazman and L. Bass, "Toward Deriving Software Architectures From Quality Attributes", Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, 1994.

[57] P.-O. Bengtsson, "Architecture-Level Modifiability Analysis (ALMA)", *Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129-147, 2004.

[58] T. Dolan, "Architecture Assessment of Information-System Families", Eindhoven University of Technology Ph.D Thesis, Department of Technology Management, Eindhoven, 2002.

[59] P. Clements, "Active Reviews for Intermediate Designs", Carnegie-Mellon University, Pittsburgh, 2000.

[60] L. Dobrica and Niemelä, E., "A Survey on Software Architecture Analysis Methods", *IEEE Transactions on Software Engineering,* vol. 28, no. 7, pp. 638-653, 2002.

[61] E. Woods, "Industrial Architectural Assessment Using Tara" in *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA),* 2011.

[62] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop and A. Zaremski, "Recommended Best Industrial Practice for Software Architecture Evaluation", Software Engineering Institute / Carnegie-Mellon University, Pittsburgh, 1997.

[63] H. Christensen, K. Hansen and B. Lindström, "Lightweight and Continuous Architectural Software Quality Assurance Using the aSQA Techinque" in *Proceedings of the 4th European Conference on Software Architecture (ECSA),* 2010.

[64] R. Kazman, J. Asundi and M. Klein, "Quantifying the Cost and Benefits of Architectural Decisions" in *Proceedings of 23rd International Conference on Software Engineering (ICSE),* 2001.

[65] R. Kazman, J. Asundi and M. Klein, "Making Architecture Design Decisions: An Economic Approach", Software Engineering Institute Carnegie-Mellon University, Pittsburgh, 2002.

[66] J. Asundi, R. Kazman, M. Klein and M. Moore, "Quantifying the Value of Architecture Design Decisions: Lessons from the Field", Software Engineering Institute / Carnegie-Mellon University, Pittsburgh, 2003.

[67] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, Documenting Software Architecture - Views and Beyond, Boston: Addison-Wesley, 2003.

[68] A. Cockburn, Agile Software Development: The cooperative game 2nd edition, Reading, MA: Addison-Wesley, 2007.

[69] P. Kruchten, "Documentation of Software Architecture from a Knowledge Management Perspective - Design Representation" in *Software Architecture Knowledge Management*, Springer-Verlag Berlin Heidelberg, 2009, pp. 39-57.

[70] R. Farenhorst and R. de Boer, "Knowledge Management in Software Architecture: State of the Art" in *Software Architecture Knowledge Management*, Springer Berlin Heidelberg, 2009, pp. 21-38.

[71] P. Kruchten, P. Lago, H. van Vliet and T. Wolf, "Building up and Exploiting Architectural Knowledge" in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA),* Pittsburgh, Pennsylvania, Usa, 2005.

[72] T. Davenport, "Saving IT's Soul: Human Centered Information Management", *Harvard Business Review,* vol. 72, no. 2, pp. 119-131, 1994.

[73] T. Dingsøyr and H. Vliet, "Introduction to Software Architecture and Knowledge Management" in *Software Architecture Knowledge Management,* Springer, 2009, pp. 1-17.

[74] M. Hansen, N. Nohria and T. Tierney, "What is your strategy for managing knowledge?", *Harvard Business Review,* vol. 77, no. 2, pp. 106-116, 1999.

[75] M. Earl, "Knowledge Management Strategies: Towards a Taxonomy", *Journal of Management Information Systems,* vol. 18, no. 1, pp. 215-233, 2001.

[76] R. Capilla, F. Nava, S. Pérez and J. Dueñas, "A web-based tool for managing architectural design decisions" in *Proceedings of the First Workshop on Sharing and Reusing Architectural Knowledge (SHARK)*, Torino, Italy, 2006.

[77] M. Babar, A. Northway, I. Gorton, P. Heuer and T. Nguyen, "Introducing Tool Support for Managing Architectural Knowledge: An Experience Report" in *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems,* 2008.

[78] A. Tang, P. Avgeriou, A. Jansen, R. Capilla and M. Babar, "A Comparative Study of Architecture Knowledge Management Tools", *The Journal of Systems and Software,* pp. 352-370, 2010.

[79] T. Dingsøyr, "Strategies and Approaches for Managing Architectural Knowledge" in *Software Architecture Knowledge Management,* Springer Berlin Heidelberg, 2009, pp. 59-68.

[80] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture", *IEEE Software*, pp. 19-27, 2005.

[81] U. van Heesch, P. Avgeriou and R. Hilliard, "A Documentation Framework for Architecture Decisions", *Journal of Systems and Software,* vol. 85, no. 4, pp. 795-820, 2012.

[82] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann and N. Schuster, "Reusable Architectural Decision Models for Enterprise Application" in *Software Architectures, Components, and Applications,* Springer Berlin Heidelberg, 2007, pp. 15-32.

[83] M. A. Babar, "Supporting the Software Architecture Process with Knowledge Management" in *Software Architecture Knowledge Management,* Springer Berlin Heidelberg, 2009, pp. 69-86.

[84] J. C. Dueñas and R. Capilla, "The Decision View of Software Architecture: Building by Browsing" in *Proceedings of 2nd European Workshop on Software Architecture (EWSA),* Pisa, Italy, 2005.

[85] P. Kruchten, R. Capilla and J. Dueas, "The Decision View's Role in Software Architecture Practice", *IEEE Software,* vol. 26, no. 2, pp. 36-42, 2009.

[86] A. Tang, M. Babar, I. Gorton and J. Han, "A Survey of the Use and Documentation of Architecture Design Rationale" in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005),* Pittsburgh, PA, USA, 2005.

[87] D. J. Anderson, Kanban: Successful Evolutionary Change for Your Technology Business, Blue Hope Press, 2010, p. 278.

[88] K. Beck, Extreme Programming Explained: Embrace Change, 2nd ed., Addison-Wesley, 2000.

[89] M. Poppendieck and T. Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley, 2003.

[90] K. Schwaber, "Scrum Development Process" in *Proceedings of the 10th annual ACM conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA),* 1995.

[91] K. Schwaber and M. Beedle, Agile Software Development with Scrum, Upper Saddle River, NJ, USA: Prentice-Hall, 2001.

[92] J. Sutherland and K. Schwaber, "The Scrum guide – the definitive guide to Scrum: The rules of the game", 2011. [Online]. Available: http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf. [Accessed 9. 9. 2011].

[93] P. M. Duvall, S. Matyas and A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley, 2007.

[94] M. Coram and S. Bohner, "The impact of agile methods on software project management" in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005. ECBS '05.*, 2005.

[95] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[96] W. Cunningham, "The WyCash Portfolio Management System" in *OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications*, New York, NY, USA, 1992.

[97] W. Zuill, "Mob Programming Basics," 14. 11. 2012. [Online]. Available: http://mobprogramming.org/mob-programming-basics/. [Accessed 23. 11. 2014].

[98] VersionOne, "7th Annual State of Agile Development survey", 2012. [Online]. Available: http://www.versionone.com/pdf/7th-Annual-State-of-Agile-

Development-Survey.pdf. [Accessed 23. 11. 2014].

[99] M. McHugh, F. McCaffery and V. Casey, "Barriers to Adopting Agile Practices When Developing Medical Device Software" in *Software Process Improvement and Capability Determination*, Springer Berlin Heidelberg, 2012, pp. 141-147.

[100] A. Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley, 2004.

[101] G. H. Fairbanks, Just Enough Software Architecture: A Risk-Driven Approach, Marshall & Brainerd, 2010.

[102] M. Leppänen, J. Koskinen and T. Mikkonen, "Discovering a Pattern Language for Embedded Machine Control Systems Using Architecture Evaluation Method", Tampere, Finland, 2009.

[103] V.-P. Eloranta, J. Koskinen,[M. Leppänen and V. Reijonen, "A pattern Language for Distributed Machine Control Systems", Tampere University of Technology, Department of Software Systems, Report, vol. 9, pp. 108, Tampere, ISBN 978-952-15-2319-9, 2010.

[104] Portland Pattern Repository, "Canonical form", 2003. [Online]. Available: http://c2.com/cgi/wiki?CanonicalForm. [Accessed 19. 9. 2013].

[105] Portland Pattern Repository, "Alexandrian form", 2011. [Online]. Available: http://c2.com/cgi/wiki?AlexandrianForm. [Accessed 19. 9. 2013].

[106] N. B. Harrison, P. Avgeriou and U. Zdun, "Using Patterns to Capture Architectural Decisions", *IEEE Software,* vol. 24, no. 4, pp. 38-45, 2007.

[107] Hillside Group, "PLoP Conferences", 2013. [Online]. Available: http://hillside.net/conferences/. [Accessed 29. 10. 2013].

[108] R. P. Gabriel, Writer's Workshop and the Work of Making Things, Boston, MA, USA: Addison-Wesley, Longman Publishing, 2002.

[109] M. Leppänen, "Patterns for Messaging in Distributed Machine Control Systems", Tampere, Finland, 2013.

[110] R. Hanmer, Patterns for Fault Tolerant Software, Wiley, 2007.

[111] IEE, "Embedded Systems and Year 2000 Problem: Guidance Notes", IEE Technical Guidelines 9, 1997.

[112] International ISO Standard, "ISO 11898-2:2003 Road vehicles - Controller area network (CAN) - Part 2: High-speed medium access unit", International Standardization Organization, 2003.

[113] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Professional, 2003.

[114] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded

Applications., Norwell, MA: Kluwer Academic Publishers, 1997.

[115] W. Herzner, W. Kubinger and M. Gruber, "Triple-T (Time-Triggered-Transmission) - A System of Patterns for Reliable Communication in Hard Real-Time Systems" in *Proceedings of 9th European Conferenfce on Pattern Languages of Programs (EuroPLoP 2004),* Irsee, Germany, Hillside Europe, 2004.

[116] EventHelix.com Inc., "STL Design Patterns II", 2014. [Online]. Available: http://www.eventhelix.com/realtimemantra/patterns/stl_design_patterns_2.htm#. UvoCAxDEndc. [Accessed 11. 02. 2014].

[117] F. Buschmann, K. Henney and D. C. Schmidt, Pattern-Oriented Software Architecture: On Patterns and Pattern Languages, Volume 5, John Wiley & Sons, 2007.

[118] L. Bass, R. Nord, W. Wood, D. Zubrow and I. Ozkaya, "Analysis of Architecture Evaluation Data", *Journal of Systems and Software,* vol. 81, no. 9, pp. 1443-1455, 2008.

[119] L. Bass, M. Klein and G. Moreno, "Applicability of General Scenarios to the Architecture Trade-off Analysis Method", Carnegie Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania, 2001.

[120] R. Kazman, S. J. Carriere and S. G. Woods, "Toward a Discipline of Scenario-based Architectural Engineering", *Annals of Software Engineering,* vol. 9, no. 1-4, pp. 5-33, 2000.

[121] M. A. Babar and S. Biffl, "Eliciting Better Quality Architecture Evaluation Scenarios: A Controlled Experiment on Top-down vs. Bottom-up" in *Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering (ISESE 2006),* New York, NY, 2006.

[122] P.-O. Bengtsson and J. Bosch, "An experiment on creating scenario profiles for software change", *Annals of Software Engineering,* vol. 9, no. 1-4, pp. 59-78, 2000.

[123] J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach, New York, NY: ACM Press/Addison-Wesley, 2000.

[124] V.-P. Eloranta, J. Koskinen, M. Leppänen and V. Reijonen, "A Pattern Language for Distributed Machine Control Systems," Tech. Report Tampere University of Technology, Tampere, 2010.

[125] F. Bachmann, L. Bass and M. Klein, "Deriving Architectural Tactics: A Step Toward Methodical Architectural Design", Carnegie Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania, 2003.

[126] N. Lassing, D. Rijsenbrij and H. van Vlient, "The goal of software architecture

analysis: confidence building or risk assessment" in *Proceedings of first BeNeLux Conference on Software Architecture*, 1999.

[127] A. Jansen, T. de Vries, P. Avgeriou and M. van Veelen, "Sharing the Architectural Knowledge of Quantitative Analysis" in *Quality of Software Architectures. Models and Architectures*, Springer Berlin Heidelberg, 2008, pp. 220-234.

[128] P. Liang and P. Avgeriou, "Tools and Technologies for Architecture Knowledge Management" in *Software Architecture Knowledge Management - Theory and Practice*, Springer, 2009, pp. 91-111.

[129] L. Bass and R. Nord, "Understanding the Context of Architecture Evaluation Methods" in *Proceedings of Joint Working IEEE/IFIP Conference onSoftware Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, 2012.

[130] M. Babar, L. Bass and I. Gorton, "Factors Influencing Industrial Practices of Software Architecture Evaluation: An Empirical Investigation" in *Proceedings of the Quality of Software Architectures 3rd International Conference on Software Architectures, Components, and Applications*, 2007.

[131] U. van Heesch, P. Avgeriou and R. Hilliard, "Forces on Architecture Decisions - A Viewpoint" in *Proceedings of Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, 2012.

[132] N. Harrison and P. Avgeriou, "Pattern-based Architecture Reviews", *IEEE Software*, vol. 28, no. 6, pp. 66-71, 2010.

[133] R. Nord and J. Tomayko, "Software Architecture-centric Methods and Agile Development", *IEEE Software*, pp. 47-53, 2006.

[134] VersionOne, "5th annual state of agile survey", 2010. [Online]. Available: http://www.versionone.com/pdf/2010_State_of_Agile_Development_Survey_Results.pdf. [Accessed 30. 12. 2011].

[135] M. Huo, J. Verner, L. Zhu and M. Babar, "Software quality and agile methods" in *Proceedings of the COMPSAC 2004*, 2004.

[136] K. Korhonen, "Evaluating the effect of agile methods on software defect data and defect reporting practices - a case study" in *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology*, 2010.

[137] VersionOne, "4th annual state of agile survey", 2009. [Online]. Available: http://www.versionone.com/pdf/2009_State_of_Agile_Development_Survey_Results.pdf. [Accessed 30. 12. 2011].

[138] Scrum Pattern Community, "Published patterns", 2011. [Online]. Available: http://sites.google.com/a/scrumplop.org/published-patterns/. [Accessed 10. 8. 2011].

[139] Scrum.org, "Scrum Extension Library", 2012. [Online]. Available: http://www.scrum.org/scrum-extensions/. [Accessed 2. 1. 2012].

[140] C. Seaman, "Qualitative methods in empirical studies of software engineering", *IEEE Software*, vol. 27, no. 2, pp. 16-22, 2010.

[141] J. Sutherland, "Scrumbut test aka the Nokia test", 2010. [Online]. Available: http://jeffsutherland.com/scrumbutttest.pdf. [Accessed 19. 11. 2011].

[142] V.-P. Eloranta, "Scrum and architecture work survey – interview questions", 2010. [Online]. Available: http://www.cs.tut.fi/~elorantv/interview_questions.html. [Accessed 10. 1. 2012].

[143] J. Sutherland, "Nokia test: Where did it come from?", 2008. [Online]. Available: http://scrum.jeffsutherland.com/2008/08/nokia-test-where-did-it-come-from.html. [Accessed 19. 11. 2011].

[144] D. Rawsthorne, "Sprint zero", 2007. [Online]. Available: http://blogs.danube.com/sprint-zero?q=blog/dan_rawsthorne/sprint_zero. [Accessed 30. 8. 2011].

[145] M. Babar, "An exploratory study of architectural practices and challenges in using agile software development approaches" in *Proceedings of the European Conference on Software Architecture (ICSA/ECSA 2009)*, 2009.

[146] K. Schougaard, K. Hansen and H. Christensen, "Sa@work a field study of software architecture and software quality at work" in *Proceedings of the 15th Asia-Pacific Software Engineering Conference APSEC 2008*, 2008.

[147] M. Babar, T. Ihme and M. Pikkarainen, "An industrial case of exploiting product line architectures in agile software development" in *Proceedings of the 13th international software product line conference (SPLC'09)*, 2009.

[148] R. Baskerville, J. Pries-Heje and S. Madsen, "Post-agility: What follows a decade of agility?", *Information and Software Technology*, vol. 53, no. 5, pp. 543-555, 2010.

[149] M. Isham, "Agile Architecture IS Possible – You First Have to Believe!" in *Proceedings of Agile 2008 conference*, 2008.

[150] R. Farenhorst, P. Lago and H. van Vliet, "Effective Tool Support for Architectural Knowledge Sharing" in *Software Architecture*, LNCS Springer, 2007, pp. 123-138.

[151] IEC, "IEC 61508-3 standard: Functional safety of electrical/electronic/programmable electronic safety-related systems part 3",

International Electrotechnical Commision, 2010.

[152] Polarion Software, "Application Life Cycle Management, Requirements & Quality Assurance Software Solutions", 2012. [Online]. Available: http://www.polarion.com. [Accessed 2. 4. 2012].

[153] J. van der Ven, A. Jansen, P. Avgeriou and D. Hammer, "Using Architectural Decisions" in *Proceedings of the Second International Conference on the Quality of Software Architecture (QoSA, 2006).*, Västerås, Sweden, 2006.

[154] P. Kruchten, P. Lago and H. van Vlient, "Building up and reasoning about architectural knowledge" in *Proceedings of the Second International Conference on the Quality of Software Architectures*, Västerås, Sweden, 2006.

[155] R. Laine and J. Peltonen, "IDLE Learning environment", 2006. [Online]. Available: http://www.cs.tut.fi/~idle/index.html. [Accessed 29. 3. 2012].

[156] M. Su, J. Hosking and J. Grundy, "KaitoroCap: a document navigation" in *Proceedings of 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011),* 2011.

[157] P. Avgeriou, P. Lago and P. Kruchten, "Towards Using Architectural Knowledge", *SIGSOFT Sofware Engineering Notes*, vol. 34, no. 2, pp. 27-30, 2009.

[158] M. A. Babar, R. C. de Boer, T. Dingsøyr and R. Farenhorst, "Architectural Knowledge Management Strategies: Approaches in Research and Industry" in *Proceedings of Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent. SHARK/ADI '07: ICSE Workshops 2007,* Minneapolis, MN, 2007.

[159] D. Weyns and B. Michalik, "Codifying Architecture Knowledge to Support Online Evolution of Software Product Lines" in *Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge (SHARK'11),* New York, NY, 2011.

[160] K. de Graaf, A. Tang, P. Liang and H. van Vliet, "Ontology-based Software Architecture Documentation" in *Proceedings of 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA),* Helsinki, 2012.

[161] A. Jansen, P. Avgeriou and J. S. van der Ven, "Enriching Software Architecture Documentation", *Journal of Systems and Software,* vol. 82, no. 8, pp. 1232-1248, 2009.

[162] H. van Vliet, P. Avgeriou, R. de Boer, V. Clerc, R. Farenhorst, A. Jansen and P. Lago, "The GRIFFIN Project: Lessons Learned" in *Software Architecture Knowledge Management,* Springer Berlin Heidelberg, 2009, pp. 137-154.

[163] V.-P. Eloranta, O. Hylli, T. Vepsäläinen and K. Koskimies, "TopDocs: Using Software Architecture Knowledge Base for Generating Topical Documents" in *Proceedings of Software Architecture (WICSA) and European Conference on Software Architecture (ECSA),* 2012.

[164] VersionOne, "6th Annual State of Agile Survey", 2011. [Online]. Available: http://www.versionone.com/pdf/2011_State_of_Agile_Development_Survey_Results.pdf. [Accessed 28. 08. 2012].

[165] V.-P. Eloranta and K. Koskimies, "Software Architecture Practices in Agile Enterprises" in *Aligning Enterprise, System, and Software Architectures,* Hershey, PA, IGI Global, 2012, pp. 230-249.

[166] U. van Heesch, V.-P. Eloranta, P. Avgeriou, K. Koskimies and N. Harrison, "DCAR - Decision Centric Architecture Reviews", *IEEE Software,* 2013.

[167] R. Kazman, L. Bass, M. Webb and G. Abowd, "SAAM: A Method for Analyzing the Properties of Software Architectures" in *Proceedings of the 16th International Conference on Software Engineering (ICSE),* 1994.

[168] U. van Heesch and P. Avgeriou, "Mature Architecting - A Survey About the Reasoning Process of Professional Architects" in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA),* 2011.

[169] V.-P. Eloranta and K. Koskimies, "Using Domain Knowledge to Boost Software Architecture Evaluation" in *Proceedings of European Conference on Software Architecture (ECSA'10),* Heidelberg, 2010.

[170] J. S. Van Der Ven, A. Jansen, J. Nijhuuis and J. Bosch, "Design Decisions: The Bridge between Rationale and Architecture, in *Rationale Management in Software Engineering*, Berlin, Springer, 2006, pp. 329-348.

[171] C. Hofmeister, 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,* vol. 80, no. 1, pp. 106-126, 2007.

[172] D. Leffingwell, Scaling Software Agility: Best Practices for Large Enterprises, Upper Saddle River, NJ: Addison-Wesley Professional, 2007.

[173] K. Desouza, Y. Awazu and P. Baloh, "Managing Knowledge in Global Software Development Efforts: Issues and Practices", *IEEE Software,* vol. 23, no. 5, pp. 30-37, 2006.

[174] A. Dande, V.-P. Eloranta, Hadaytullah, A.-J. Kovalainen, T. Lehtonen, M. Leppänen, T. Salmimaa, M. Sayeed, M. Vuori, C. Rubattel, W. Weck and K. Koskimies, "Software Startup Patterns. An Emipiral Study", Tampere University of Technology, Department of Pervasive Computing, Report 4, Tampere, 2014.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3507-9
ISSN 1459-2045