Architectural design decisions

Jansen, Antonius Gradus Johannes

CHAPTER 1

INTRODUCTION

1.1 Software engineering

Through the ages, mankind has created systems. New inventions and technologies have allowed the creation of ever more advanced systems. These advancements have had a tremendous impact on mankind for better or worse. For example, on the positive side, people can nowadays fly to the moon, view a world cup football final with over 600 million others, and life expectancy has doubled in the last 500 years. On the negative side, nuclear bombs can now kill millions of people. All of which is possible due to having systems in place that enable mankind to accomplish these amazing feats.

The interdisciplinary field of systems engineering focusses on the development and structure of such complex artificial systems. Different engineering disciplines (e.g. electrical engineering, mechanical engineering) deal with different aspects of these complex artificial systems from different perspectives. As new inventions and technologies arise, new engineering disciplines are created to make use of them. Consequently, system engineering becomes an even broader discipline.

A recent addition to system engineering was formed by computer and software engineering. In the last 50 years, increasingly more systems have started to contain computers. For example, cars now contain well over 80 different computers (e.g. for braking, steering, and navigation) [23], whereas they did not use them in the past. Computers offer unique added value to systems, as they excel in providing flexibility and adaptability to a system. Furthermore, they can perform simple tasks faster, more reliably, and in hostile environments, than human operators. Computers can also be systems in their own right, i.e. a computer system. A distinction in computer systems is often made between the hardware and software. The hardware comprises the physical components (e.g. processor, memory, etc.), which together form a computer. The discipline of computer engineering or hardware engineering, is concerned with this part of a computer system. Software, on the other hand, is a collection of computer programs with associated procedures and documentation, which allow someone to perform some task on the hardware. In the last two

decades, software has become the competitive edge for many system creating organizations. Not in the least part due to mass fabrication of computer hardware, which has made it economical for many applications to adapt the software to the hardware, instead of adapting the hardware.

The discipline of software engineering is concerned with the development, operation, and maintenance of software [146]. Software engineering research is all about making bigger, better, and faster software. Bigger, as in creating ever more complex and bigger software systems to deal with even more complex problems. For example, within Philips the size of the embedded software for televisions doubles every two years [169]. Better, as in creating software that has more quality and addresses problems found in a better way. For example, the operating system Microsoft Windows XP crashes far less often than it predecessor Windows 3.1. Faster, as in reducing the time it takes to develop, operate, and maintain software. For example, nowadays powerful Integrated Developers Environments (IDEs) assist software engineers with integrating third party software, thereby reducing development time [16].

1.2 Software architecture

One sub-discipline within software engineering is concerned with studying software architectures, which is a kind of high-level (abstract) design of the software of one or more systems. Currently, there is no agreement to what exactly software architecture entails. This is evident from the hundreds of different definitions found in both literature and the software architecture community [145]. One popular definition is from [11]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Software architecture are created, evolved, and maintained in a complex environment. The architecture business cycle [11] of figure 1.1 illustrates this. On the left hand side, the figure presents different factors that influence a software architecture through an architect. It is the responsibility of the architect to manage these factors and take care of the architecture of the system. An important factor is formed by requirements, which come from stakeholders and the developing organization.

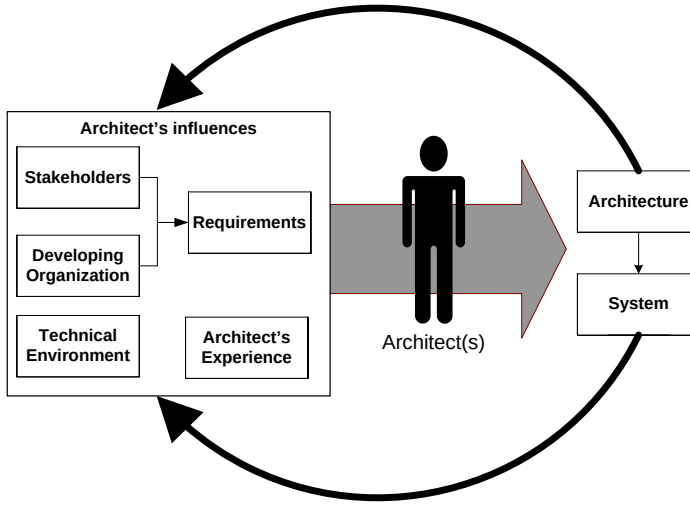

Figure 1.1: The architecture business cycle from [11]

Requirements explicitly state *what* the system is supposed to do. It is the responsibility of the architect to make sure that the software architecture defines *how* this could be achieved.

The architecture business cycle contains a feedback loop, within which the architect's influences themselves are influenced by both the system and architecture. This feedback loop exists, since the perception of the system and the architecture influences the stakeholders. This is illustrated in figure 1.1 by the arcing arrows from the system and architecture back to these influences. Following is a description of how each factor influences the software architecture and vice-versa:

Stakeholders Requirements come from many different people and organizations (e.g. end users, developers, project managers, customers, shareholders, upper management, government, maintainers, and sales people), which have an interest in a system. Each of these *stakeholders* has different concerns that they wish the system to address. It is rather common that these concerns are in conflict with each other.

One particular type of requirements, the Non-Functional Requirements, (NFR) often causes conflicts among stakeholders. NFRs are requirements about the quality of the software. Different kinds of qualities exist, e.g. maintainability, flexibility, security, performance, usability, portability, and scalability. To express a particular quality a design or software system delivers, the term *quality attribute* is used. A NFR therefore defines the level a quality attribute of a design should have. In practice, not all the NFRs of the stakeholders can be satisfied by the quality attributes

of any design, which forces the architect to make trade-offs between them. The feedback of the architect from the architecture to the stakeholders consists of these trade-offs. For example, there is usually a tradeoff between security and usability, as many security measures cause usability problems. The stakeholders will have to agree what level of security is needed and what level of usability is still acceptable.

Developing Organization Besides the organizational goals described in the requirements, software architectures are also influenced by the vision, business strategy, and structure [33] of the developing organization. For example, if an organization consists of five development teams, it is very likely that the architecture will be decomposed in five different parts. In addition, the skills available to a organization typically influence the kind of software architectures considered. An organization might consider reorganizing their structure to better fit the way their architecture is organized.

Technical environment Standard industry practices and techniques that are commonplace in the architect's professional community influence a software architecture. For example, certain technologies are extremely hyped, making architectures that enable the use of such technologies a more favorable choice. Exploring and considering architectural options outside an industry's familiar territory requires a great deal of bravery and professionalism from an architect and is therefore often left aside.

Architect's experience Although positive architectural results in the past are no guarantee for the future, software architects often prefer architecture solutions that have worked for them in the past. The opposite can be said about solutions they have tried, but failed to deliver. Some of these solutions can be generalized and become architectural patterns [25] or styles [138]. Exposure to these architectural patterns influences the solutions an architect will come up with.

Besides this exposure to architectural patterns and (un)successful systems, software architecture education and training also enhances the architect's experience. Consequently this might influence a software architecture, as an architect might want to try out certain learned patterns or techniques.

The architecture business cycle describes the different factors influencing a software architecture. However, it does not describe the different uses of a software architecture. In short, a software architecture is used for the following purposes:

- **Blue-print** The major purpose of a software architecture is to outline a design, i.e. be a blue-print, for the software of a system. With additional effort, this design can be fleshed-out into a detailed design, which in turn can be implemented to create the software for a system.

- **Roadmap** A software architecture allows one to plan ahead the evolution of the software of a system and use it as part of a technology roadmap. This allows a software architect to align the software with a company's mid to long term business strategy, thus improving or maintaining a company's technical advantage in the market place.
- **Communication vehicle** A software architecture description can be used as a communication vehicle, as it enables different stakeholders to communicate about the major decisions made. In this way, a software architecture allows different people to steer and influence the software of a system.
- **Work divider** A software architecture can be used as a work divider, as it decomposes software in smaller parts. This allows software engineers to work, to a certain degree, in parallel on the software.
- **Quality predictor** A software architecture can be used as an early predictor of the quality of a deployed system. This is especially useful in a green field situation, as changing architectural decisions later in the life cycle are at least an order of a magnitude more expensive to perform.

1.2.1 Software architecture description

As there are many different roles a software architecture can fulfil, it will come as no surprise that there are many ways in which software architectures are described. To describe a software architecture, people use different forms of communication or combinations of them. The following forms are commonly used:

- **Natural language** is the most frequently used form, both oral and written, for describing software architectures.
- **Templates** provide molds for describing a software architecture using natural language. It achieves this by pre-describing the elements and relationships of the software architecture that should be documented. In many cases, a good and appropriate template will describe a software architecture more consistently and concisely than natural language.
- **Diagrams** are a very popular form to describe software architectures. This form excels in communicating complex relationships between concepts, which is very handy for the different abstractions used in software architecture.
- **Pictures** in the form of photos or illustrations are used to explain important concepts using metaphors.
- **Formal language** is a form in which the software architecture is formally described, e.g. using a meta-model. This model describes the concepts, their relationships, and semantics for describing a software architecture. For exam-

ple, in Model Driven Architecture (MDA) [90] the objective is to define such a formal model in so much detail that a software implementation could be (semi-) automatically derived from it.

The IEEE-ANSI standard 1471 [70] presents a recommended practice for describing software architectures. It is based on the work of documentation approaches like the Siemens four view [67] and Kruchten's 4+1 views [92]. These approaches use a combination of natural language, templates, and diagrams for describing a software architecture. The IEEE-ANSI standard is rather general and abstract, as it provides a conceptual framework for documentation approaches. Furthermore, it does not present the details of how software architectures should be described. The Views and Beyond (V&B) approach of the Software Engineering Institute (SEI) [29], as well as other documentation approaches, try to fill in these details. To guide the description of the architecture, documentation approaches use the concept of a view. A view is a representation of a whole system from the perspective of a related set of concerns [70]. In this sense, views define conceptual perspectives for looking at an architecture. The aim of a documentation approach is therefore to define interesting views of system elements and their relationships, which together describe a software architecture. For example, Kruchten's 4+1 approach [92] describes four views: logical, process, physical, and development view. They are combined together with the "+1" use case view, which decribes the use cases and scenarios supported by the architecture.

The IEEE-ANSI standard 1471 takes the concept of a view one step further with the introduction of the concept of a viewtype. A view can be classified to be of a certain viewtype. The Views & Beyond approach [29] argues that for describing a software architecture, one should provide at least one view from each viewtype. The viewtypes they distinguish are the following:

- **Module viewtype** describes the structure of the software in terms of its implementation units.
- **Component & Connector viewtype** describes the run-time principal processing units of the executing system.
- **Allocation viewtype** describes how the software relates to non-software structures in the environment.

A different type of approach towards describing software architectures are Architecture Description Languages (ADLs) [107]. They use a formal language for their description of an architecture. Compared to the documentation approach most of the ADLs focus on the Component & Connector viewtype. Diagrams are used to

visualize and sometimes offer the ability to graphically edit the model. Some ADLs (e.g. UniCon [137]) offer a code-generation feature, which allows one to generate a stub framework based on the architecture model described in the ADL. The stub framework can be used as a basis for the implementation of the system.

1.3 Architectural knowledge

A recent development in software architecture research is the notion of Architectural Knowledge (AK). AK encompasses the knowledge involved with software architectures. Architectural knowledge is vital for the architecting process, as it improves the quality of this process and of the architecture itself [44]. What exactly the notion of AK entails is still a topic of ongoing research and debate [38]. Some define AK as $AK = design\ decisions + design$ [95], others as $AK = drivers, decisions, analysis$ [62], and some take a broader perspective by including processes and people aspects [39]. Most people seem to agree that at least one part of AK is about the rationale, assumptions, and context of decisions that lead to a particular design.

In the rest of this section, we present some of the different dimensions AK has and explain the notion of AK in more detail. To exemplify this notion, the section concludes with a description of a domain in which architectural knowledge is very visible: software product lines.

1.3.1 Dimensions of architectural knowledge

Knowledge and architectural knowledge in particular have many different dimensions. Each dimension describes a different aspect of architectural knowledge. Three of these dimensions are presented here: the type of knowledge, the producer-consumer, and the knowledge management strategy.

The first dimension is the type of knowledge. In knowledge management, a distinction is often made between two types of knowledge: implicit and explicit knowledge [112]. Implicit or tacit knowledge is knowledge residing in people's heads, whereas explicit knowledge is knowledge which has been codified in some form (e.g. a document, or a model). Two forms of explicit knowledge can be discerned: documented and formal knowledge. Documented knowledge is explicit knowledge which is expressed using natural language in documents. Formal knowledge is explicit knowledge codified using a formal language or model of which the exact semantics are defined. For example, the source code of a software system is formalized knowledge. Figure 1.2 presents these different knowledge types.

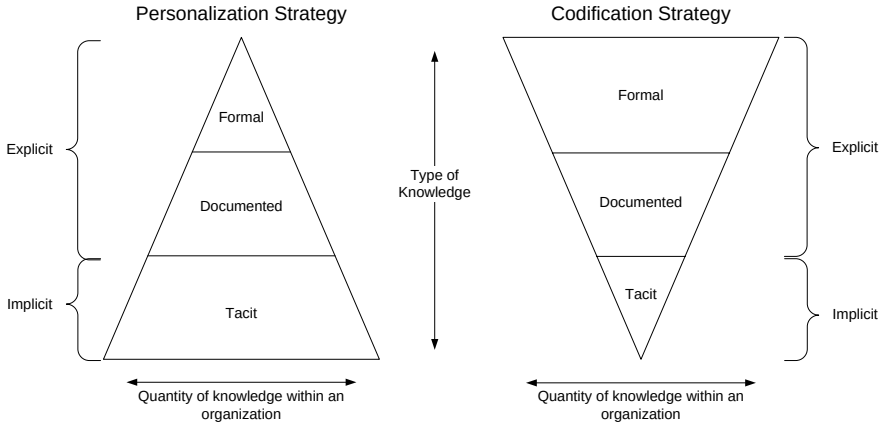

Figure 1.2: Pyramid of knowledge types and the associated knowledge management perspective

Apart from the type of knowledge, figure 1.2 also visualizes another dimension of AK: the organizational knowledge management strategy, which is the way organizations manage their knowledge. Organizations can employ two distinct strategies for managing their knowledge: a *personalization* or *codification* strategy [7, 63]. In a personalization knowledge management strategy, an organization leaves most of the knowledge tacit. Only the knowledge about who knows what is made explicit. Knowledge is transferred in this strategy directly from people to people through socialization [112]. Figure 1.2 illustrates this on the left hand side, as the pyramid of knowledge is broad for the tacit knowledge and becomes smaller for the documented and formal knowledge.

An organization with a codification strategy, on the other hand, carefully codifies and stores explicit knowledge in documents and databases. Such organizations try to reuse this explicit knowledge as much as possible. In figure 1.2, this is illustrated on the right hand side. The explicit (i.e. formal and documented) knowledge is relatively large compared to the tacit knowledge in an organization. The choice for either strategy depends on the economic model used and the way human resources are managed within an organization. For more information about these aspects, we refer to [63].

The last and third dimension of AK is the consumer and producer dimension [98]. Figure 1.3 visualizes this dimension, which gives insight into how people deal with architectural knowledge. Two actors are shown: the consumer and producer. In the architecting process, a person could play both roles simultaneously. The cubes in the figure represent architectural knowledge, which can be of any type (i.e. be tacit, documented, or formal). The figure illustrates the creation of architectural

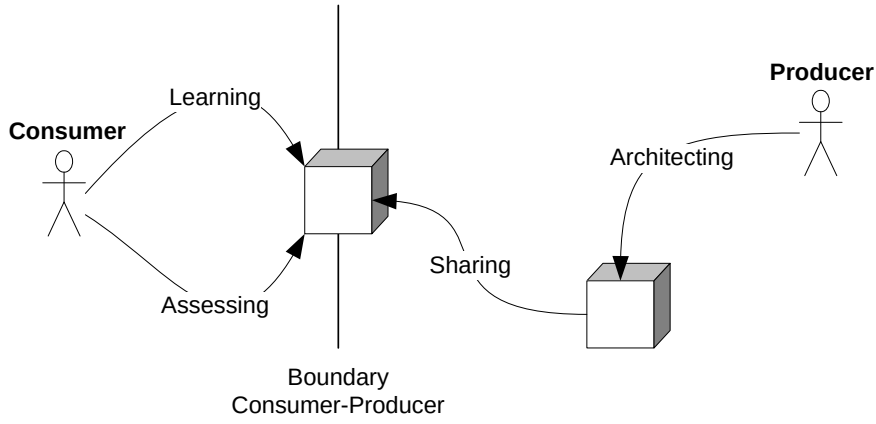

Figure 1.3: Producer-consumer perspective on Architectural Knowledge from [98]

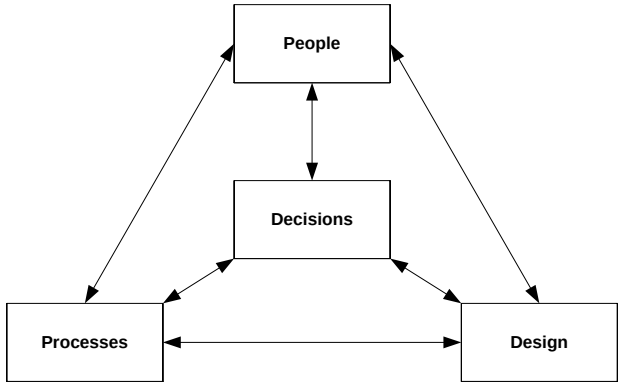

Figure 1.4: The four parts making up the Griffin Core Model [39] of Architectural Knowledge

knowledge through architecting. Architectural knowledge is shared by making the architectural knowledge available to a consumer. The consumer can learn the architectural knowledge and assess it (e.g. in a review). The last action allows the consumer to provide feedback on the architectural knowledge to the producer.

1.3.2 Defining architectural knowledge

As pointed out in the introduction of this section, the notion of AK and what it entails is still subject of ongoing research [38]. This section presents one opinion on this rather broad concept of AK. To get a better grip on this concept, we have developed a meta-model [39] in the Griffin project [56]. This meta-model, or core

model as we call it, describes the concept of architectural knowledge. Figure 1.4 presents the four distinct parts, which together make up this concept. In short, these four parts are the following:

Processes Software architectures influence the processes in an organization and vice-versa [33, 34]. For example, knowledge of the software architecture is instrumental in creating working units for the division of work in an organization [117]. Knowledge about the processes supported by a system is therefore important architectural knowledge.
People Many different stakeholders are involved in architecting. Balancing their concerns (e.g. expressed in requirements) and resolving conflicts is an important aspect of architecting. Therefore, knowledge about the people involved, their concerns and relationships is important architectural knowledge.
Decisions To come to an architecture design, decisions need to be made. This includes decisions about which of the stakeholders concerns are deemed important enough to be addressed in the architecture design. In addition, it also includes decisions about the architecture design itself, which often require a difficult balancing act between the aforementioned concerns. Knowledge of these decisions is crucial, as they form the basic underpinning of the architecture design.
Design Knowledge about the software architecture design forms the cornerstone of architectural knowledge. Central is the notion of the architecture design, which can be expressed in one or more languages (both natural and formal). Using such a language, an architecture design can be captured in one or more artifacts[1] (e.g. word documents, powerpoint presentations, etc.). Example of these languages to express an architecture design include Architecture Description Languages (ADLs [107]).

The Griffin core model [39] describes AK from a conceptual perspective, i.e. it describes the major concepts and their relationships that make up architectural knowledge. Underlying this core model for architectural knowledge is the vision to see architecting as a decision making process in which AK is consumed and produced. In this process, concerns from various stakeholders are turned into architectural solutions. The place architectural decisions have in this process is illustrated in figure 1.5. The figure is based on an earlier model from [98], but extended with the problem space, abstraction levels, and the place of architectural design decisions. This figure presents the following three significant dimensions of decision making:

- First, there is the distinction between problem and solution space. The problem space is the domain containing all the problems of the environment that

[1] Remark that all four parts (design, processes, people, decisions) can be expressed in artifacts

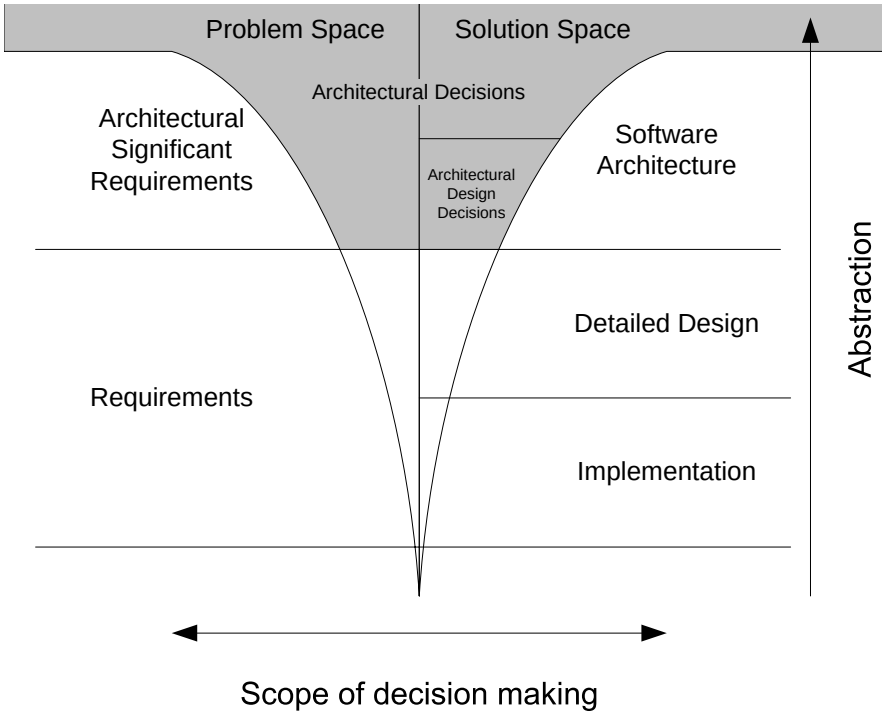

Figure 1.5: The funnel of decision making

a system could address. The solution space is the domain containing all possible system solutions [35, 143].

- The second dimension is formed by the level of abstraction. In both problem and solution space, decisions can be made at different levels of abstraction. These levels are visualized on the left and the right side of the figure.
- Third and last, there is the dimension of the scope of decision making, which is non-orthogonal to the first two dimensions. Depending on the abstraction level, the scope of decision making becomes smaller and smaller, i.e. is funneled. The top of the funnel is open ended, i.e. as broad in scope as the problem and solution spaces themselves. If this funneling does not reach the end-point, the decision making process does not converge. Hence, the system development does not lead to a system implementation.

Architectural decisions are located in both problem and solution space and made on a particular architecture abstraction level. Generally, the scope of decision making

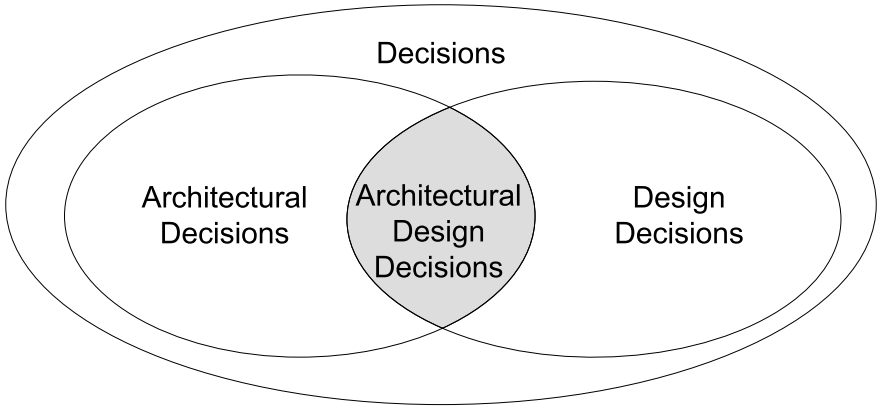

Figure 1.6: Type of decisions and their relationships

is very broad and no clear boundaries are (initially) defined. A first iteration for a new system (i.e. a green field situation) starts at the top of this funnel and proceeds downward in abstraction, thereby increasingly reducing the scope for the system. Later iterations (hopefully) restart with a smaller funnel, as most of the scope is set by earlier iterations. Hence, less time will be needed for the higher abstraction levels.

Figure 1.5 presents a rather idealistic perspective on the decision making process during software development, as it leaves out two common problems with this process. First, aligning the decisions inside the funnel is no easy task. It is (sadly) rather common for detailed design and implementation decisions to be misaligned with the architectural decisions. For example, a weak implementation of a strong security strategy is such a misalignment. Hence, the need for verification of a system and its architecture to address such issues in future iterations. Second, very often, seemingly "small" decisions turn out to be architectural decisions in hindsight. For example, the threading strategy to use can turn out to be a major architectural decision instead of an insignificant and detailed one. This is another reason why iterative software development is a good idea, as it allows reconsideration of such decisions and their impact on the overall design.

There exists a distinction between architectural decisions and design decisions. Architectural design decisions are both architectural decisions and design decisions. Figure 1.6 illustrates the commonalities and differences of these three concepts in a Venn diagram. In short, the commonalities and differences for each concept are:

- **Architectural Decisions** are decisions that influence the software architecture.

For example, decisions that address architecturally significant requirements are per definition architectural decisions. Architectural decisions that are *not* architectural design decisions are those decisions that affect the software architecture in an indirect way. These architectural decisions are mostly about people and processes. For example, the choice to use a particular development process is an architectural decision that might indirectly influence the resulting architecture.

- **Design Decisions** are decisions in the solution space that directly influence the design of a system. Not all design decisions are architectural decisions. For example, detailed design decisions are per definition not architectural decisions.
- **Architectural Design Decisions** are decisions in the solution space that directly influence the design of the software architecture. For example, choosing a particular architectural style [138] is an architectural design decision.

1.3.3 Design decisions and variability management

To get a better understanding of what the concept of a design decision entails, we examine a particular domain in which this concept is very visible. Design decisions play a crucial rule in the design and use of Software Product Lines (SPLs) [19]. The aim of an SPL is to exploit the commonalities among different products. To this end, an SPL defines a product line architecture in which different elements can be reused from a common asset base. To derive a specific product, decisions need to be made among the alternatives the reusable asset base provides. Some of these decisions are design decisions or even architectural design decisions, which makes the domain of SPLs interesting for studying (architectural) design decisions.

In the domain of SPLs, modeling alternatives and decisions is called variability modeling [140]. This models how different alternatives in a product line affect the functionality and quality of a product. Achieving this for design decisions in general is something we would like to achieve. The common asset base of an SPL provides several alternatives for a decision, which are called variants in variability modeling. To denote at which points in the SPL these kind of decisions should be made, a so-called variation point exists. Consequently, variability modeling makes the (architectural) design decisions of an SPL explicit. For example, a variability model for an SPL of car engine controllers will contain different variants for leisure and sports cars. A variation point might be the amount of fuel injected into the engine or the timing of this action.

In a sense, deriving a product from an SPL is comparable to navigating through the decision making funnel of figure 1.5 with the variability model providing a mapping between the problem and the solution space. The variability model (i.e.

the map) describes which decisions should be made (i.e. the variation points) and what kind of alternatives (i.e. variants) are available. Typically, when deriving a product from an SPL a large amount of these decisions have to be taken, due the vast amount of variation points in most SPLs. For most products, common sets of these decisions exist in the form of so-called 'features'. This allows one to make different derivations in an SPL based on the differences in terms of features between products, without having to make a manual decision for each variation point individually. In other words, features form a way to combine many decisions to more abstract (architectural) design decisions. For example, for the car engine SPL, a feature might be that a driver is able to switch to different engine settings.

From an architectural knowledge perspective, variability modeling is concerned with modeling only one particular class of design decisions; those which are *explicitly* being postponed. This allows an SPL to make these choices later in the life-cycle, i.e. when deriving a specific product from the SPL. It also makes SPLs a good starting point to investigate design decisions, as variability models make these delayed decisions explicit and visible. More information on the relationship between variability modeling and design decisions can be found in [141].

1.4 Problem statement

One of the major problems with architectural knowledge is architectural knowledge vaporization. In this process, an organization loses its tacit architectural knowledge. This can happen due to a number of reasons (see also [64]):

- The availability of people for an organization changes over time. For example, employees start working for a competitor or retire.
- Fast changes in a system's enviroment, both in business and technology, as it becomes harder and harder to relate the AK to a system's originally (intended) environment, makes recalling this AK itself difficult.
- Architects consume or produce AK without realizing this fact. Hence, they are unaware of the need to make the AK explicit.
- Making AK explicit is often deferred to a later moment in the life cycle. However, due to the forgetful nature of humans such AK is easily lost. For example, in a survey 74.2% of the respondents indicated that they forget half or more of their own design decisions over time [150].

- The effort it takes to make AK explicit is bigger than the expected benefits. Hence, the organization takes the AK vaporization for granted, i.e. it uses a personalization knowledge management strategy (see section 1.3.1).
- Architects don't know how to make AK explicit.

Losing AK (and thereby architectural decisions) is most critical, as it contributes to a number of problems the software industry is struggling with [79]:

- **Expensive system evolution**. Systems need to evolve to keep up with the changing world surrounding it. The requirements a system is expected/required to fulfill change and consequently the system needs to change. Typically, starting from scratch is not an option. Instead, an existing system is often evolved to meet the changed requirements. To evolve a system, new architectural decisions need to be taken. If, however, due to knowledge vaporization, the architectural knowledge is lacking, then adding, removing, or changing architectural decisions becomes highly problematic. Architects may violate, override, or neglect to remove existing decisions, as they might be *unaware* of them. This issue, which is also known as *architectural erosion* [72, 119, 168], results in high evolution costs.
- **Lack of stakeholder communication**. Stakeholders usually come from different backgrounds and have different concerns that the architecture must address. If architectural decisions are not shared among the stakeholders, it is difficult to perform tradeoffs, resolve conflicts, and set common goals, as the reasons behind the architecture are not clear to everyone. Knowledge vaporization can lead to architectural decisions not being shared, as an organization becomes no longer aware of all of them.
- **Limited reusability**. Knowledge vaporization is a direct threat to effectively reusing architectural artifacts. In the first place, an organization needs architectural knowledge to become aware of suitable reuse opportunities. Second, to prevent remaking past mistakes, knowledge is needed of the decision process leading up to the artifact. Third, architectural knowledge is required of the assumptions of these decisions to determine if the artifact is suitable for the situation at hand.

1.5 Research questions

The previous section presented the problems architectural knowledge vaporization contributes to. However, a solution was not presented. To find a (partial) solution to this problem is the central theme of this thesis. In other words, the overall research question that motivates this thesis is:

How to reduce the vaporization of architectural knowledge?

The concept of architectural knowledge is rather broad (see section 1.3 on page 7). Therefore, this thesis focusses on a specific part of architectural knowledge. In the past, the decision part of architectural knowledge did not receive much attention of the software architecture community. The community primarily concentrated on architectural design and architectural processes, leaving out the decision and people aspects (see section 1.3.2 on page 9). This thesis therefore concentrates on one of the less investigated types of architectural knowledge; the decision type. Thus, the main research question this thesis tries to answer is:

RQ: How to reduce the vaporization of architectural decisions?

The answer to this research question depends on the knowledge management strategy (see section 1.3.1) of an organization. Making knowledge explicit is a good solution for organizations using a codification strategy, whereas knowledge redundancy is a good solution for the ones using a personalization strategy. In this thesis, the focus is on the codification strategy, i.e. making the knowledge of architectural decisions explicit.

Before we can determine how to eliminate architectural decision vaporization, we need to understand what decisions are. For this, we focus on one type of architectural decisions: architectural design decisions (see section 1.3.2). To this end, we formulated the following research question:

RQ-1: What are architectural design decisions?

Once we know what features and architectural decisions are, we would like to have the means to make them explicit to prevent knowledge vaporization from taking place. Models of these concepts can provide these means. Thus, the following research is posed:

RQ-2: How can we model architectural design decisions?

To improve our understanding for answering this research question, we start with a domain in which decisions are very visible. As explained in section 1.3.3, Software Product Lines (SPLs) explicitly model decisions using variability modeling. A special class of these decisions are formed by features, which abstract from many small decisions. This makes features comparable in complexity to architectural decisions and an ideal candidate for (partially) understanding what architectural decisions are. To this end, we formulated the following more detailed research question:

RQ-2.1: How can we model features in an SPL?

A software architecture contains multiple architectural design decisions, which have complex dependencies between each other. To describe combinations of architectural design decisions, a model for these decisions needs a way to deal with these dependencies. Thus leading to the following research question:

RQ-3: How to deal with architectural design decision dependencies?

As we first model features to learn more about how to model architectural design decision in general, we also have a similar research question for features:

RQ-3.1: How to deal with feature interactions?

Architectural decisions are consumed and produced (see section 1.3.1) in the architecting process, which is part of the architecture business life cycle (see section 1.2). It is in this process that architectural decisions vaporize, whereas they could be of invaluable use. This raises the following two research questions:

RQ-4: What is the added value of explicit architectural decisions in the architecting process?

RQ-5: How is making architectural decisions part of the architecting process?

Related to the aforementioned research questions are questions about how tools could support the production and consumption of architectural decisions in the architecting process. To investigate this, we formulated the following two research questions:

RQ-6: What are existing automated support tools for managing architectural design decisions and what do they support?

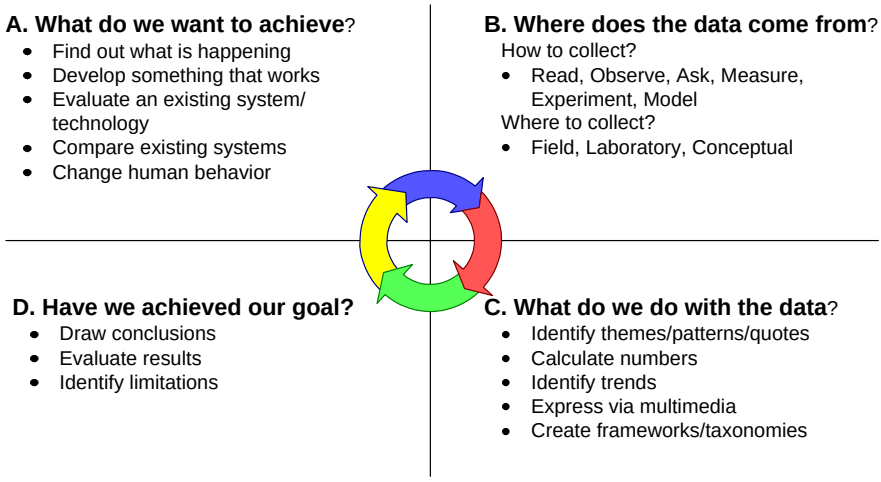

Figure 1.7: The research process framework by [69]

RQ-7: How to provide good tool support for architectural decisions?

As noted in section 1.4, one of the reasons for AK vaporization (and therefore architectural decisions) is the habit to defer the task of making AK explicit. If this happens, one might like to recover architectural decisions at a later moment. How to perform such a recovery is the subject of the last research question of this thesis:

RQ-8: How can we recover architectural design decisions?

1.6 Research methods

1.6.1 Introduction

Software engineering and the rest of computing science lacks well defined and known research processes, as found in disciplines like physics, biology, and medicine [55, 136]. This is partly due to the inmaturity of the research field and the abstract nature of the subject of software engineering research, as opposed to other engineering disciplines [94].

A first attempt to remedy this situation has been undertaken by the ACM SIGCSE committee on teaching Computer Science Research Methods (SIGCSE-CSRM) [139]. They describe a research process framework (see figure 1.7), which consists

of four different questions that as a whole describe the general research process. The four questions are rather general in nature and not specific to computing science. However, what is specific to computing science is the type and breadth of the answers to these questions.

To answer the questions of the general research process (see figure 1.7) computing researchers use a wide variety of different research methods [53, 69]. Each method outlines a different process for obtaining these answers. Research methods, however, do not describe what kind of answers are interesting. Therefore, Shaw has developed a research classification framework, which describes the kind of answers that are of interest for software engineering research [136]. In short, she classifies research based on the type of the following three aspects:

- **Research questions** What kind of research questions are interesting for software engineering researchers? This corresponds to the more general question A in figure 1.7: What do we want to achieve?
- **Research results** A classification of the kind of research results, which help to answer the aforementioned research questions. This covers the following question of the general research framework of the SIGCSE-CSRM (see figure 1.7): What do we do with the data? (C). Indirectly, this also covers question B: Where does the data come from? Since the research result strived for is known, it is not hard to think up how to get the data.
- **Validation techniques** The framework classifies the kind of evidence that can be used to demonstrate the validity of the result. In the general research framework (see figure 1.7), this relates to question D: have we achieved our goal?

The work of Shaw is unique, as it provides an overview of the content of various research methods in a software engineering setting. Most publications on research methods either focus on a meta-analysis of used research methods in publications (e.g. [53, 69]) or describe a specific research method in isolation (e.g. [105, 179]). In the latter case, the research method is nearly always described for the use in another disipline (e.g. social sciences) than software engineering. To use such a research method in software engineering, it often needs to be adapted to this specific research context. Shaw's framework makes precisely such a transformation for various methods.

In the remainder of this section, the research methods used in this thesis are presented first. This is followed by a more in-depth description of the research questions, results, and validation techniques used in software engineering in general and this thesis in particular.

1.6.2 Research methods

To get a better picture on the research methods used in software engineering, Glass et al [53] tried to identify, amongst others, those research methods that are commonly used in journal publications. Not surprisingly, they found that the majority of research methods used in software engineering are qualitative in nature, as opposed to the more quantitative methods used in other engineering research disciplines. They found that the conceptual analysis (both informal and mathematically), and concept implementation, i.e. a proof of concept, are the two most used research methods in software engineering. Case studies, data analysis, surveys, and simulations are used as well, but considerably less then the two aforementioned research methods.

Based on the work of Glass et al [53], the SIGCSE-CSRM committee tried to come up with a better list of research methods. The main motivation for this work was that in the analysis of Glass et al 54% of the journals used the conceptual analysis research method. However, a complete detailed description of this research method was lacking. In their revision, the committee identified 55 different research methods being used in computing science [69]. Of these 55, the following three research methods are used in this thesis:

- **Interview** This is a research method for gathering information in which people are posed questions by an interviewer. The interviews may be structured or unstructured both in the questions asked by the interviewer, as well as the answers available to the interview subject. This leads to the following four types of interviews [105]:
  - **Informal conversational interview** In this type, no predetermined questions are asked in order to remain as open as possible to the interviewee's nature and priorities. During the interview the interviewer tries to "go with the flow".
  - **General interview guide approach** The guide approach is intended to ensure that the same general areas of information are collected from each interviewee. This provides more focus than the conversational approach, but still allows a degree of freedom and adaptability in getting the information from the interviewee.
  - **Standardized, open interview** The same open-ended questions are asked to all interviewees. This facilitates faster interviews that can be more easily analyzed and compared. However, it does require the interviewer to have a good set of questions to start with.

- **Closed, fixed-response interview** In this type, all interviewees are asked the same questions and asked to choose answers from among the same set of alternatives. This format is useful for those not practiced in interviewing and makes analyzing the results relatively easy. The downside of this is the inflexibility, as both questions and answers are predetermined.

In general, the benefit of the interviewing research method is its ability for getting in-depth information surrounding a particular topic. The major drawback of the method is that it is very time consuming and resource intensive. This thesis includes a chapter that uses the results of a general interview guide approach with software architects and project managers (see [162]) to address *RQ-7: How to provide good tool support for architectural decisions?*

- **Critical analysis of the literature** [179] This research method is a historical method, which collects and analyzes data from published material. To provide a careful evaluation, an evaluation framework can be created that describes the criteria on which the literature is being evaluated. The analysis part provides the opportunity to draw general conclusions from a broad range of approaches.

  The major benefit of this method is that it can provide useful information on a broad range of different approaches, while access to this information comes at a relatively low cost. The major weakness of this method is *selection bias*, i.e. the tendency of authors to publish mostly positive results and leave out inconsistent results. This might lead to incorrect conclusions during the analysis of the examined material. In this thesis, this research method is used to address the following research questions:

  - *RQ-4: What is the added value of explicit architectural decisions in the architecting process?*
  - *RQ-5: How is making architectural decisions part of the architecting process?*
  - *RQ-6: What are existing automated support tools for managing architectural design decisions and what do they support?*

- **Proof of concept** [55, 83] This research method is also known as proof of principle. It involves building something and then let that artifact stand as an example for a more general class of solutions. The act of creating a solution in an iterative manner improves the understanding of the concept under consideration. Failed or incomplete iterations provide evidence to the creator about the problems associated with the concept. In a way, this research method is similar to software development [104], as both try to create something [55]. The major difference lies in what they want to achieve. The proof of concept research

method is about creating methodologies, concepts, and techniques, whereas in software development the goal is to create only a working software system.

Closely related to the proof of concept research method is the proof by demonstration method [69]. Both are similar, as they build an artifact to provide the proof of something. However, the main difference lies in the fact that with a proof by demonstration an existing technology is hypothesized to have a set of benefits in a new domain [55], whereas in a proof of concept neither the concept nor the benefits are known upfront.

The main drawback of this research method is the high risk of the artifact failing long before anything is learned about the concept for which proof is sought. In addition, the artifact may become more important to the researcher then the concept that needs to be proved. This is partially due to the research method, which ignores the formulation of a hypothesis up front, but rather lets it emerge during the process.

In this thesis, the proof of concept research method is primarily used to answer the following research questions:

- *RQ-1: What are architectural design decisions?*
- *RQ-2: How can we model architectural design decisions?*
- *RQ-2.1: How can we model features in an SPL?*
- *RQ-3 How to deal with architectural design decision dependencies?*
- *RQ-3.1 How to deal with feature interactions?*
- *RQ-7: How to provide good tool support for architectural decisions?*
- *RQ-8: How can we recover architectural design decisions?*

1.6.3 Research question types

Research questions form the starting point of many research methods. Choosing the right method for answering a research question still is an art in software engineering research. Nevertheless, it is important to know, which kind of research questions are "interesting" for software engineering researchers. In her software engineering research classification [136], Shaw distinguishes five types of research questions to be of interest. In short, these are the following illustrated by the research questions of this thesis (see section 1.5 on page 16):

- **Method or means of development** The central research question of this thesis is of this type: *RQ: How to reduce architectural decision vaporization?*, as well as *RQ-3 RQ-3.1*, and *RQ-8*

- **Method for analysis** *RQ-6* is partially of this type: *What are existing automated support tools for managing architectural design decisions and what do they support?*
- **Design, evaluation, or analysis of a particular instance** Both *RQ-4: What is the added value of explicit architectural decisions in the architecting process?* and the aforementioned *RQ-6* are of this particular type.
- **Generalization or characterization** The majority of research questions found in this thesis are of this type. They include *RQ-1*, *RQ-2* and *RQ-2.1*, *RQ-5*, and *RQ-7* (see section 1.5 on page 16).
- **Feasibility** This thesis does not include any research questions of this particular type.

1.6.4 Research results

The use of one or more research methods to answer the formulated research questions creates various research results. Shaw's classification framework identifies eight different kinds of such results. Following is a short description of each research result type, together with a description of the research results, as they are found in this thesis:

- **Procedure or technique** The result is a new or better way to perform some task. An example of this result type is the Architecture Design Decision Recovery Approach (ADDRA) presented in chapter 7.
- **Qualitative or descriptive model** A model describing the structure or taxonomy of a problem area. Examples of this result type include the different versions of a conceptual model for describing architectural design decisions, as found in chapters 3 and 4. Other examples found in this thesis are the architecting process models of chapters 3 and 7, which describe the place architectural decisions have in the architecting process.
- **Empirical model** A model providing predictive power based on observed data. This thesis does not include research results of this type.
- **Analytical model** A structural model precise enough to support formal analysis or automatic manipulation. Two examples of this result type can be found in this thesis: the feature model of chapter 2 and the Archium meta-model presented in chapters 4 and 5. The feature model obviously models and describes features in SPLs. It is also used to investigate the problem of feature interactions. The Archium meta-model does the same, but for architectural design decisions in general.

- **Notation or tool** A formal (graphical) language to support a technique or model or a tool supporting such a language. The Archium tool (see chapters 4 and 5) implements the Archium meta-model and is an example of the tool research result type. The Archium tool provides tool support for architectural decisions. An example of such a notational research result is the decision trace view, which is presented in chapter 4. This view shows how multiple architectural design decisions influence the software architecture design.
- **Specific solution** A solution to a particular application problem, which illustrates the use of some software engineering principles. The recovered (see chapter 7) and contemplated (see chapter 4) architectural decisions of the Athena case are examples of this.
- **Answer or judgement** The result of a specific analysis, evaluation or comparison. An example of such a research result is the evaluation framework for architectural evolution presented in chapter 6. Architectural decisions form an important part of this evaluation framework.
- **Report** A report about interesting observations and discovered rules of thumb. The lessons learned in chapter 7 of using the Architecture Design Decision Recovery Approach (ADDRA) are an example of a report result.

1.6.5 Validation techniques

Validation techniques help a researcher to check the validity of his or her research results. As there are many different research results in software engineering research (see section 1.6.4 on the preceding page), it will come as no surprise that there are many different validation techniques as well. This subsection presents a classification of these validation techniques by Shaw [136]. For the validation techniques used in this thesis, appropriate examples are provided within the description of this classification:

- **Analysis** The data is analyzed and is found satisfactory for what we want to achieve. The following types of analysis can be distinguished:
  - **Formal analysis** A rigorous derivation and proof using formal semantics.
  - **Empirical model** Analysis of the data on the actual use of the research result.
  - **Controlled experiment** The conclusions from a carefully designed statistical experiment.
- **Experience** The research result has been used on real examples by someone else and the evidence of its correctness / usefulness / effectiveness is validated by any of the following techniques:

  - **Qualitative model** A narrative of the application of the research result.
  - **Empirical model** Data is collected on the practice of the research result and statistically analyzed.
  - **Notation / tool technique** A comparison with other notations or tool techniques is made with similar results in actual use.

- **Example** An example of how the research result works is provided. Two different types of examples can be discerned:

  - **Toy Example** A simplified example, which might have been motivated by reality. This validation technique is used in chapter 3 with a CD player to illustrate the concept of architectural design decisions. Chapter 4 uses this technique to illustrate the composition technique for dealing with decision dependencies in Archium. In a similair fashion, the toy example of a video shop is used in chapter 2 to validate solutions to the problem of feature interactions.
  - **Slice of life** A system that the author has developed. This validation technique is used a lot in this thesis. Chapter 2 uses a prototype implementation of a video shop rental system. Chapters 4 and 7 use the Athena case to validate the Archium system and the recovery of architectural design decisions with ADDRA. The Archium tool was also validated by the development of a chat application in chapter 5.

- **Evaluation** The criteria against which the results are evaluated are developed up front. The following types of evaluation are distinguished:

  - **Descriptive models** The research result is evaluated to the extent that it successfully describes the phenomena of interest. This technique is used in chapter 5 to validate some aspects of the Archium tool, using use cases based on interviews with architects.
  - **Qualitative models** These models describe the extent to which the research result accounts for the phenomena of interest. An example of the application of this technique can be found in chapter 6, which presents an evaluation framework and applies it on various architectural tools, including Archium.
  - **Empirical models** Models built according to the research results that fit the real data.

- **Persuasion** The researcher elaborates on the validity of the research result and gives some convincing arguments for it.

  - **Technique** Give an explanation of how the use of the technique in question can create certain benefits. This validation technique is used in chapter 7

alongside the slice of life validation technique to convince the reader about the validity of the ADDRA.

- **System** A reasoning about the benefits a system would have if constructed according to the research results.
- **Model** An argument why a particular model is reasonable. Chapter 3 uses this validation technique to persuade the reader about the validity of the rationale and the software architecting process models. Similarly, chapter 2 tries to convince the reader about the use of different solutions to deal with the effects of feature interactions.

- **Blatant assertion** No serious attempt is made to evaluate the result.

Table 1.1 summarizes the research question types, research results, and validation techniques per research question of this thesis. From this table the research method used in this thesis can be distilled. In short, this method is to analyze relevant aspects of software development by developing a model and validating it through experiences and examples. This is achieved by using the "proof of concept" and the "critical analysis of the literature" research methods (see section 1.6.2 on page 20).

1.7 Overview of this thesis

The main body of this thesis is based on publications in a book (chapter 3), journals (chapters 2, 7), and international conferences (chapters 4, 5, and 6). Apart from some minor corrections, these chapters are the same as these publications. An exception is formed by chapter 6, where the original accepted version is used, instead of the shortened published version. The chapters are not presented in a chronological order, but instead in a logical order to enhance the readability of this thesis. To denote the chronological order, the relative order of each publication is described in a time-line element. In addition, for each chapter, the relationships it has to the previously stated research questions is made explicit (see section 1.5). In short, this thesis consists of the following chapters:

First class feature abstractions for product derivation [74] In the context of a product line, one would like to derive products based on feature selections. In this paper, we argue that features are a subset of the solution an architecture provides. Furthermore, by representing a feature with a first class representation we can automatically derive products. However, so-called 'feature interactions' make this derivation process very complicated. This paper concludes that the origin of the feature interaction problem lies in the composition of the features and this is the fundamental problem feature based product

RQ	RQ type	Research result	Validation technique
RQ-1	- Generalization or characterization	- Qualitative or descriptive model	- Persuasion model
RQ-2	- Generalization or characterization	- Analytical model	- Slice of life
RQ-2.1	- Generalization or characterization	- Analytical model	- Slice of life
RQ-3	- Method or means of development	- Notation or tool	- Toy example - Slice of life
RQ-3.1	- Method or means of development	- Analytical model	- Toy example - Persuasion model
RQ-4	- Design, evaluation, or analysis of a particular instance	- Qualitative or descriptive model	- Persuasion system
RQ-5	- Generalization or characterization	- Qualitative or descriptive model	- Persuasion Model
RQ-6	- Method for analysis - Design, evaluation, or analysis of a particular instance	- Qualitative or descriptive model - Answer or judgement	- Qualitative model
RQ-7	- Generalization or characterization	- Notation or tool - Specific solution	- Descriptive model - Slice of life
RQ-8	- Method or means of development	- Procedure or technique - Qualitative or descriptive model - Specific solution - Report	- Slice of life - Technique

Table 1.1: Overview of the classification per research question

derivation needs to address. The paper identifies three basic solutions one can use to address this issue.

While writing this paper we noticed that our concept of a feature was quite general. Especially, our viewpoint that a feature is a subset of the solution made us realize that features can be seen as a special kind of design decision.

Time-line: First paper written. **Chapter:** 2 **Authors:** Anton Jansen, Rein Smedinga, Jilles van Gurp, and Jan Bosch **Research questions:** RQ-2.1: How can we model features in an SPL?, RQ-3.1: How to deal with feature interactions?

Design Decisions: The Bridge between Rationale and Architecture [163] This paper argues that design decisions form the natural bridge between rationale and architecture. Both the rationale management and the architecting process are analyzed and compared to each other. To relate both processes, the bridging concept of a design decision is needed. The benefit of relating both processes is that it opens the way to (systematically) capture the rationale behind an architecture. The paper presents a global introduction into the Archium approach, which describes how design decisions could be documented.

Time-line: Fourth paper **Chapter:** 3 **Authors:** Jan van der Ven[2], Anton Jansen[2], Jos Nijhuis, and Jan Bosch **Research question:** RQ-4: What is the added value of explicit architectural decisions in the architecting process? RQ-5: How is making architectural decisions part of the architecting process?

Software Architecture as a Set of Architectural Design Decisions [77] The next conceptual step in our thinking is to see the architecture as a set of architectural design decisions. In this perspective, the software architecture is the *result* of a design decision making process. To support this notion, the architecture part of the Archium is presented.

Time-line: Third paper. **Chapter:** 4 **Authors:** Anton Jansen and Jan Bosch **Research questions:** RQ-1: What are architectural design decisions? RQ-2: How can we model architectural design decisions?

Tool support for Architectural Decisions [79] The problem statement and the motivation for our research into design decisions was greatly improved with this paper. This paper clearly identifies the problem of *knowledge vaporization* and the problems that were earlier indentified as consequences of this general

[2]Both authors contributted equally to this paper

problem. Furthermore, motivation for the relevance of Archium is demonstrated by the elaboration of various use-cases that are relevant with regards to the use of design decisions. To make this possible, the paper also describes the Archium tool in more detail, especially the requirement model used in Archium. **Time-line:**Fifth paper. **Chapter:** 5 **Authors:** Anton Jansen, Jan van der Ven, Paris Avgeriou, and Dieter Hammer **Research question:** RQ-7: How to provide good tool support for architectural decisions?

Evaluation of Tool Support for Architectural Evolution [76] This paper presents an evaluation framework to judge and evaluate different approaches on their support for tracking design decisions and the evolution of an architecture. Using this framework, five existing approaches are evaluated from the software architecture and knowledge management domain perspective. The paper concludes that there exists a gap between approaches from the knowledge management and the software architecture community with respect to the integration of design decisions into a software architecture.

The chapter in this thesis includes the accepted long version of this paper, whereas a short version was published in the conference proceedings. Furthermore, this paper was written before the Archium approach was developed and therefore does not evaluate this approach. To remedy this, an additional supplement is added to this chapter, which uses the same framework to evaluate Archium. This text was originally part of the 'Tool support for Architectural Decisions' paper, but was removed in the final version due to space constraints.

Time-line: Second paper.**Chapter:** 6 **Authors:** Anton Jansen and Jan Bosch **Research question:** RQ-6: What are existing automated support tools for managing architectural design decisions and what do they support?

Documenting after the fact: recovering architectural design decisions [78] The Archium approach is a forward engineering approach, while often one needs to recover design decisions a long time after they were made. This is not a trivial task to perform. This paper describes ADDRA, an approach to systematically recover design decisions in such situations. ADDRA differs from many other recovery approaches, as it uses a combination of tacit and explicit knowledge. The paper concludes that the considered solutions (i.e. alternatives), especially the ones that are not chosen, are the most difficult part of a design decision recovery. Hence, this is essential knowledge to capture in a forward engineering approach.

Time-line: Sixth paper, initially written before the second paper. **Chapter:** 7 **Authors:** Anton Jansen, Jan Bosch, and Paris Avgeriou **Research questions:**

RQ-8: How can we recover architectural design decisions? RQ-5: How is making architectural decisions part of the architecting process?

Other papers that have been published during the writing of this thesis, but are not part of this thesis are the following:

Athena, a large scale programming lab support tool *Anton Jansen. Proceedings of the Dutch National Computer Science Education Congress (NIOC 2004)* [75].

Using Architectural Decisions *Jan S. van der Ven and Anton Jansen and Paris Avgeriou and Dieter K. Hammer. Short paper, proceedings of the Second International Conference on the Quality of Software Architecture (Qosa 2006)* [162].