Chapter 1

Introduction

“The secret to getting ahead is getting started.”

- Mark Twain

1.1 Introduction

When I was young, I always took a city plan with me when visiting an unknown town. If I wanted to know my location, I managed with the street-signs, the index of the city plan and some orienteering-skills. When I would meet up with someone, I used my wired phone to contact the person a few days ahead to make an appointment of where and when to meet. My sister used to send me letters when she lived in Australia. It took some weeks to communicate, but it was great to be able to hear something from her from the other side of the world. I never do any of the above-mentioned activities anymore, software radically changed them all.

Now, when I visit a town, the primary concern is that my smart-phone is charged, so the software can assist me when I need it. GPS satellites help me to show me were I am, Internet applications provide detailed maps of the world. I call my appointment when I am near to see where we meet, and send a message to my friend in the US as I walk there. It is great to see his response almost instantly from the other side of the world.

Software has an enormous impact on everyday live. It enables us to be more productive and have rich communication with each other, but also facilitate the distribution of fake-news and addicts us to our mobile devices. Software is included in traditionally physical objects like toothbrushes or thermostats. Data is produced in enormous quantities and this data is increasingly used to gain insights, make better decisions or sell products. Even in domains that have for a long time increased productivity by innovating the physical devices are being disrupted by software-driven products.

Software systems are increasingly interconnected, which creates great opportunities, but also great challenges when building these systems. For example, the BIOSCOPE project \(^1\) involved 5 companies and dozens of systems that needed to interact. It combined the data from weather prediction, satellite images, drone images and predictive algorithms on crop growth and diseases to provide automated advise to farmers. One of the products delivered potato haulm killing taskmaps

\(^1\)https://business.esa.int/projects/bioscope
that can be used direly in (GPS-guided) tractors, based on satellite or drone images that are guaranteed to be less then ten days old.

It is challenging to build software systems like these. Many decisions have to be made to make sure all parts integrate well to meet business needs. For example, where and how should the data be stored? What data is communicated at which frequency? What is done when one of the systems fails? The answers to these topics are typical examples of decisions that form the architecture of software systems. This architecture can be seen as the backbone of the system. Documenting and discussing the software architecture is extremely important for the success of a system [67]. Visualizations in the form of diagrams or models are used to communicate the architecture design. Figure 1.1 shows a diagram that was used in the BIOSCOPE project to discuss the interaction between the system parts.

![Diagram of BIOSCOPE project](image)

**Figure 1.1**: An architectural diagram used in the BIOSCOPE project.

When systems evolve, it is hard to remember why architectural decisions were made, even if the design is visualized and documented. This effect is known as knowledge vaporization [75]. Even though the results of a decision might be documented, the decision process and the reasoning behind the decision is lost, while the system evolved. This thesis investigates how architectural design decisions can be preserved, recovered or reused in order to address architectural knowledge vaporization.
1.2 A Brief History of Software Development

Developing software is a time-consuming and tedious process. In the past decades, the ideas on how to build reliable software have evolved. Software engineering used to be an engineering practice, similar to traditional engineering practices like construction or manufacturing. As time passed, the software industry matured through trial and (sometimes very expensive) error. In order to manage the constantly decreasing time-to-market, the software development process changed. To show the changes, roughly four periods can be distinguished in software engineering, as summarized in Table 1.1.

In the first period (up until ~1980), the computer was new, and specialized companies and research institutes were experimenting with software. This software was mostly monolith (single purpose and single system) [199]. In the beginning

<table>
<thead>
<tr>
<th>Period</th>
<th>Application</th>
<th>Process</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980 - 1995</td>
<td>Software is available for a larger audience, and software development becomes easier.</td>
<td>Waterfall methodology, focus on predictability of projects. Increasingly more expensive to develop software.</td>
<td>Emphasis on good design by documenting architectures. Start of reusing components.</td>
</tr>
<tr>
<td>1995 - 2007</td>
<td>Adoption of the Internet. Software is becoming essential to companies, maintenance is increasingly important. Specialized IT departments emerge, while companies arise that solely do software products.</td>
<td>Increasing complexity as more systems interact. Agile approaches emerge. Projects are technology-focused (“can we make this work?”).</td>
<td>Increasing complexity of software architecture. Extensive architectural documentation based on styles and patterns.</td>
</tr>
<tr>
<td>2007 until now</td>
<td>Software gets into every part of human life. For more companies, software becomes the distinctive element of the business. Much easier to start a software business.</td>
<td>Agile matured, challenges remain in scaling and continuous deployment. User focused projects (“Can we make a business out if this?”).</td>
<td>Consolidation of styles in specific domains. Lightweight architectural documentation, close to the source code.</td>
</tr>
</tbody>
</table>

Table 1.1: Historical Change in Software Engineering and Architecture
Chapter 1. Introduction

of this period, there was no distinction between software and hardware, as both were sold together and software only worked for the hardware it was designed for. The IBM System/360 \(^2\) (development started in 1965) was the first to change this. In this family of computers, the same software could be used in different hardware versions. In the beginning of this period, there was no clear software engineering process as hardware engineering practices were adopted for software development. The designs were closely related to the implementation as there were not many layers of abstraction necessary; the design was the code. Software architecture was not existing, as Kruchten et al. [113] stated, ‘until the late 1980s, the word “architecture” was used mostly in the sense of system architecture’. This changed in the next period.

In the second period (1980-1995), computers and software entered the homes of people with the introduction of the Personal Computer and operating systems like MS-DOS\(^3\). With the greater adoption of third-level programming languages like COBOL or C, computer programs were getting larger and more abstract. It was increasingly more difficult to create software as the cost increased (exponentially) with the size of the systems. Large, long-running projects were hard to predict and often ran over time and budget, or completely failed [33]. The interest from industry and academia in software architecture increased. Because of the high costs of failing projects, there was an increasing interest in carefully documenting the architecture in order to prevent expensive errors. Research emphasized good architectural design to manage cross-cutting concerns like non-functional requirements. Frameworks for structured reuse of code emerged (e.g. COM components and dlls).

Around the change of the millennium (period 1995-2007), the Internet emerged and the increasing communication between software systems increased the complexity software development. Software was used in more domains (B2C and B2B), while companies arose that developed software as their main product. Maintenance of software systems got increasing attention as systems were longer in use and always on. It turned out that problems from bad design decisions could be very expensive (e.g. the Millennium Bug \(^4\)). In order to address the predictability of projects, experiments were conducted with agile software development industry settings. The Agile Manifesto\(^5\) (2001) expressed the common agile values and principles. With these, the emphasis on control and up-front design was slowly being replaced by an more iterative and empirical approach. Projects were technology focused (“Can we make this work?”), and software architecture was responsible for addressing the technological challenges, while meeting nonfunctional qualities like performance or scalability. Methods for quality attribute based architcting, like SAAM [106] and ATAM [107] were developed and used. Research on Architecture Description Languages (ADLs) [138], documenting architectures [42], views [67, 116], architectural styles [164] and patterns [81] was dominant in this period.

\(^2\)https://en.wikipedia.org/wiki/IBM_System/360
\(^3\)https://en.wikipedia.org/wiki/MS-DOS
\(^4\)https://en.wikipedia.org/wiki/Year_2000_problem
\(^5\)http://agilemanifesto.org/
The last period (2007 until now) started with the introduction of the iPhone\(^6\), and therewith the start of mobile Internet. In this period, the Internet matured and stretched to mobile devices and other physical objects (the Internet of Things). As Woods describes, software systems went ‘from being “always on” to being ”used from anywhere for anything” ’ [199]. Standardization of many aspects of the Internet occurred, resulting in Software as a Service (SaaS) and Platform as a Service (PaaS) solutions. This democratized software development as it was possible for a much larger group to develop software. The focus of projects was to validate the business ideas, instead of exploring technological possibilities [156]. Agile software development proved it’s value, starting with small (startup) companies but gaining adoption across the whole software industry. The battles on the architectural styles and patterns were mostly settled as domains grew to a common understanding for their key architectural choices, reducing the need for architectural decisions [92]. In some cases, the ecosystem made the architectural decisions for the developers (e.g. the App stores, J2EE or .NET), while in other cases successful design decisions were copied, which resulted in similarity in design. For example, all large web-frameworks use the MVC architectural style, and REST of SOAP implementations share the same design decisions. In software architecture, there is an increasing emphasis on documenting light-weight decisions, to make it possible to change direction just-in-time [152]. The domain of the software architect is stretched to include automated testing, Continuous Integration and Continuous Delivery, as well as monitoring and updating running systems [92]. As architecture decision-making shifts to being more of a team activity [177], the role of architect changes from decision maker to knowledge manager [197], responsible for sharing knowledge.

### 1.3 Software Architecture

As the previous section described, software architecture changed when software engineering did. The change in perception on software architecture becomes clear when looking at the different definitions that have been used for software architecture. One of the first definitions used is from Perry and Wolf [149]:

A software architecture is a set of architectural elements that have a particular form. We distinguish three different classes of architectural elements:

- processing elements;
- data elements; and
- connecting elements.

This definition defines different mandatory elements for the architecture, and is focused on the design itself. The definition is inspired by the architecture used in hardware and construction, by focusing on views that help to communicate the design. This is similar to the more abstract definition used by Garlan and Shaw two years later [68]:

\(^6\)https://www.youtube.com/watch?v=9hUIxxyE2Ns8
The framework we will adopt is to treat an architecture of a specific system as a collection of computational components — or simply components — together with a description of the interactions between these components — the connectors.

The perception of an architecture as a set of components and connectors has been dominant for a long time. Many architectural styles and patterns have been defined in terms of components and connectors. However, even though architects were using views with components extensively, some aspects were missing for being able to understand the design and the system. The rationale or reasoning behind the decisions was often forgotten, and lacked a clear representation in the architecture. Forgetting the rationale of decisions caused design erosion [75], making it harder to maintain software systems. Also, as agile development practices were adopted more often, software architecture needed to be more lightweight, to be able to change with the changing business needs. Decisions needed to be deferred to later in the product life-cycle, without missing the rationale behind the decisions. Bosch [27] explicitly mentions the design decision as the main element of software architecture:

The key difference from traditional approaches is that we do not view a software architecture as a set of components and connectors, but rather as the composition of a set of architectural design decisions.

In this thesis, the definition of software architecture as a set of architectural design decisions is adopted. In the next section, this concept is explored in more depth.

1.4 Architectural Design Decisions

With the increasing interest in the decisions as first-class entities for architecture design, software architecture entered a new phase [27]. As the component and connector view was closely connected to the implementation of the design (the result), the design decision view was connected to the decision process (how we reach the result) [38]. For the domain of architectural design decisions, there are several terms that are used that need a proper introduction as they are used throughout this thesis. A Design Decision is a decision in the solution space that directly influences the design of a system [96]. Architectural Design Decisions can be seen as a subset of Design Decisions, that specifically concern the software architecture of a system. An Architectural Design Decision can for example concern the adoption of an architectural style [68] or the inclusion of a specific component in the system [188]. There is a thin line between Design Decisions and Architectural Design Decisions, which follows the discussion on what architecture is and what not [65]. This thesis does not intend to provide answers in this discussion. The context for this thesis is software architecture, so if not mentioned otherwise, Design Decisions are considered Architectural Design Decisions.

One of the main benefits of using Design Decisions as first-class entities is that the architecture design can be split into smaller, easier to comprehend pieces, that
1.4. Architectural Design Decisions

together form a whole architecture. Unlike dividing the system in functional components, Design Decisions can address cross-cutting concerns like performance or maintainability, affecting multiple elements of the system. Compared to traditional architectural documentation, they are easier to comprehend as they are smaller and focus on one decision topic. A Design Decision can consist of text and fragments of existing (architectural) views, as part of artifacts. Van Heesch et al. [85] state that research on Architectural Design Decisions focuses on decision templates, decision models, and annotations. Kruchten introduced a classification of Architectural Design Decisions [114], while the GRIFFIN project placed the decision process in a broader context by introducing the concept of architectural knowledge [23], which includes aspects like concerns and architectural design. To visualize the concepts used in this thesis, Figure 1.2 shows a domain model for Architectural Design Decisions. This domain model describes the main elements around Architectural Design Decisions and their relationships.

![Domain Model for Architectural Design Decisions](image)

**Figure 1.2:** Domain model for Architectural Design Decisions.

- **Concern.** A concern is an interest to the systems development, its operation or any other aspect that is critical or otherwise important to one or more stakeholders. The Concern can arise from different sources, like business needs, specific requirements, the development organization or previously made Decisions.

- **Decision Topic.** A Decision Topic is a specific question that needs to be answered to progress with the design. It addresses a specific Concern.
Chapter 1. Introduction

• **Alternative.** To address a Decision Topic, one or more potential Alternatives can be proposed. The alternatives can be reflected in one or more Artifacts, can contain Rationale for the decision, and can be ordered by several Rankings. When an Alternative is chosen as the solution for a Decision Topic, it becomes a Decision.

• **Decision.** One Alternative can be chosen to implement for a Decision Topic, this is the Decision. The selection of the right Alternative to a Decision can be based on one or more Rankings.

• **Artifact.** An artifact is a general term for anything used in the design or implementation of a system. This can be a file (e.g. an architecture or requirement description), a model (e.g. UML models) or written code.

• **Rationale.** The Rationale of an Alternative describes, with text or supporting views, the reasoning behind the Alternative; why would this be a good or bad choice?

• **Ranking.** A Ranking is a way to order Alternatives. This can be either explicit (e.g. an ordered list of alternatives based on the performance) or implicit as part of the decision process.

Several domain models for architectural knowledge exist [22, 172], and attempts show that these models can be mapped on each other [131]. The model used in this thesis is derived from the work we did in the GRIFFIN project [190]. It contains the main elements from the ‘Decision’ package we developed [61] in the context of this project. The model described in Figure 1.2 is used in the chapters of this thesis with a different focus. In Chapter 3, the decision process is analyzed, connecting the Rationale and Alternatives to the Decisions (see Figure 3.3). In Chapter 4, the domain model is used as basis for the experiment with the developed tooling. This model (see Figure 4.7) adds details to the Concerns, and connects the Decision element to documentation where the knowledge is further detailed. In Chapter 8 the domain model is used when searching for design decisions in the history of source code. Here, the focus is on identifying the three main elements: the Alternatives, the Rationale, and the Decision.

Architectural Knowledge like presented in the domain model comes in three different forms: tacit, documented and formal. Tacit knowledge [144] exists as thoughts in human brains, and is accessible by that person only. Documented knowledge helps to communicate, as it can be read by multiple people. However, the interpretation of the documentation can be different between people. Even the person who wrote the work can have problems understanding the text later as tacit assumptions that base the text can be forgotten. Formal knowledge (e.g. metadata or models) can help to increase the understandability and consistency, while it can also be processed by software. The research in this thesis is conducted on all types of knowledge for the software architecture domain. Architectural Knowledge [22] consists of the design, as well as the context, the underlying assumptions and the Architectural Decisions [119]. All this knowledge is essential for understanding the architecture and the architecture decisions are the elements that tie all the knowledge together. In this thesis, the main focus is on architecture decisions, either tacit, documented or formal.
1.5 Research Questions

The field of software architecture clearly acknowledges the importance of architectural knowledge [43, 60], and specifically architectural decisions [96, 178] in order to address knowledge vaporisation [177]. Knowledge vaporisation is the problem that knowledge about the (architecture) design is not available when needed, typically during the evolution of the system. There are several reasons why knowledge vaporisation occurs. First, it is hard and time-consuming to capture knowledge in artifacts. Second, it is difficult to maintain this knowledge when the system evolves. Knowledge vaporisation affects the cost of maintenance and evolution, increases the chances on errors and slows down the development. Some solutions to address knowledge vaporisation have been proposed by academia [43, 60, 96, 178]. However, these ideas do not seem to find large-scale adoption in industry [197]. This thesis explores industry needs on managing architectural knowledge and shows how these needs can be supported in order to address knowledge vaporization.

First, we investigate the industrial needs for managing architectural knowledge, where we focus on architectural decisions. We explore user scenarios where architecture decisions are created and used, based on interviews conducted with industry partners. The first research question addresses the exploration of industrial needs:

![Figure 1.3: Research Questions](image)

**RQ1: What are the industry needs for managing architecture decisions?**

From the research done on this first research question, the scenarios have been divided in 27 use cases for working with architectural knowledge. These use cases involve (multiple) themes of knowledge vaporisation. The use of **artifacts** to preserve architectural knowledge was subjected in 15 use cases. Elements from the decision **process** came back in 9 of the use cases while **reuse** of decisions was pointed out in 10 of the use cases. The remaining three research questions subject these themes, as shown in Figure 1.3.
The scenarios from the initial research showed that it is hard to preserve and manage architectural decisions in typically used artifacts like architectural documents. There is a great availability of tools that support the documentation of architecture decisions [4]. However, the adoption in practice of tools is very low [197] and practical, adequate tools are uncommon [38]. It is often unclear what to capture and how to capture it [38]. We have investigated the research field of rationale management [35, 54] to see how they cope with preserving tacit knowledge in artifacts. We explored how this can be applied in software architecture research. In addition, we investigated what kind of tooling decision-makers need to preserve their tacit knowledge. Therefore, the second research question of this thesis is:

**RQ2: How can tacit knowledge about architectural decisions be preserved for later use?**

The research results from the second research question help us to understand how tacit knowledge can be preserved in artifacts. Architecture artifacts are the result of a decision process. This is rarely a structured or standardized process. Many stakeholders of the system influence the decisions made. In this process, there is often unawareness that decisions are made [197], and there is a lack of motivation or incentive to capture them [38]. The quality of this decision process is hard to assess. However, there are many factors that influence the decision process. We investigated what the main elements are that influence the architecture decision process. Therefore, the third research question of this thesis is:

**RQ3: How does the architecture decision process influence the decision results?**

In our work on the decision process, we saw that making decisions fast to cope with the speed of the changing market is essential for businesses. Reuse of decisions that are made in the past will help to increase this decision speed. The need for data to base decisions on is increasing rapidly [28]. For architectural decisions, typically reused artifacts are architectural styles and patterns. Specific design decisions, for example which component to use, are very project-specific. Reuse of these architectural decisions between projects and systems [4] is very difficult. However, sharing data is becoming more common, as you can see in the open source community. As part of the research in this thesis, we searched for data on architectural decisions in open source projects. We investigate if it is possible to gather data on the occurrence of specific decisions in the past, to assist decision makers in the present. Therefore, the fourth and last research question for this thesis is:

**RQ4: How can architecture decision makers reuse decision data?**

The four research questions are addressed this thesis. The mapping of the research questions to the chapters of this thesis is described in Section 1.7. A summary of the results is given in the conclusions in Chapter 9.
1.6 Research Methodology

Research in software engineering is a combination of data-driven and social sciences research. On the one hand it is possible to prove aspects of software development (e.g. deduce the correctness or efficiency of algorithms), while on the other hand creating software is a social activity where determinism is not possible as validation. Opinions, discussions and emotions highly influence the effectiveness of a software project.

Hevner et al. [86] describe two paradigms for research in information systems. These paradigms are similarly relevant for software engineering research. First, they describe the behavioral-science perspective. Hevner et al. state this ‘seeks to develop and justify theories (i.e., principles and laws) that explain or predict organizational and human phenomena surrounding the analysis, design, implementation, management, and use of information systems’ [86]. Second, they describe the design-science paradigm [52], where research is seen as a problem-solving activity. This is in line with Shaw, [163] who describes software engineering research by defining a specific goal:

Generally speaking, software engineering researchers seek better ways to develop and evaluate software. They are motivated by practical problems, and key objectives of the research are often quality, cost, and timeliness of software products.

With this definition, the focus is on the practical usefulness of the research (utility), not on the development of theories (discover truth). Hevner et al. [86] argue that these two paradigms can be seen as two sides of the same coin, both aiming to discover useful IT artifacts. Latour [124] showed that research is a social construct, where social interactions and research artifacts together construct meaningful theories. Hevner et al. [86] identify four types of IT artifacts that are used in software engineering research:

- **constructs**: vocabulary and symbols
- **models**: abstractions and representations
- **methods**: algorithms and practices
- **instantiations**: implemented and prototype systems

This thesis reports on research that developed several of these types of IT artifacts. We iterated the build-and-evaluate loop often several times. When we learned from our initial results, we changed the theory and the artifacts and once again evaluated the results. This way of working is similar to the validated learning approach advocated by Eric Ries [156] for new product development. While the IT artifact can be used as a goal for design-science research, one other aspect is important too: the **evaluation** of the artifact [86]. The contribution of this thesis will be addressed by describing the artifact and the evaluation of the conducted research in the conclusions in Chapter 9.
1.6.1 Research Techniques

In software engineering research, many research techniques are used. In the following list, the used research techniques in this thesis are summarized.

- **Interview** As software engineering is a social activity, it is important to get a solid idea of the opinions of participants. Structured interviews [161] are a way to gather data on existing problems and chosen solutions. Two chapters in this thesis are based on interviews as research methodology: Chapter 2 and 7.

- **On-line Survey** A more data-driven way to acquire knowledge from people that are involved in software projects is by conducting an on-line survey [154]. One of the chapters in this thesis is based on the results of a conducted survey: Chapter 6. The survey was codified, and statistical analysis has been used to validate the research assumptions.

- **Participant Observation** In order to identify problems in software projects, we conducted research on several case studies as participant researchers [161]. In these cases, the researchers were part of a project team, and comparative multi-case analysis methodology [55] was used to generalize over the cases to an abstract model. This methodology was used in Chapter 5.

- **Proof of Concept** In order to validate that a proposed methodology or tool is feasible, a prototype, or proof of concept, is used [74]. With this technique, the developed prototype shows the feasibility of the approach, and demonstrates a solution that is generalizable to usage in practice. In Chapter 3, an example case is modeled in a proof of concept tool Archium [101] to show the feasibility of the approach. In Chapter 4, an example implementation is used in combination with a controlled experiment to show the feasibility of the approach. Last, in Chapter 8, the possibility of mining decisions and getting access to rationale is shown with an example implementation.

- **Controlled Experiment** With the quasi-controlled experiment [102] used in Chapter 4, subjects were instructed to use a developed tool in order to validate hypotheses about expected efficiency.

- **A/B Experiment** As an experimental technique, the validation of work presented in Chapter 8 was done by an A/B experiment [47]. In this experiment, the subjects were unaware they were part of the research, and two groups of subjects were used to validate additional assumptions.

1.7 Structure of this Thesis

The chapters of this thesis consist of published material. Apart from minor changes (e.g., references or small corrections), these chapters are the same as the publications. In two cases, the chapter is a combination of two publications (Chapter 2 and Chapter 8). The chapters are not ordered chronologically; the work is ordered
1.8. Publications

The research presented in this thesis has been published in international conferences, as book chapters as well as in international journals. The chapters consist of these publications, sometimes with minor changes to improve the readability. The research in this thesis is performed by Jan Salvador van der Ven, which in most cases is the first author of the publication. For all the publications, at least one senior researcher co-authored the paper, where they helped primarily with feedback on the structure and contents.

In Chapter 2, we identified the industrial needs for architectural knowledge by conducting interviews at four companies that were involved in the Griffin project. The interviews were conducted by the involved researchers (the author of this thesis conducted the interviews at one of the partners). Together with Anton Jansen, I constructed the use case model presented in this work, based on the interview results. The use case model creates the context for this thesis as it gives an overview of the needs for architectural knowledge management. I also did the validation of a set of use cases in cooperation with ASTRON, as described in the chapter. The work in Chapter 2 is based on the following publications:


### Table 1.2: Structure of this Thesis

<table>
<thead>
<tr>
<th>Research Question</th>
<th>Chapter</th>
<th>Research Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1</td>
<td>Chapter 2</td>
<td>Interview</td>
</tr>
<tr>
<td>RQ2</td>
<td>Chapter 3</td>
<td>Proof of Concept</td>
</tr>
<tr>
<td></td>
<td>Chapter 4</td>
<td>Proof of Concept &amp; Controlled Experiment</td>
</tr>
<tr>
<td>RQ3</td>
<td>Chapter 5</td>
<td>Participant Observation</td>
</tr>
<tr>
<td></td>
<td>Chapter 6</td>
<td>On-line Survey</td>
</tr>
<tr>
<td></td>
<td>Chapter 7</td>
<td>Interview</td>
</tr>
<tr>
<td>RQ4</td>
<td>Chapter 8</td>
<td>Proof of Concept &amp; A/B Experiment</td>
</tr>
</tbody>
</table>

by the topics of the publication to increase the readability of this thesis. The chapters follow the order of the research questions.

Chapter 1 introduces the research questions. Chapter 9 summarizes the contribution of this thesis and presents future work. The contribution of the other chapters to the research questions is shown in Table 1.2.
In Chapter 3, the relationship between software architecture and the rationale management field is explored. The concept of the explicit architectural design decision is introduced to bridge the gap between these worlds and enable better understanding of architectural decisions. This work is written in cooperation with Anton Jansen. My contribution to this work is the comparison of the processes and the validation of the ideas with the example. The work on Archium was done by Anton. Chapter 3 is based on the following publication:


Chapter 4 has been previously published with Anton Jansen as first author, as he did the most of the writing of this manuscript. I started this research with the initial idea on annotating architectural decisions in documentation. I created the first version of the Word plug-in. As third and last author of this research, I contributed with the initial idea, and the structuring and writing the manuscript. Chapter 4 is based on:


The work of Chapter 5 describes three dimensions that can be used to classify the nature of architectural decisions: the architect, the artifacts and the periodicity. The dimensions are distilled by me from the industrial cases. I participated in four of the five companies as participant researcher, the data of the last company came from the coauthor Jan Bosch. The identification of the problems in the cases was done by me, while Jan Bosch assisted me with proof-reading of the work and feedback on the content. Chapter 5 is based on:


In Chapter 6, we assess beliefs around software architecture, based on the results of an on-line survey. I created and distributed the survey. I summarized the beliefs and conducted the coding and analyses on the results of the questionnaire. The work from Chapter 6 is based on:


Similar to Chapter 3, Chapter 7 explores the relationship between software architecture and another field, in this case the field of lean startup. It describes how the concept of design decisions and pivots show similarities. The work is based on
interviews with founders and experienced architects at five different companies. I conducted these interviews, created the conceptual model and conducted the analysis of the pivots. Also, I create the comparison of the concepts and developed the guidelines. The work in Chapter 7 is based on:

- Jan Salvador van der Ven and Jan Bosch. “Pivots and Architectural Decisions: Two Sides of the Same Medal? What Architecture Research and Lean Startup can learn from Each Other”. In: *Proceedings of International Conference on Software Engineering Advances (ICSEA 2013)*. 2013, pp. 310–317

Chapter 8 describes how architectural design decisions can be mined from the version management of open source projects. The idea came from me, and I created the software that was used to do the mining, visualization, and validation with the email experiment. I developed the criteria for the programming languages and the assessment of them. I conducted the analysis of the early results with subject matter experts as well as the email experiment. The chapter is based on the extended version of this research, which is currently under review, supplemented with a part from the previously published version. The work in Chapter 8 is based on the following manuscripts:


- Jan Salvador van der Ven and Jan Bosch, "Towards Reusing Decisions my Mining Open Source Repositories". *Submitted to an international Software Engineering Journal*, 2018

Additional publications done by the author of this thesis, which are not included as chapters in this thesis, are:


