Architectural design decisions

Anton Jansen

To whom it may concern.

Correspondence information
E-mail mail@antonjansen
Website http://www.antonjansen.com

Cover design Van Kelckhoven BNO, Groningen, The Netherlands
Press Grafische Industrie de Marne, Leens, The Netherlands

ISBN: 978-90-367-3494-3

RIJKSUNIVERSITEIT GRONINGEN

Architectural Design Decisions

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. F. Zwarts,
in het openbaar te verdedigen op
vrijdag 19 september 2008
om 16:15 uur

door

Antonius Gradus Johannes Jansen

geboren op 20 februari 1979
te Oldehove

Promotor	:	Prof. dr. J. Bosch
Copromotores	:	Prof. dr. D.K. Hammer Dr. P. Avgeriou
Beoordelingscommissie	:	Prof. dr. I. Crnkovic Prof. dr. K. Koskimies Prof. dr. J.C. van Vliet

ISBN: 978-90-367-3494-3

ACKNOWLEDGMENTS

"Luctor et emergo" would nicely describe the relationship I have with this thesis. In retrospect, this thesis has been in many aspects the ultimate confrontation with some of my own limitations. Happily enough, I have fought and overcome them, but I couldn't have done this without the help of many people. Some of them I would like to especially acknowledge here.

First of all, I would like to thank my (co-)supervisors: Jan Bosch, Dieter Hammer, and Paris Avgeriou for their patience and guidance. In particular, Jan, for demonstrating to me that no research goal is too high to aim for. Dieter, for bringing reality to Jan's visions and Paris for always having wise advise available when I was stuck.

While writing this thesis, I have been a member of the Software Engineering an ARCHitecture (SEARCH) at the university of Groningen. The research group has seen many changes over the years. I would like to thank my (former) colleagues for providing an inspirational working environment: Jilles van Gurp, Johanneke Siljee, Ivor Bosloper, Marco Sinnema, Sybren Deelstra, Lisette Bakalis, Theo Dirk Meijler, Mugurel Ionita, Eelke Folmer, Michel Jaring, Jan Salvador van der Ven, Jos Nijhuis, Peng Liang, Marco Aiello, Rein Smedinga, Trosky Callo, Neil Harrison, Ahmad Waqas, Eirini Kaldeli, and Elie El-Khour. In particular, I would like to thank Jilles for teaching me how to write papers. It took some time, but I finally seem to manage! Ivor and Johanneke, your reference to ArchJava came at the right moment. It has given me the perspective I needed and thus the motivation to keep going. Lisette, thank you for helping seeing things a PhD in perspective and regard it as just another project. Marco and Sybren, the magic duo, the small-talks we had during many tea breaks and Friday afternoon drinks were much appreciated. Last but not least, Jan Salvador for our nice and fruitful cooperation on architectural decisions and for bringing the small-talks with Marco and Sybren to the next level of vagueness and ambiguity.

Many insights presented in this thesis have come out from numerous discussions with students about this subject. In particular, I would like to thank the Archium

crew: Joris van der Burgh, Frans Kremer, Zef Hemel, Marnix Kok, Wouter-Tim Burgler, Robert-Slagter, and Mattijs Kersten, without you all, we never would have gained so much valuable insight. For helping me placing my work in the broader context of architectural knowledge and providing a nice related project to work in, I would like to thank the VU people of the Griffin project: Viktor Clerc, Patricia Lago, Hans van Vliet, Remco de Boer, and Rik Farenhorst.

I would also like to thank the secretaries, Ineke Schelhaas, Esmee Elshof, and Desiree Hansen, for helping me dealing with the university bureaucracy. In addition, this thesis would not have been possible without the boundless effort and legacy of the system *nix administrators: Harm Paas, Jurjen Bokma, and Peter Arends. Thank you for helping me out with a millions of things.

I wouldn't have survived the last 5 years without spending some time with my friends and on my hobbies. I would like to thank Ron Heffels for giving me the new hobby of cycling to remain fit. The members of the online gaming community "The Conclave" for providing countless hours of fun. Matthew "Kaz" Chaplain for sharing his unique perspectives on things and proofreading parts of this thesis. Hanneke Kruidhof, for the many conversations we had in reflecting upon our lifes and work. My movie buddies, Koen de Raadt and Reinco Hof, for keeping me up-to-date movie wise. My friends, Gert Jan Kamstra, Martin Heres, Jasper de Boer, Arend Smit, Adriaan Renting, and Jeroen Hoeboer for sharing countless adventures these years, let's hope more are still to come!

Finally, to wrap-up this lengthy acknowledgement, I would like to thank my parents Gerard and Nynke, and my brother Wiebren, for their unconditional love and support during all these years.

Anton

CONTENTS

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.

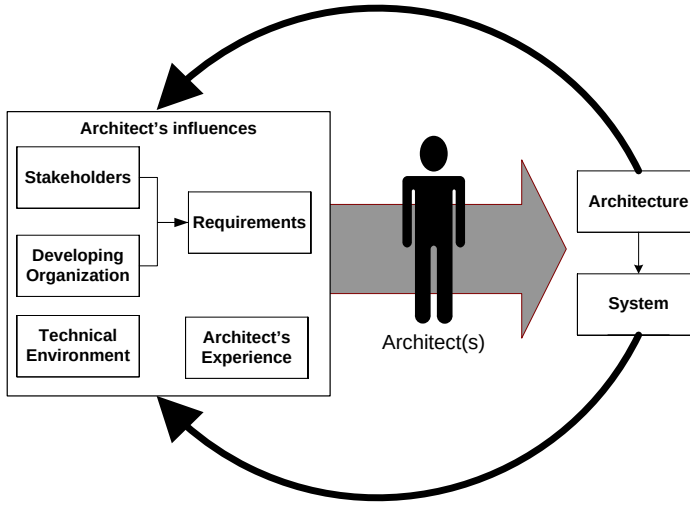

Figure 1.1: The architecture business cycle from [11]

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.

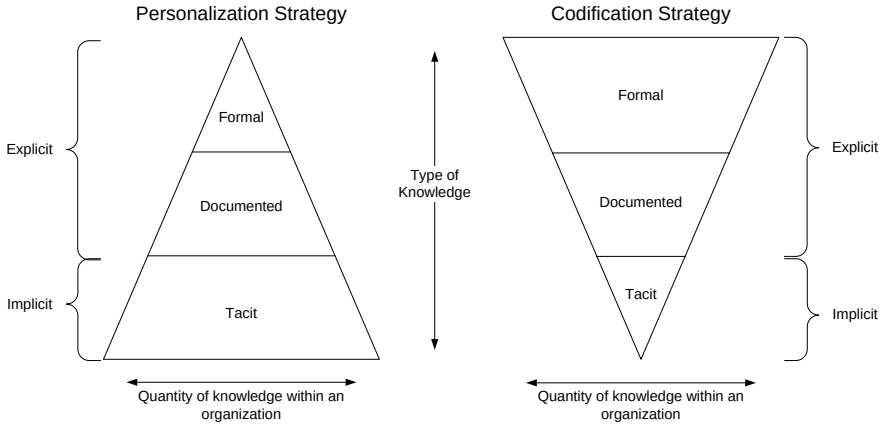

Figure 1.2: Pyramid of knowledge types and the associated knowledge management perspective

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

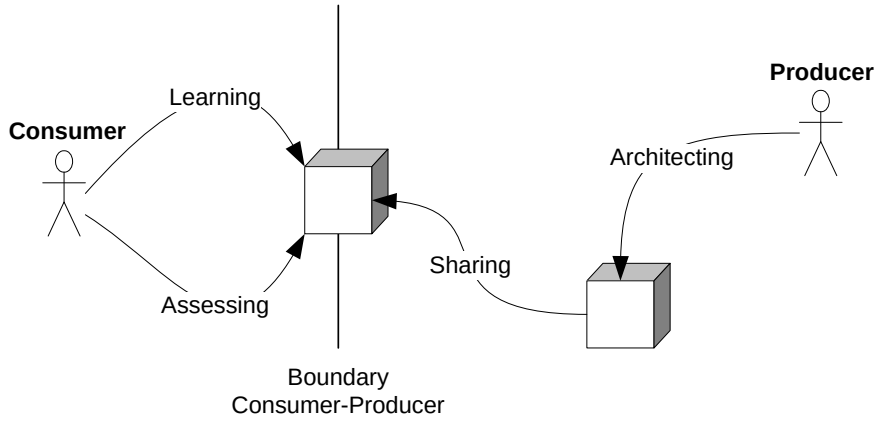

Figure 1.3: Producer-consumer perspective on Architectural Knowledge from [98]

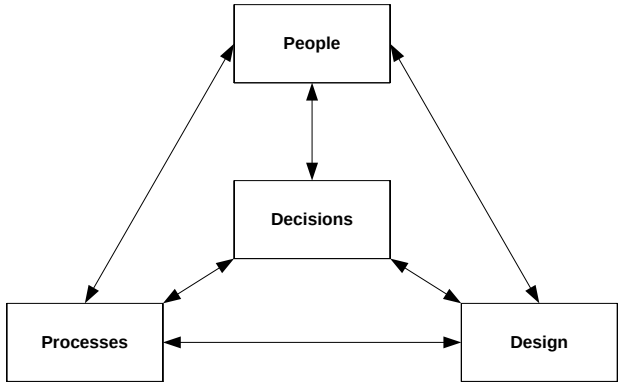

Figure 1.4: The four parts making up the Griffin Core Model [39] of Architectural Knowledge

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

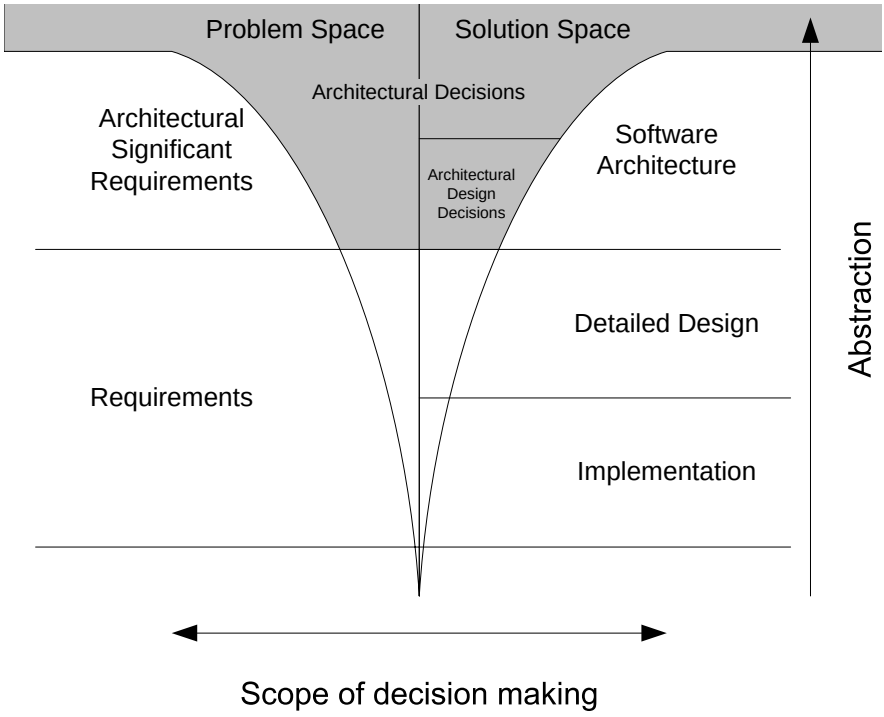

Figure 1.5: The funnel of decision making

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

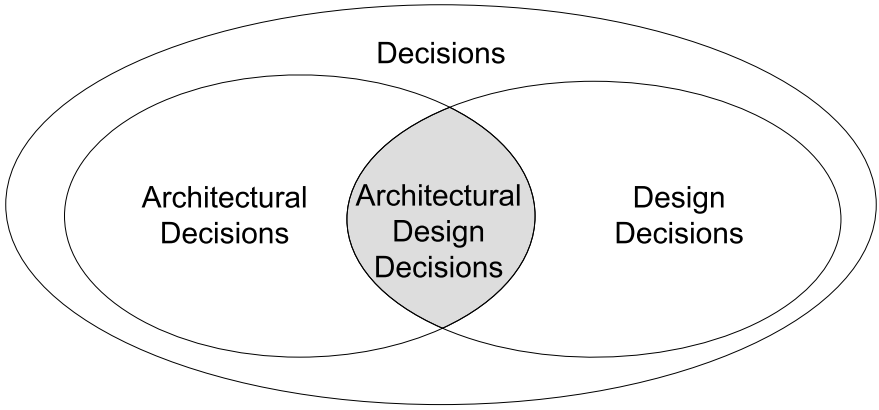

Figure 1.6: Type of decisions and their relationships

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?

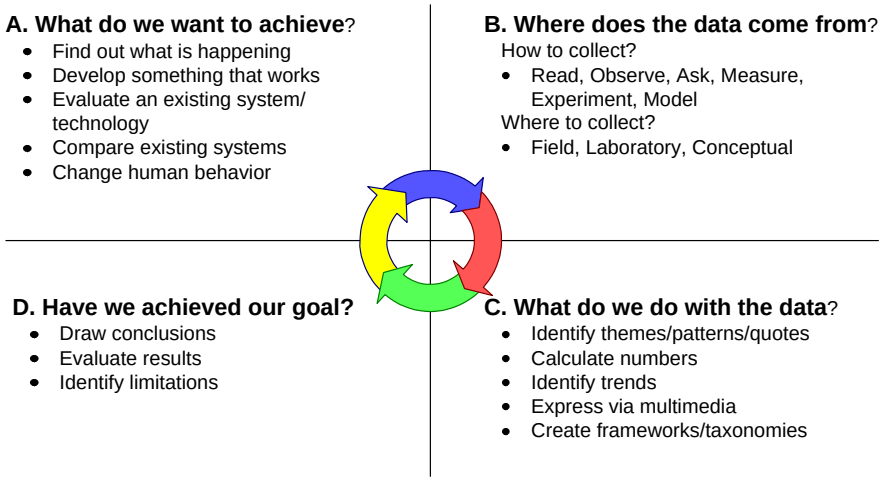

Figure 1.7: The research process framework by [69]

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].

CHAPTER 2

FIRST CLASS FEATURE ABSTRACTIONS FOR PRODUCT DERIVATION

Published as: Anton Jansen, Rein Smedinga, Jilles van Gurp, and Jan Bosch, First class feature abstractions for product derivation, Special issue on Early Aspects: Aspect-oriented Requirements Engineering and Architecture Design. IEE Proceedings Software 151(4), pp. 187-197, August 2004.

Abstract

The authors have observed that large software systems are increasingly defined in terms of the features they implement. Consequently, there is a need to express the commonalities and variability between products of a product family in terms of features. Unfortunately, technology support for the early aspect of a feature is currently limited to the requirements level. There is a need to extend this support to the design and implementation level as well. Existing separation of concerns technologies, such as AOP and SOP, may be of use here. However, features are not first class citizens in these paradigms. To address this and to explore the problems and issues with respect to features and feature composition, the authors have formalised the notion of features in a feature model. The feature model relates features to a component role model. Using our model and a composition algorithm, a number of base components and a number of features may be selected from a software product family and a product derived. As a proof of concept, the authors have experimented extensively with a prototype Java implementation of their approach.

2.1 Introduction

Software applications grow larger and larger, are maintained for longer periods of time and need to be updated frequently to evolve with new needs and changing consumer requirements. To cope with this increasing size of software applications a software product family (SPF) [19, 20] approach can be used. An SPF is designed for a family of (domain) related applications. It consists of a product-line architecture and a set of reusable components. Specific applications may be derived from the SPF by selecting, enhancing and adapting components. We have observed that, during product derivation, the differences between products are usually defined in terms of features [19, 57, 84, 85].

Features are an example of early aspects [127], crosscutting concerns at the requirements and architectural level. Features are used in requirement engineering to define optional or incremental units of change [52]. They have a many-to-many relationship to the individual requirements [19]. Tracing features to the implementing SPF components is complex. In ideal cases, a particular feature implementation is localised to a single module; but in many cases features will cut across multiple components [58]. Consequently, features are an important early aspect to consider, as they try to bridge the gap between the problem domain and solution domain [157].

The product derivation process in an SPF is a timeconsuming and therefore expensive process. The reason for this is that there is a mismatch between the way products are defined (i.e. in terms of features) and the variability offered by the SPF. Requirements changes generally result in changing and=or adding features to the SPF. However, typically features have no first class representation in the SPF implementation. During product derivation, developers must make adaptations to the SPF's provided feature set in order to implement product specific features. Consequently, changing the implementation to meet new requirements is potentially expensive because code related to one feature may be spread over multiple software components.

Ideally, new or changed features would be captured in separate pieces of code that can be changed and maintained independently. Thus changes during the product derivation would be limited to those pieces of code. The main topic of this paper is giving features a first class representation in SPFs so that during product derivation, product developers may select features from the SPF and reuse them in their products. There are a number of problems associated with this type of product derivation. A key contribution of this paper is that we identify those problems and demonstrate in our approach how these can be worked around or solved.

We use a top-down approach of analysing the issue of feature based product derivation. Our top-down approach starts with the modelling of concepts, such as features and SPFs, in terms of sets. With the help of this formal description of the feature model, composition problems are identified. Several solutions to these composition problems are presented. One of these solutions is used in a prototype implementation that is also presented in this paper. The three contributions of this paper are:

- A feature model, which relates features of an SPF with component roles.
- A classification of feature composition problems and potential solutions to these problems.
- A demonstration of how features can be realised at the implementation level. This opens the way to automatically derive a product from an SPF based on a selection of available features.

2.2 Features in software product families

To clarify our approach, an examination of how features and SPFs are related is presented. After this, an informal outline of our approach is presented. The approach involves features, actors and roles. At the end of the section, the approach is demonstrated using the example of a video shop renting system.

2.2.1 Software product families (SPFs)

An SPF consists of a base implementation (e.g. B) and a number of features (e.g. $F_1 ... F_2 5$). A product may be derived from the SPF by selecting an arbitrary number of these features and combining these with the base implementation (e.g. $B + F_9 + F_1 8 + ... + F_2 3$). The base implementation itself can also be seen as a set of (standard) features, i.e. an SPF then becomes, for example, $F_1 + F_2 + F_3 + F_4 + F_9 + F_1 8 + ... + F_2 3$, where some features (e.g. 1 to 4) are standard features (in FODA these are called mandatory features [84]) and others are optional features. In this paper, base components model entities, which cannot easily be decomposed into features. Legacy code components and domain components are examples of these base components.

The properties of the composition operator '$+$' are our primary interest in this paper. Of course, it would be ideal if that operator were associative i.e. $[(F_1 + F_2) +$

$F_3] = [F_1 + (F_2 + F_3)]$, and commutative, i.e. $F_1 + F_2 = F_2 + F_1$. Then, a product developer would be able to arbitrarily combine features. The developer of each feature would not need to worry about interaction with other features. This way, it would not make any difference at what point in time and/or development a certain feature is brought into a feature composition. However, in general (as will be argued in the following section) this operator is neither associative nor commutative, because of feature dependencies: one feature may depend on another feature. This is the case if one feature cannot operate without another feature.

A further complication is that the composition of features might introduce feature interaction [52]: a feature interaction is some way in which one or more features modify or influence another feature in describing the system's behaviour set.

An example of feature interaction can be found in Microsoft Outlook. Outlook, a popular e-mail client for Windows, has two related features: work off-line and 'send immediately'. The work off-line feature enables users to use the e-mail client without having a permanent connection to their mail server. While working off-line, Outlook caches the different actions of the user and executes them when the user switches to on-line mode. On the other hand, if the send immediately feature is enabled, a message is sent immediately when a user presses the send button.

Clearly both features influence each other. The send immediately feature should be disabled if the user is working off-line. However, this is not the case in Outlook. At present, Outlook still tries to send a message even though the user is working off-line. A potential cause for this problem could be that both features were implemented independently from each other. The problem only surfaces when both features are enabled. Consequently, unit testing will not detect this problem.

Feature interactions, like the example of Outlook, are very common in large software systems. As Zave observes, this type of problem potentially makes the composition of features incomplete, inconsistent, nondeterministic, hard to implement, etc. (see [178]). The method introduced in this paper aims to keep the composition complete, consistent, deterministic and implementable.

2.2.2 Roles, actors and base components

The modelling of an SPF, as a base implementation composed of a set of selected features, demands a more detailed modelling of a feature. Our approach does not assume how the inner workings of the base components are defined, only the assumption that some of them exist. The relationship between a feature and the base components is defined through a role-based approach.

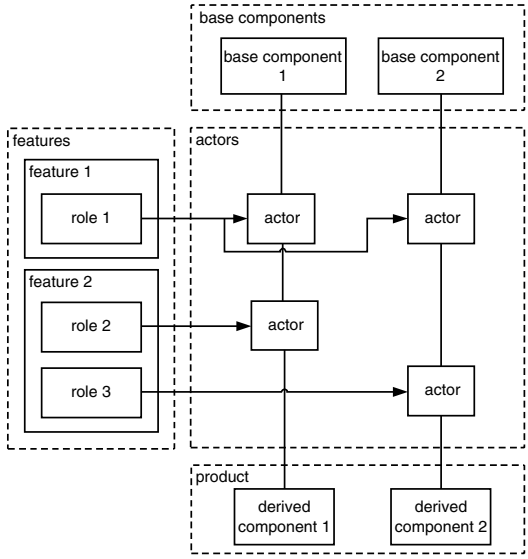

Figure 2.1: Conceptual view of the feature model

Role modelling is used, because features typically affect more than one domain component simultaneously. In this perspective, a feature can be viewed as a collection of base components playing roles of a feature. However, there is no simple one-to-one relationship between the roles of a feature and the roles 'played' by the base components. This simple relationship does not exist, because we want to define the features and their roles independent of the base components. As a consequence of this flexibility, there may be a base component playing more than one role of a feature. The opposite is also possible: one role of a feature can be played by several base components. To model this many-to- many mapping of roles and base components, our approach uses the concept of actors.

Note that with the term 'actor', a different concept is meant than is used in parallel object-oriented programming [1]. In the feature model, an actor is a first-class representation of a base component and the roles it plays for a single feature. Figure 2.1 visualises the various feature model elements and their relationships. The base components visualised in the top part of figure 2.1 are entities, which cannot easily be decomposed into features and belong to the base SPF implementation. On the left side of figure 2.1 two features containing one and two roles are presented, visualising the fact that roles are part of a feature. At the centre there are four actors. The top two actors consist of base components 1 and 2, both playing role 1. The bottom two actors are base component 1 playing role 2 and base component 2

playing role 3. At the bottom of figure 2.1, the derived components are situated. A derived component is a base component that incorporates actors playing the roles of the selected features. The concepts we discussed here form the basis of our composition approach, which will be elaborated on in section 2.4. Before that, however, we provide an example.

2.3 Case

Throughout the rest of this paper a video rental administration system is used to illustrate various aspects of the feature model. A quick domain analysis provides the following domain components for the video shop system: a **VideoShop** component, a **Video** component and a **Customer** component. These three domain components are the base components of this case. For the remaining part of the paper components are typeset in **bold**. Features are typeset underlined. The following features have been selected for this case:

- VideoRental: A **Customer** can rent a **Video**
- ReturnVideo: A **Customer** can return a **Video** that is rented
- AmountDiscount: A **Customer** receives a certain discount when renting more then one **Video**
- RegularCustomerDiscount: A regular **Customer** receives a certain discount when renting a **Video**
- AgeControl: Only a **Customer** above a certain age may rent a certain **Video**

These features are selected because they illustrate the various issues of the feature model. The system should always contain the features VideoRental and ReturnVideo, for the system to have a minimal of functionality. Both features, however, will not be part of the base components because the specification of the features might change over time. The other features are optional. Some features are dependent on each other, e.g. all optional features depend on VideoRental. Also, some features will have feature interaction, for example AmountDiscount and Regular Customer Discount, both influence the amount of money a customer has to spend.

Figure 2.2 presents an overview of the video shop case. The different features of the video shop, e.g. VideoRental, ReturnVideo, etc., can be found on the left side. The base components (**Videoshop**, **Customer**, **Video**) are found on the top. The features consist of one or more roles, for example the VideoRental feature

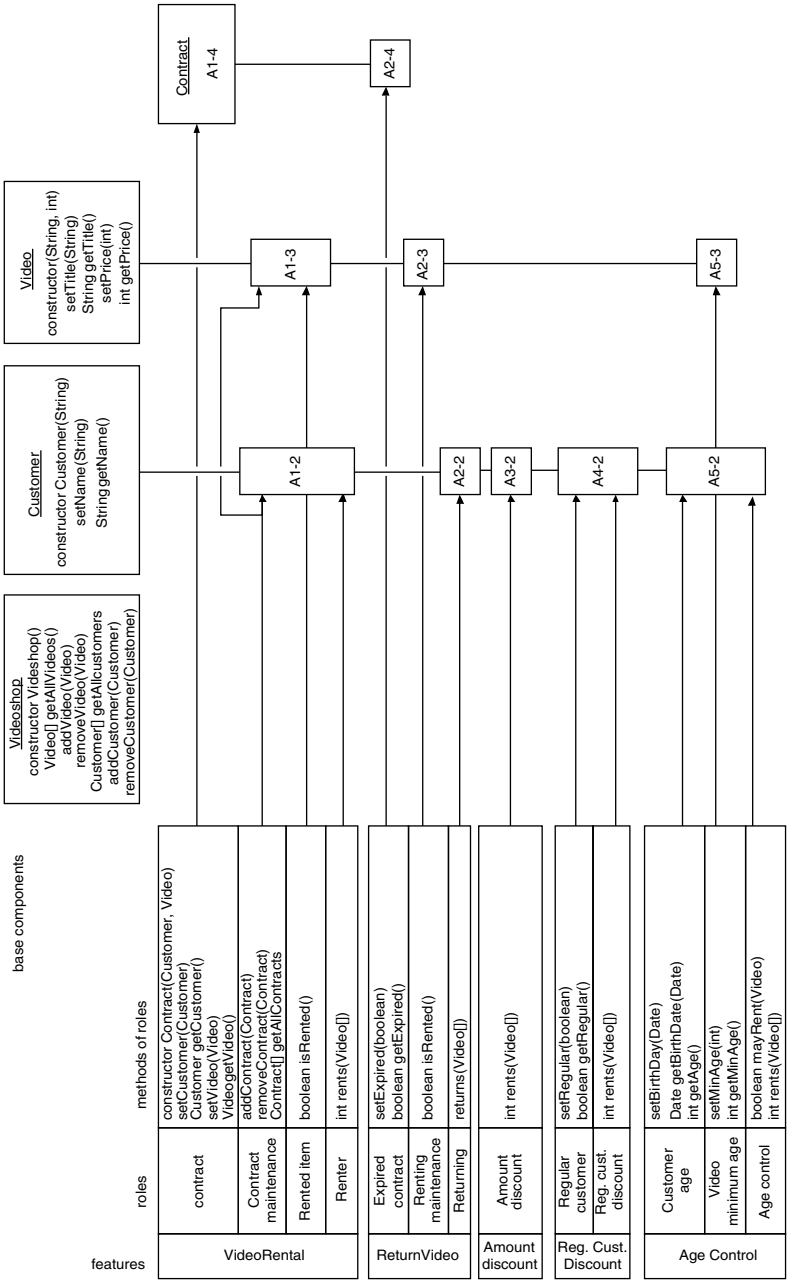

Figure 2.2: Features, roles, and actors of the video shop renting system

consists of the following roles (roles are in `teletype`): `Contract`, `Contract Maintenance`, `RentedItem`, `Renter` and `Maintenance`.

The first role, `Contract` of the VideoRental feature, introduces a new concept into the feature model. The `Contract` role introduces a new component in the composition, the **Contract**, which is not directly related to any existing base component. New concepts in the domain may be added by roles defining new components.

Because features are decomposed further into roles, the core of our composition method consists of mapping the defined roles onto the base components. The mapping of the different features and roles is visualised in figure 2.2. For each of the features, the roles are presented and the functionality of the roles is visualised by displaying the signatures of the corresponding methods. Furthermore, the functionality of the base components is visualised.

When a role is mapped onto a base component, an intermediate component is created (we refer to these intermediate components as actors); these are the little rectangles in the middle. Each actor has a unique name. For example the name of the **Contract** actor is A1-4. The first number indicates that the actor belongs to the first feature (i.e. VideoRental). The second number indicates the base component to which the role has been mapped. The `Contract` role has been mapped to a new 4th base component.

2.4 Formalising the notion of features

Composition of features such as those described in section 2.2 is far from trivial. Therefore, to be able to identify potential composition problems, the notion of features is formalised. In section 2.5, we use this formal model to derive the properties of the composition operator.

The feature model is formally defined in terms of sets. Mappings of elements of one set to elements of another set (e.g. $A \rightarrow B$) denote relationships between those elements. In the model a method signature is denoted by an operationSignature, a unique identifier for the complete definition of the method itself without an implementation; for example, in Java this is the header of a method, including the method name, the list of parameters, and the type of the returned value. A set of such signatures is called an interface:

$$interface = \{operationSignature_i | i \in signatureSet\}$$

With this notation, an interface is denoted as a set of operation signatures, named $operationSignature_1$, $operationSignature_2$, etc., where $signatureSet$ is the complete set of the available operation signatures.

A role is a set of interfaces and a one-to-one mapping of the operation signatures of the interfaces to implementations of these operation signatures. In imperative languages this implementation can be seen as a code block, i.e. the body of a method without the header. A role can now be defined as:

$$\begin{aligned} role \quad = \quad \{ \quad & \{interface_k | k \in interfaceSet\}, \\ & \{operationSignature_{k_i} \rightarrow implementation_{k_i} | \\ & k \in interfaceSet \wedge i \in signatureSet_k\} \ \} \end{aligned}$$

The mapping describes that an operation signature is implemented by associating an implementation with the operation signature. A role is a partial implementation, mapped onto a component.

Separate roles in a feature are required to model the fact that one component can have multiple roles in the context of a feature. To do the mapping of a role onto a base component, an intermediate form may be used. In a feature, the implementations are mapped onto actors. An actor is a set of roles from a feature, mapped to a base component. An actor can be seen as an intermediate component.

A feature is a set of roles, a set of actors, and a many-to-many mapping from roles to actors, i.e.:

$$\begin{aligned} feature \quad = \quad \{ \quad & \{role_r | r \in roleSet\}, \\ & \{actor_o | o \in actorSet\}, \\ & \{role_i \rightarrow actor_j | i \in roleSet \wedge j \in actorSet\} \ \} \end{aligned}$$

A role may be mapped to more than one actor. Also, more than one role can be mapped to the same actor. One role can map to more than one actor, if the corresponding code is going to be used in more than one component. Although this will in general be considered a signal of bad design, it is not excluded in our model.

A software product family (SPF) consists of all features and all base components:

$$SPF = \{feature_f | f \in featureSet\} \cup \{baseComponent_o | o \in baseSet\}$$

A specific product, derived from a SPF, consists of a selected number of features, a set of derived components, and the mapping from the actors to the derived component implementations, which in turn are derived from the base components, i.e.:

$$\begin{aligned} product \quad = \quad \{ \quad & \{feature_s | s \in selectedFeatures \subseteq featureSet\}, \\ & \{component_c | c \in compSet\}, \\ & \{actor_i \rightarrow component_i | i \in actorSet_s \wedge j \in compSet\} \ \} \end{aligned}$$

The set of derived components is derived from the set of base components through the mapping of the actors to the base components. The component set is explicitly included, as there can be a need for base components that do not have roles mapped on them, but are used by included features.

For example, this is the case for BC1 in figure 2.3. As a consequence of this definition, the set of derived components contains at least as many elements as the set of base components, i.e.:

$$baseComponent_o \subseteq components_c$$

The transformation from a base component to a derived component is not formalised here. This transformation is the main issue in our approach and is investigated further in the following sections. Figure 2.3 illustrates our approach: methods are mapped onto the components through the actors. Actors may introduce new components, which are independent of the defined base components. These new components are dependent on an additional, initially empty, base component **none**.

Note that we have introduced three types of components: the base components, new components and derived components. The base components come from a domain model or are legacy components. The new components are components introduced by the roles of new features. The derived components are components generated for a specific derivation.

Our model currently does not model feature dependencies. The reason for this is that modelling dependencies complicates the feature model too much for our purposes. Therefore, the assumption has been made that the dependencies among the features are known and can be resolved before applying the composition operator.

Feature dependencies only influence the *order* in which the composition operator is applied. However, this does not affect *how* the composition operator should work. Consequently, we do not explicitly model feature dependencies because they are not relevant for our purpose of examining the properties of the composition operator.

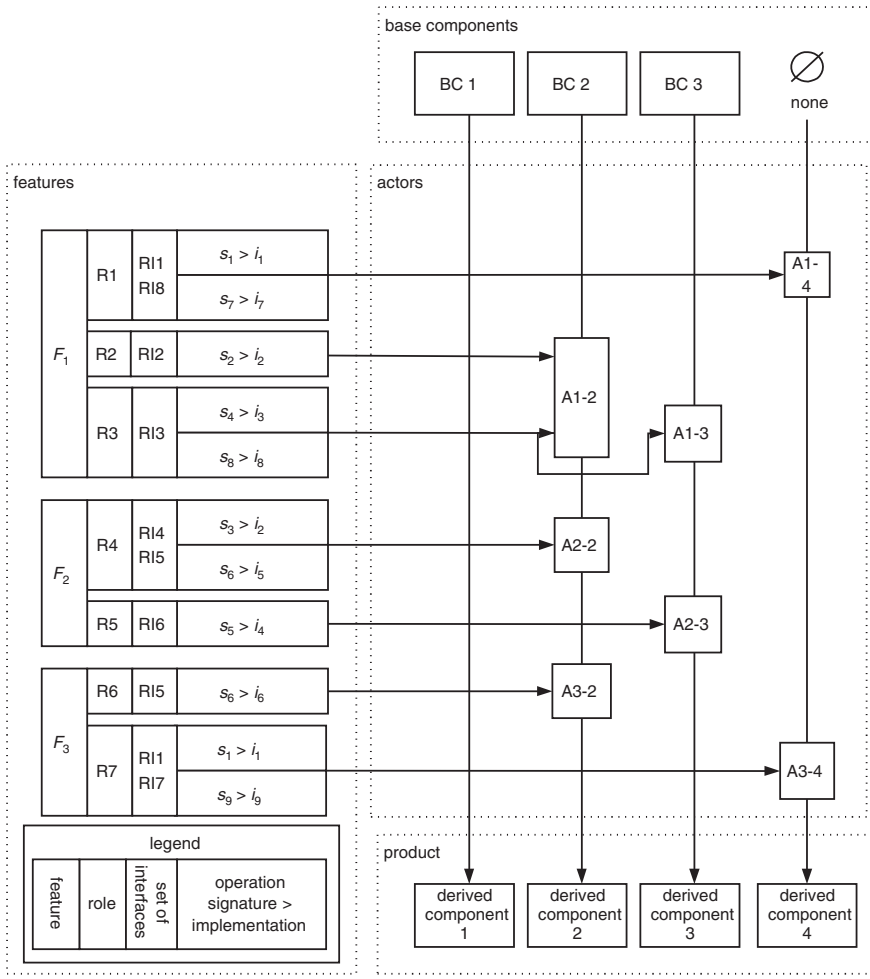

Figure 2.3: Graphical representation of feature composition

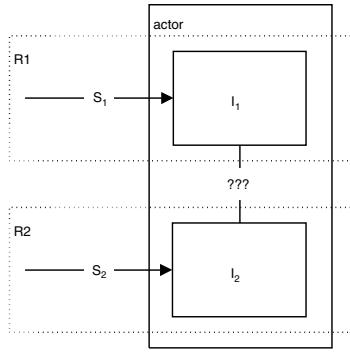

Figure 2.4: Composition of an actor

2.5 Compositon operator

In this section, the composition operator for the feature model is investigated. The composition operator in the feature model is used to compose features with base components. A feature, however, consists of one or more roles that map onto an actor. This section investigates how an actor can be composed of roles and base components, what the associated composition problems are, and how these problems may be solved.

2.5.1 Introduction

An actor consists of roles and base components. In the feature model, the derived components contain the functionality of the corresponding base components and actors of the selected features that are mapped to the base component. As mentioned earlier, feature dependencies complicate the composition process. Ideally, the order in which features are mapped to base components would not affect the semantics of the derived components. However, because of the dependencies, the order does matter (i.e. the composition operator is not commutative). Conceptually, the actors are accumulated on top of the base components (i.e. each actor is composed with the composition of all previous actors and the base components). Each actor combines various roles of a feature that are mapped to the same base component.

For example, figure 2.4 visualises the composition of two roles (R1 and R2). To simplify the composition problem, figure 2.4 does take into account that an actor can be composed with a base component or another actor. However, this has no

consequences for the composition operator. Both roles (R1 and R2) consist of one operation, denoted by S_1 and S_2, and an implementation for this operation (i.e. I_1 and I_2). The actor that results from the composition of role A and B should contain the unified behaviour of roles A and B. Both implementations A and B are considered to be black boxes. The composition of the actor then becomes the problem of 'gluing' both implementations together, as denoted in figure 2.4 with the question marks.

2.5.2 Analysing the composition of roles

Both operation signature and implementation have an effect on the 'glue' that is needed to compose the implementations. By looking at the relationships between the operation signatures and the implementations, four different types of composition can be identified:

Signatures and implementations are all different. Figure 2.3 illustrates this: roles R2 (with $s_2 \rightarrow i_2$) and R6 (with $s_6 \rightarrow i_6$). R2 is mapped onto actor A1-2 and R6 is mapped onto actor A3-2. Both A1-2 and A3-2 are mapped onto the same base component BC2. An example in the video shop (figure 2.2) is the `Renter` and `Returning` roles.

This situation does not raise any problems because there is no interaction between the implementations.

The signatures are different and the implementations are equal. In figure 2.3 this is illustrated in roles R2 (with $s_2 \rightarrow i_2$) and R4 (with $s3 \rightarrow i_2$). Role R2 is mapped onto actor A1-2 and R4 onto A2-2. Both A1-2 and A2-2 are mapped onto the same base component BC2. The video shop example does not contain this situation.

This situation does not present any problems either. It might signal bad design because different signatures can be implemented using the same implementation so the signatures might be considered equal instead of different.

Both the signatures and implementations are equal. This looks like copy & paste reuse and also a bad practice. In figure 2.3 this is illustrated in roles R1 (with $s_1 \rightarrow i_1$) and R7 (again $s_1 \rightarrow i_1$). R1 is mapped onto actor A1-4, R7 onto A3-4 and both actors are mapped onto the same base component BC4.

The video shop example does not contain this situation. Although code fragments appear double in the resulting application there are no serious problems. Problems may arise, however, when maintenance is needed (code needs to be repaired in different places). A simple solution for this kind of problems is simply mapping the different code fragments to just one fragment.

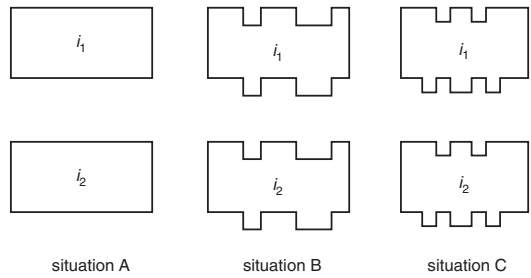

Figure 2.5: Graphical representation of variations of having one signature with two implementations

The signatures are equal and the implementations are different. In figure 2.3 this is illustrated in roles R4 (with $s_6 \rightarrow i_5$) and R6 (with $s_6 \rightarrow i_6$) Role R4 is mapped onto actor A2-2, R6 onto actor A3-2 and both actors are mapped onto the same base component BC2. In the video shop case an example of this situation can be found with the operation `rents` in the roles `Renter` and `AmountDiscount`. This is a serious problem that requires further investigation. The remainder of this section is devoted to this problem.

Of the four described combinations for composing the roles, only the last one is problematic. The combination of one signature with more than one implementation can be illustrated best with Lego building blocks. An operation signature is the header and the implementation the body of a method. Equal operation signatures therefore means that the headers are equal, and thus the parameter list and return type of the methods implementing the operation signature are equal and only the body is different.

A Lego brick can be seen as a shape representing the operation signature: the shape of the top of the brick illustrates the parameter list and the shape of the bottom of the brick illustrates the return type. The inside of the brick represents the implementation. Because in situation four, the signature is the same for both implementations, the Lego bricks have the same shape. This results in three ways to combine the implementations, as illustrated in figure 2.5:

A. No input parameters, no output. The operation signature of the implementations has no return type and no parameters. This is the easiest situation because the implementations may just be concatenated.
B. Input is equal to output. If the implementations have the same input type as the return type, the implementations may be piped together. However, this requires that the semantics of the input and output match.

C. Input and output are different. In this situation the implementations can neither be concatenated nor piped together. Some glue code may be needed to transform both implementations into one implementation.

In all cases, a transformation is needed that combines two different implementations into one implementation for the same operation signature. In Lego terms this compares to building a new stone with the same shape (i.e. with the same input parameters and output parameter). Problems arise because of initialisation at the beginning of each of the two implementations, the output parameters of each of the implementations, and side-effects such as, for example, exception handling.

Any implementation of a feature composition will need to make specific design choices with respect to these transformations. First, three alternatives are presented for these transformations. This is followed in section 2.6 with a discussion on a prototype algorithm where several design choices are made with respect to these transformations.

2.5.3 Composing implementation blocks

There are several ways of combining the two implementations with the same operation signatures, but there are three basic forms:

- **Concatenation:** The implementations can be concatenated: i_a; i_b or i_b; i_a. Concatenating the implementations can only be done if the output of the first implementation can be used as input for the second. Thus, concatenation can only be used in situations A and B of figure 2.5. Even then, side effects such as exception handling may prevent successful concatenation.
- **Skipping:** One of the implementations can be skipped: i_a or i_b. Skipping one of the implementations requires an additional criterion to be able to select which one of the implementations to skip.
- **Implementation mixing:** The implementations may be mixed (e.g. by superimposing [18], inheritance or delegation). Mixing the implementations, the last possibility of the three, requires knowledge of both the implementations i_a and i_b. Suppose i_a is in a feature F_a, i_b is in a feature F_b, and F_a is dependent on the feature F_b. Then it is possible to use i_b everywhere in the code of i_a; because they have equal operation signatures. At this point this is best illustrated by comparing this kind of mixing with using an original method of a superclass in the redefined method in the subclass by calling `super` in Java.

There are a number of issues with the different combination strategies described above:

- **Scope of variables:** Both implementations may have a common set of local variables with different semantics, which may raise some conflicts when both implementations are combined. A possible solution is to automatically rename conflicting local variables.
- **Side-effects:** Both implementations may have conflicting side-effects. For example, both implementations may throw an exception. The code of the second implementation may never be executed if the first implementation throws an exception. There are many subtle ways both implementations may conflict that need to be considered when combining the implementations.

It should be pointed out that other approaches (e.g. AspectJ [87, 88]) exhibit the same sort of problems. In particular, AspectJ has become a complex language due to the fact that it tries to solve/work around these issues.

2.6 Prototype implementation of feature model

The previous section used the formal model of feature composition to examine where exactly feature composition becomes problematic, namely when combining implementations with the same signature into one implementation. We have outlined three strategies, that may be combined, for doing so. However, there are a number of issues that prevent an universal solution to this problem. Any implementation of our feature composition model requires that these issues be addressed in some way. In this section, a composition algorithm is outlined, which addresses these issues and is demonstrated with the video shop prototype. The section ends with an overview of implementation issues encountered and solutions that may resolve these issues.

2.6.1 Prototype

The prototype that is presented in this section is intended as a proof of concept. Consequently, a limitation of this is that the prototype is missing some features that would be available in a complete implementation. The composition operator as implemented only supports one variant of the implementation mixing composition solution (see section 2.5.3). In addition, an important limitation is that automatic product derivation is not supported with a compiler. Instead, the transformation of the features and the base components into the derived components is done manually. However, it will be possible to automate this in the future.

The main motivation for building the prototype was to demonstrate that the composition technique described earlier can be implemented in a mainstream programming language, such as Java. However, features are not a first-class entity in Java. Consequently, the feature model entities have to be mapped to Java language constructs. Some of the design choices regarding this mapping are:

- **Feature:** A feature is a collection of roles in the feature model. Neither features nor roles have a representation in the Java language. However, Java supports the concept of a package that may be used to group various classes together. The prototype uses the package construct to group roles of the same feature together.
- **Role:** In the feature model a role is a collection of interfaces and some code blocks. A Java class also has interfaces and code blocks. Therefore, a role is implemented as a Java class in our prototype.
- **Base component:** Similarly, base components are also implemented as Java classes.
- **Actor:** The goal of feature composition is to combine the base components and the roles in such a way that the result has the composed behaviour of both. When a role is mapped to a base component, an intermediate placeholder class (i.e. an actor) is created that inherits from the base component class.
- **Derived component:** Derived components are the result of the composition of selected features and base components. As stated before, actors combine base components and roles using inheritance. However, the derived components should incorporate all the composed behaviour of all the actors and base components. Since Java does not have multiple inheritance, the derived component cannot be constructed by letting them inherit from all the actors' classes.

  To work around this problem, actors inherit from each other. Consequently, the last actor of a base component will have the required composed behaviour of a derived component. Therefore the derived component is an empty placeholder Java class, which inherits from the last actor defined on a base component. An example of this can be found in figure 2.6. A more in-depth description of the composition process and an algorithm for the composition in the Java language can be found in [73].

In figure 2.6 a UML example of the prototype for the feature model of figure 2.1 is presented. The packages of the prototype are visualised as grey boxes which can contain other packagers or classes. The classes are the white rectangles and inheritance is illustrated as an arrow from the subclass to the parent class.

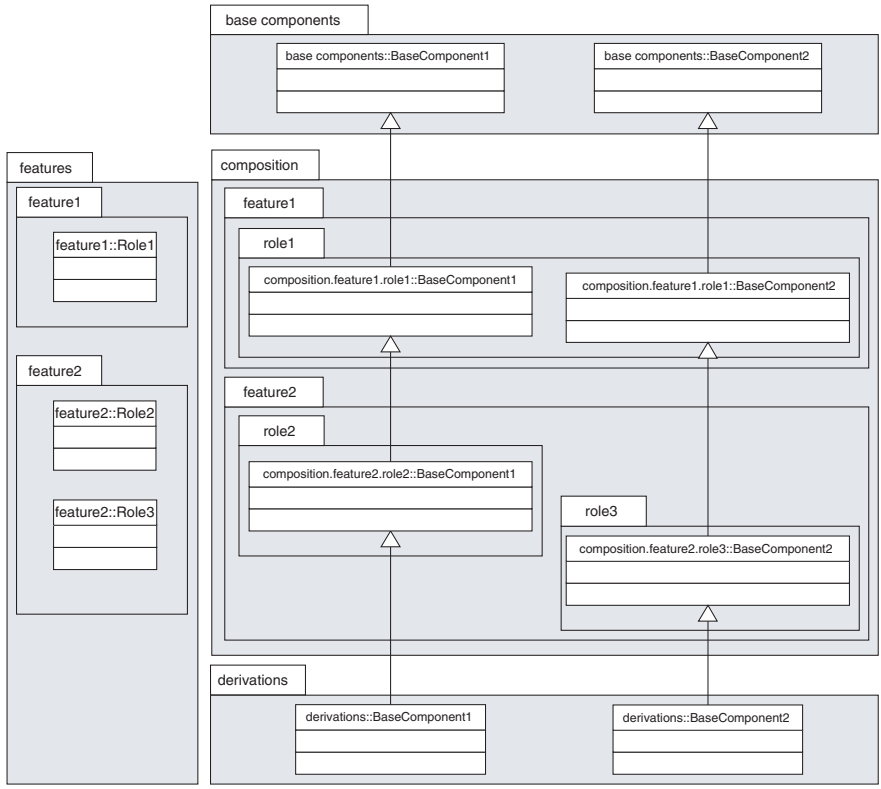

Figure 2.6: UML diagram of prototype implementation of figure 2.1

2.6.2 Potential issues for automatic composition

This section presents a description of the various implementation issues that arose during the implementation of the prototype. The issues discovered are believed to be general for implementing automatic feature composition based on the feature model. The main issues found during the implementation of the prototype are:

- the lack of a dependency model in the feature model
- an instantiation problem in the roles
- traceability of roles, features and actors

The lack of a dependency model in the feature model is an issue an implementation has to deal with. The feature model assumes that a feature introduced earlier is not

dependent on a later feature. The fact that this does not have to be the case in an implementation requires modelling of the dependencies at least at the implementation level. Dependencies also help to define the context of a feature in which a role is specified; hence it restricts the knowledge an implementer needs to implement a role. Experience has taught that there are four types of dependencies:

- *Dependency between roles of the same feature:* An example of dependencies among roles in the same feature can be found in the VideoRental feature of the video shop case (see figure 2.2). One can only rent a *Video* if the concept of a *Contract* is introduced (this is in the `Contract` role) and the necessary `Contract` operations in the *Customer* and *Video* (these are in the `ContractMaintenance` role) are known.

- *Dependency between roles of different features:* Dependencies between roles of different features are the basis for feature dependencies. Feature dependencies in the feature model are the result of roles depending on roles or actors of a different feature. An example is the role `ExpiredContract` of ReturnVideo, which is dependent on the VideoRental feature, because of the needed concept of a *Contract*. This concept is introduced in the VideoRental feature by the `Contract` role.

- *Dependency between a role and an actor:* This dependency is different from a role depending on another role, because in this dependency a role is dependent on the composition of a role and a specific base component, which is an actor. An example is the `Returning` role of the Return_Video feature. The *Contract* that the `Returning` role uses should have the `Contract` role (from the VideoRental feature) and also the `ExpiredContract` role (from the ReturnVideo feature), to be able to expire a contract for this *Customer*.

- *Dependency between a role and a composition of multiple actors:* This dependency is a dependency between a role and a composition of roles and a base component. An example of this dependency is in the `Returning` role of the ReturnVideo feature. The *Video* returned should contain the `ContractMaintenance` role and the `RentingMaintenance` role to be able to determine whether the *Video* is already rented and to add a new *Contract* to the *Video* if this is not the case.

The prototype implements the two role related dependencies with the help of the traditional Java dependency model (i.e. the use of the `import` statement). The two other dependencies are only partially realised. The recursive way actors are

composed makes the use of traditional Java dependencies between a role and an actor semantically different.

The second implementation issue is the instantiation problem. At the moment the roles are written it is not determined which class should be instantiated, because other features can be added or removed on the fly. Observe that the last actor of a component contains the complete composed behaviour for that component; this is due to the recursive composition behaviour of the actors. Each of the components has its own derived component, which should contain the complete composed behaviour for that component depending on the selected features.

The derived component, therefore, could inherit from the last defined actor for the component. If, during the derivation process, the derived component confirms to this inheritance and has a stable name, then it can be instantiated in the different roles.

Another implementation issue is the traceability of roles, features and actors, which is required for debugging the derived components. In the prototype the traceability of the different actors is accomplished by the first class representation of the actors. The name of the package in which the actor class is defined is determined by the feature and role names. The name of the actor class is equal to the name of the base component on which it is mapped, resulting in a complete reverse mapping from the derived components back to the roles, features and base components.

2.7 Related work

This section provides an overview of related work and their relationship with this paper. Separation of concerns, features, role modelling, and software product families and software architecture are the four areas of interest of which related work is examined.

2.7.1 Separation of concerns

Separation of concerns is the appliance of the divide-and-conquer paradigm on software design. By separating different concerns in separated entities the design becomes easier, but the 'gluing' of the pieces becomes harder.

Subject orientated programming (SOP) [65] uses the concept of different views on an entity. Each view has it own object hierarchy. Composition rules define how the different object hierarchies can be combined into a single unified object hierarchy.

Our approach differs in the focus, which is on the collaboration aspect of feature related variability and not on functional hierarchical differences.

Aspect oriented programming (AOP) [88] uses the concept of aspects to capture functionality that is crosscut in normal object decomposition. So-called join points provide hooks to merge aspects with the objects. One of the implementations of AOP is AspectJ [87]; here the join points are the method activations. A conceptual model stating what aspects are is missing in AOP. The feature model presented in this paper can be used as a conceptual model for aspects, with the aspects implementing the composition of our feature model.

Multi-dimensional separation of concerns [153], as implemented in the HyperJ [116] approach, models different concerns in independent dimensions. Rules defining the relationships between the independent entities of the dimensions guide the necessary composition process for system generation. Our feature model can be viewed as a two-dimensional instance of a multi-dimensional separation of concerns model. The first dimension is the concern of the base components, the second the feature related variability dimension. The resulting matrix of actors is very similar to a composition expression in hyperslice programming. However, our model is different as it explicitly distinguishes features, and adds additional semantics (interface, roles) and notation (see figure 2.3). The feature model is not restricted to only these two dimensions of separation of concerns, because no restrictions on the dimensionality of the base components or features are defined.

The composition problem found in this paper (see section 2.5) is universal for multi-dimensional separation of concerns, because each concern model will only describe a part of the behaviour of an entity. However, each concern model needs to overlap/relate to other concern models, otherwise a composed view of an entity is not possible. Multi-dimensional separation of concern, therefore, has the inherent problem that an operation of an entity could have multiple behaviours that should be combined, resulting in a composition problem.

2.7.2 Features

A more global view of how features can bridge the gap between the problem and solution domain is presented in [157]. In their view, features are composed out of requirements fragments and realised in one or more design fragments, which make up the complete design. In this perspective, features can be seen as an example of early aspects [127] as a crosscutting concern during the requirement engineering and architecture design phase. This paper presents a more detailed description of

how the design fragments, i.e. aspects, making up the feature can be modelled and composed for making the complete design.

Our approach is not unique in trying to model features in the solution domain. The feature-oriented domain analysis (FODA) [84] method is a method for identifying features during domain analysis. FODA uses the representation of feature trees to visualise the variability and dependencies of features. Later, the feature-oriented reuse method (FORM) [85], which is a superset of FODA, was developed to prescribe how the FODA feature model could be used to develop domain architectures and components for reuse. The main difference between our approach and FORM is the traceability of features at the design and implementation level, which is not the case with FORM. This traceability is lost during the FORM application-engineering phase.

Prehofer [123, 124], uses feature oriented programming (FOP) to compose features into objects. FOP is an extension of the object-oriented programming paradigm. It uses separate entities called lifters to model feature interaction and separates core functionality from feature functionality. Our approach differs in two ways: the first is that a first class entity (the actor) for the composed behaviour is present, enabling the definition of a feature based on the composed behaviour of two other features. The second difference is that a feature is not one static class but consists of different roles being mapped onto different domain components.

Zave [178] discusses a distributed feature composition technique (DFC) for telecommunication services. She uses a pipe & filter style architecture, with the features being components, switches and routers connecting the components with connectors to form a chain of components through which data can move. Components can be added and removed by the switch, thereby changing the current feature set. The main difference with our approach is the fact that we do not require a particular architectural style to be used and features do not have to be contained in a single component.

The relationship between features and SPFs is also mentioned in [57], which proposes a feature-driven aspect-oriented product line. The main idea is to use a feature-driven analysis and design method like FeatuRSEB [59] to develop a feature model. One or more aspect-orientated implementation techniques [36] can then be used to implement features in separate code fragments, which in turn can be composed based on the selected features for a given composition. The global idea is the same as the approach in this paper; however, the focus of this paper is more on the composition of features, whereas [57] is more a global modelling view.

2.7.3 Role modelling

The idea of role modelling has also been studied by other authors. The OOram method [156] uses role models based on collaborations of different roles. Two or more role models can be composed to form a new and more complex role model. The general role modelling ideas presented in OOram are used at various abstraction levels. The main difference with our approach is in the composition of two role models. In OOram this is only done at a structural level; that is, only the structural requirements are validated and compositional problems have to be solved by the developer him/herself.

Role models can also be integrated into object-oriented frameworks [129]. The main focus of the role model used in this approach is on the interface part. The notation proposed in [129] could be used to denote the relationships between the roles of the features in our model. The main difference is that a coupling between an interface and an implementation of a role is not made in their model, which is something our feature model does. The composition problem we have identified is therefore not relevant in their approach, because the composition problem finds its roots in a coupling between an interface and an implementation.

Fowler [48] defines design patterns that can be used to implement roles in an object-oriented language. The main concept of the patterns is that the objects are dealing with a single object that has multiple changeable types. An object is therefore aware of the multiple typing of the other objects and has to act on this. In the feature model, this does not have to be the case, because we want to be able to develop unrelated features independently. The use of mixins [142], which are abstract subclasses representing a mechanism for specifying classes that will eventually inherit form a super class, is another approach to compose collaborations. The difference with our approach lies in the mapping of the roles of the collaborations to the objects. These can only be mapped to a single domain object, and multiple roles cannot be mapped to the same domain object. These restricution do not apply to our feature model.

2.7.4 Software product families and software architecture

The software product family (SPF) [19] is an approach for developing software, not on an application base, but on the basis of a family of related applications. The commonalities between the individual products (applications) can be used to create so-called common assets, which are reusable components that can be customised for the individual products. The field of variability management [61] of

SPFs mainly concerns how the differences between the products can be managed. This paper presents a method about how the common assets can be customised for a specific product based on a selection of features, and is therefore a form of variability management.

In the context of SPF, van Deursen [165] also use features to customise the common assets to derive a product based on a selection of features. They use packages as features and the merging of source trees to accomplish feature composition. The merging of the source trees takes place at so-called variation points. A variation point in their approach is a simple switch statement, defining the different variations on the code level.

A problem with this approach is the definition of the variation points. The variation points have to be programmed out manually in the form of switches, and this mixes the feature related code with the common asset code. This reduces the traceability and the reuse capabilities of the feature related code, because of the lack of first class representation, which is present in our approach.

The software architecture (SA) [10] of each derived product is a variant of the SPF architecture. The feature model incorporates the components of the SPF architecture through the use of the base components. The feature model can therefore modify the architecture of a product by adapting the existing base components by mapping new roles of features on it and introducing new connectors and components resulting from features.

2.8 Conclusions and future work

In this paper, we have investigated the potential of using the early aspect of features in the solution domain of software product families (SPFs). Our main focus was on the design and implementation level. Starting at the design level, a feature model was presented. The model showed how features could be modelled as a collection of roles, thereby relating for the first time a feature model to a role model. The roles in turn can be played by different base components, resulting in actors. At the implementation level a way to implement the model is outlined. The model and the outlined implementation strategy are illustrated and validated with a prototype implementation.

With the help of a formalised version of the model a compositional problem is identified. The composition of two roles in an actor becomes problematic when both roles have different implementations for the same method. To solve the composition problem there are three potential solutions: skipping, concatenation and

mixing. One or more of the three solution forms can be used to solve composition problems.

By predefining how the composition is done and which composition solution to use in the case of a composition problem, we can keep the composition of our feature model complete, consistent, deterministic and implementable. This facilitates the automatic derivation of products in an SPF based on a selection of features.

Remaining open issues that we intend to address in future work are:

- *Automatic composition support:* An open issue of the prototype is the absence of a compiler that supports the composition process. At the moment, the necessary composition steps still have to be done manually, which makes the application of the composition process a very time-consuming and error-prone one. However, we have already defined an algorithm for the composition process, which can be programmed out easily, automating the composition process.

- *Scalability of the feature model:* Scalability of the feature composition model is one of the main aspects that demand additional validation. Although the feature composition model was designed for use in SPFs, it has not yet been demonstrated whether the model can be scaled up to this level.

- *Lack of a dependency model:* The feature model does not include a dependency model. The combinations of features that are possible for product derivation are directly related to the feature and roles dependencies. So, it is important to extend the feature model with a dependency model. The first steps have already been taken in section 2.6.2, where four dependency relation types are already identified.

- *Validation of the feature model:* A correct and complete validation of the feature model is difficult to accomplish. Cases can be used to validate the feature model, as does the video shop case in this paper, for example.

These open issues form the basis for further work. We would like to investigate how automatic composition support can help with the scalability of the feature model. For decent automatic composition support, the feature model should be extended to include a dependency model. Additional research is planned with an additional case helping us to investigate the scalability of the feature model and providing additional validation of our approach.

CHAPTER 3

DESIGN DECISIONS: THE BRIDGE BETWEEN RATIONALE AND ARCHITECTURE

Published as: Jan S. van der Ven[1], Anton G. J. Jansen[1], Jos A. G. Nijhuis, Jan Bosch. Design Decisions: The Bridge between Rationale and Architecture, chapter 16 , Rationale Management in Software Engineering, Allen H. Dutoit, Raymond McCall, Ivan Mistrik, Barbara Paech Editors, pp. 329-346 , Springer-Verlag, April 2006.

Abstract

Software architecture can be seen as a decision making process; it involves making the right decisions at the right time. Typically, these design decisions are not explicitly represented in the artifacts describing the design. Instead, they reside in the minds of the designers and are therefore easily lost. Rationale management is often proposed as a solution, but lacks a close relationship with software architecture artifacts. Explicit modeling of design decisions in the software architecture bridges this gap, as it allows for a close integration of rationale management with software architecture. This improves the understandability of the software architecture. Consequently, the software architecture becomes easier to communicate, maintain and evolve. Furthermore, it allows for analysis, improvement, and reuse of design decisions in the design process.

[1] Both authors contributted equally to this paper

3.1 Introduction

Software design is currently seen as an iterative process. Often used phases in this process include: requirements discussions, requirements specification, software architecting, implementation and testing. The Rationale Unified Process (RUP) is an example of an iterative design process split into several phases. In such an iterative design process, the software architecture has a vital role [119].

Architects describe the bare bones of the system by making high-level design decisions. Errors made in the design of the architecture generally have a huge impact on the final result. As such, a lot of effort is spent on making the right design decisions in the initial design of the architecture. However, the rationale underlying the architecture is not usually documented, because the focus is only on the results of the decisions (the architectural artifacts). Therefore the evaluated alternatives, tradeoffs, and rationalisations about the made decisions remain in the heads of the designers. This tacit knowledge is easily lost. The lost architecture knowledge leads to evolution problems [168], increases the complexity of the design [21], and obstructs the reuse of design experience [93].

To solve the problem of lost architectural knowledge, techniques for managing rationale are frequently proposed. Experiments show that maintaining rationale in the architecture phase increases the understandability of the design [22]. However, creating and maintaining this rationale is very time-consuming. The connection to the architectural and design artifacts is usually very loose, making the rationale hard to use and keep up-to-date during the evolution of the system. Consequently, there seems to be a gap between rationale management and software architecture.

To bridge this gap, we unite rationale and architectural artifacts into the concept of a design decision, which couples rationale with software architecture. Design decisions are integrated with the software architecture design. By doing this, the rationale stays in the architecture, making it easier to understand, communicate, change, maintain, and evolve the design.

Section 3.2 of this chapter introduces software architectures. Section 3.3 discusses how rationale is used in software architectures. Section 3.4 introduces the concept of design decisions. Section 3.5 presents a concrete approach that uses this concept. After this, related and future work is discussed, followed by a summary, which concludes this chapter.

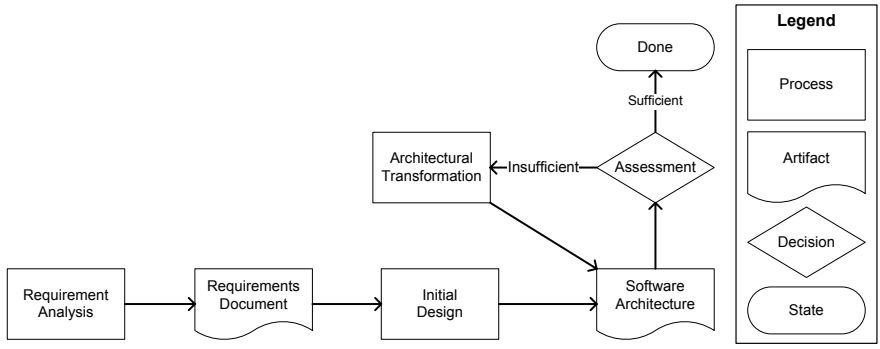

Figure 3.1: An abstract view on the software architecture design process

3.2 Software architecture

This section focuses on the knowledge aspects of software architectures. In subsection 3.2.1, the software architecture design process is discussed. Next, different ways are presented to describe software architectural knowledge in subsection 3.2.2. Subsequently, the issue of knowledge vaporization in software architecture is discussed in subsection 3.2.3.

3.2.1 The software architecture design process

A software architecture is based on the requirements for the system. Requirements define what the system should do, whereas the software architecture describes how this is achieved. Many software architecture design methods exist (e.g. [11] and [19]), and they all use different methodologies for designing software architectures. However, they can all be summarized in the same abstract software architecture design process.

Figure 3.1 provides a view of this abstract software design process[2] and its associated artifacts. The main input for a software architecture design process is the requirements document. During the initial design the software architecture is created, which satisfies (parts of) the requirements stated in the requirement document. After this initial design phase, the quality of the software architecture is assessed. When the quality of the architecture is not sufficient, it is modified (architectural modification).

[2]Remark this is a rather simplistic view, as the iterative interation with requirements is missing. Chapter 7 on page 143 provides a more detailed and precise view.

To modify the architecture, the architect can among other things, employ a number of tactics [11] or adopt one or more architectural styles or patterns [138] to improve the design. This is repeated until the quality of the architecture is assessed sufficient.

3.2.2 Describing Software Architectures

There is no general agreement of what a software architecture is and what it is not. This is mainly due to the fact that software architecture has many different aspects, which are either technically, process, organization, or business oriented [19]. Consequently, people perceive and express software architectures in many different ways.

Due to the many different notions of software architectures, a combination of different levels of knowledge is needed for its description. Roughly, the following three levels are usually discerned:

Tacit/Implicit knowledge In many cases, (parts of) software architectures are not explicitly described or modeled, but remain as tacit information inside the head(s) of the designer(s). Making this implicit knowledge explicit is expensive, and some knowledge is not supposed to be written down, for example for political reasons. Consequently, (parts of) software architectures of many systems remain implicit.

Documented knowledge Documentation approaches provide guidelines on which aspects of the architecture should be documented and how this can be achieved. Typically, these approaches define multiple views on an architecture for different stakeholders [70]. Examples include: the Siemens four view [67], and the work of the Software Engineering Institute [29].

Formalized knowledge Formalized knowledge is a specialized form of documented knowledge. Architecture Description Languages (ADL) [107], formulas and calculations concerning the system are examples of formalized knowledge. An ADL provides a clear and concise description of the architectural concepts used, which can be communicated, related, and reasoned about. The advantage of formalized knowledge is that it can be processed by computers.

Often, the different kinds of knowledge are used simultaneously. For example, despite that UML was not invented for it, UML is often used to model certain architectural concepts [29]. The model structure of UML contains formalized knowledge, which needs explanation in the form of documented knowledge. However, the use of the models is not unambiguous, and it is often found that UML is used in different ways. This implies the use of tacit knowledge to be able to understand and interpret the UML models in different contexts.

3.2.3 Problems in software architecture

There are several major problems with software architecture design [21, 76, 93]. These problems come from the large amount of tacit architectural knowledge. Currently, none of the existing approaches to describe software architectures (see section 3.2.2) give guidelines for describing the design decisions underlying the architecture. Consequently, design decisions only exist in the heads of the designers, which leads to the following problems:

- **Design decisions are cross cutting and intertwined.** Typical design decisions affect multiple parts of the design. However, these design decisions are not explicitly represented in the architecture. So, the associated architectural knowledge is fragmented across various parts of the design, making it hard to find and change the decisions.
- **Design rules and constraints are violated.** During the evolution of the system, designers can easily violate design rules and constraints arising from previously taken design decisions. Violations of these rules and constraints lead to architectural drift [119], and its associated problems (e.g. increased maintenance costs).
- **Obsolete design decisions are not removed.** When obsolete design decisions are not removed, the system has the tendency to erode more rapidly. In the current design practice removing design decisions is avoided, because of the effort needed, and the unexpected effects this removal can have on the system.

As a result of these problems, the developed systems have a *high cost of change*, and they tend to *erode quickly*. Also, the *reusability* of the architectural artifacts is *limited* if design decision knowledge vaporizes into the design. These problems are caused by the focus in the software architecture design process on the resulting artifacts (e.g. components and connectors), instead of the decisions that lead to them. Clearly, design decisions currently lack a first-class representation in software architecture designs.

3.3 Rationale in software architecture

To tackle the problems described in the previous section, the use of rationale is often proposed. Rationale in the context of architectures describes and explains the concepts used, alternatives considered, and structures of systems [70]. This section describes the use of rationale in software architectures. First, an abstract

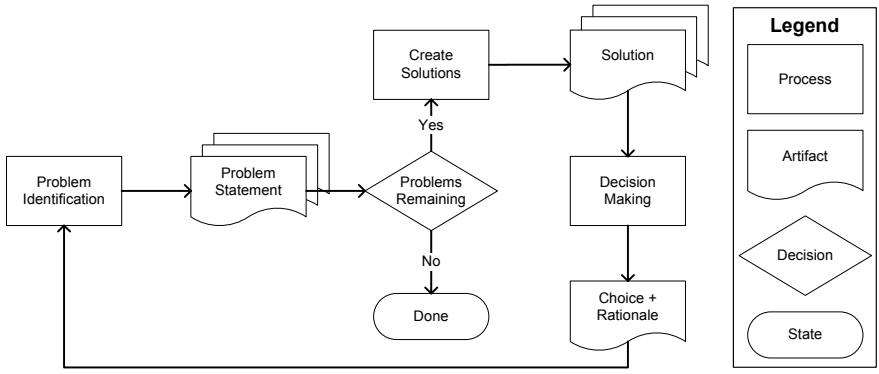

Figure 3.2: An abstract view on the rationale management process

rationale construction process is introduced in subsection 3.3.1. Then, the reasons for rationale use in software architecture are described in subsection 3.3.2. The section is concluded with a summary of problems for current rationale use in software architecture.

3.3.1 The rationale construction process

A general process for creating rationale is visualized in figure 3.2. First, the problems are identified (problem identification) and described in a problem statement. Then, the problems are evaluated (problems remaining) one by one, and solutions are created (create solutions) for a problem. These solutions are evaluated and weighed for their suitability of solving the problem at hand (decision making). The best solution (for that situation) is chosen, and the choice is documented together with its rationale (Choice + Rationale). If new problems emerge from the decision made, they have to be written down and be solved within the same process.

This process is a generalized view from different rationale based approaches (like the ones described in [42]). Take for example QOC, and the scenario described in [102]. The design of a scroll bar for a user interface is discussed. There are several questions (problems), like "Q1: How to display?". For this question, there are two options (solutions) described, “O1: permanent” and “O2: appearing”. In the described example, the second option is considered as the best one, and selected. However, this option generates a new question (problem), “Q2: How to make it appear?”. This new question needs to be solved in the same way. Other rationale management methods can be mapped on this process view too.

3.3.2 Reasons for using rationale in software architecture

As is discussed in [42], there are many reasons for using rationale in software projects. Here, the most important reasons are discussed, and related to the problems existing in software architecture.

- **Supporting reuse and change** During the evolution of a system and its architecture, the rules and constraints from previous decisions are often violated. Rationale needs to be used to give the architects insight into previous decisions.
- **Improving quality** As posed in the previous section, design decisions tend to get cross-cut and intertwined. Rationale based solutions are used to check consistency between decisions. This helps in managing the cross-cussing concerns.
- **Supporting knowledge transfer** When using rationale for communication of the design transfer of knowledge can be done over two dimensions: location (different departments or companies across the world) and time (evolution, maintenance). Transferring knowledge is one of the most important goals of an architecture.

3.3.3 Problems of rationale use in software architecture

As described in this section, rationale could be beneficial in architecture design. However, most methods developed for capturing rationale in architecture design suffer from the following problems:

- *Capture overhead*. Despite the attempt to automate the rationale capture process, both during and after the design, it is still a laborious process.
- For the designers, it is *hard to see the clear benefit* of documenting rationale about the architecture. Usually, most of the rationale captured is not used by the designer itself, and therefore capturing rationale is generally seen as boring and useless work.
- The rationale typically *loses the context* in which it was created. When rationale is communicated in documented or formalized form, additional tacit information about the context is lost.
- There is *no clear connection from the architectural artifacts to the rationale*. Because the rationale and the architectural artifacts are usually kept separated, it is very hard to keep them synchronized. Especially when the system is evolving, the design artifacts are updated, while the rationale documentation tends to deteriorate.

As a consequence of these problems, rationale based approaches are not often used in architecture design. However, as described in section 3.2.3, there is a need for documenting the reasons behind the design. The following section describes an approach which couples rationale to architecture.

3.4 Design decisions: the bridge between rationale and architecture

The problems from 3.2.3 and 3.3.3 can be addressed by the same solution. This is done by including rationale and architectural artifacts into one concept: the design decision. In the following subsection, the two processes from 3.2.1 and 3.3.1 are compared. In subsection 3.4.2, design decisions are introduced by example and a definition is presented in 3.4.3. The last subsection discusses designing with design decisions.

3.4.1 Enriching architecture with rationale

The processes described in subsections 3.2.1 and 3.3.1 have some clear resemblances. Problems (requirements) are handled by Solutions (software architectures / modifications), and the assessment determines if all the problems are solved adequately. The artifacts created in both processes tend to describe the same things (see figure 3.3). However, the software architecture design process focuses on the results of the decision process, while the rationale management focuses on the path to the decision.

Some knowledge which is captured in the rationale management process is missing in the architecture design process (depicted as black boxes in figure 3.3). There are two artifacts which contain knowledge that is not available in the software architecture artifact: not selected solutions and choice + rationale. On the other hand, the results of the design process (the architecture and architectural modifications), are missing in the rationale management process.

The concept of first-class represented design decisions, composed of rationale, architectural modifications, and alternatives, is used to bring the two processes together. A software architecture design process no longer results in a static design description of a system, but in a set of design decisions leading up to the system. The design decisions reflect the rationale used for the decision making process, and form the natural bridge between rationale and the resulting architecture.

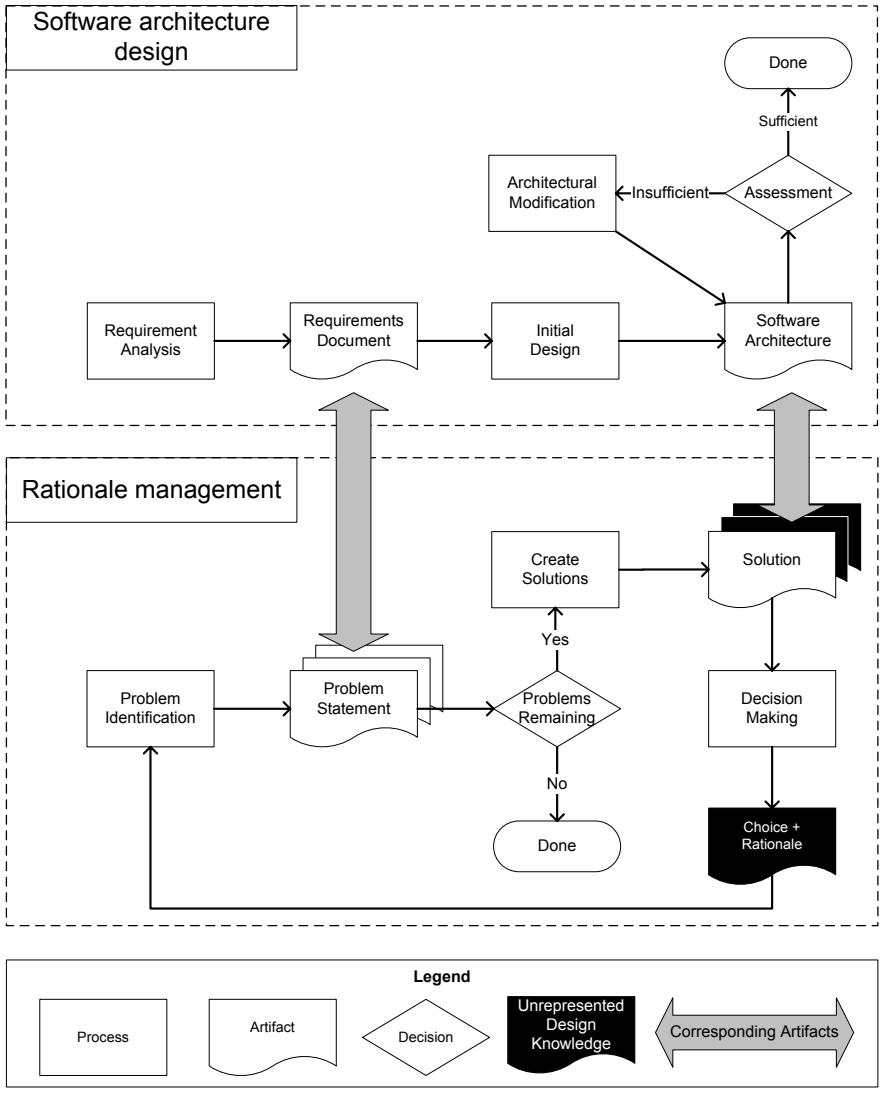

Figure 3.3: Similarities between software architecture design process and the rationale management process

3.4.2 CD player: a Design Decision Example

This subsection presents a simple case, which shows the impact of designing an architecture with design decisions. The example is based on the design of a compact disc (CD) player. Changing customers' needs have made the software architecture of the CD player insufficient. Consequently, the architecture needs to evolve.

The software architecture of the CD player is presented in the top of figure 3.4, the current design. The design decisions leading to the current design are not shown in figure 3.4 and are instead represented as one design decision.

The CD player's architecture is visualized in a component and connector view [29]. The components are the principal computational elements that execute at run-time in the CD player. The connectors represent which component has a run-time pathway of interaction with another component.

Two functional additions to the software architecture are described. First, a software-update mechanism is added. This is used to update the CD player, to make it easier to fix bugs and add new functionality in the future. Second, the internet connection is used to download song information for the played CD, like lyrics, additional artist information, etc.

As shown in figure 3.4, design decisions are taken to add the described functionality. The design decisions contain the rationale and the functional solution, represented as documentation and an architectural component and connector view. Note, that the rationale in the picture is shortened very much because of space limitations. The added functionality is directly represented by two design decisions, *Updater* and *SongDatabase*.

The first idea for solving the internet connectivity was to add a component which handled the communication to the Patcher. This idea was rejected, and another alternative was considered, to create a change to the Hardware Controller. This change enabled the internet connectivity for the Internet song db too, and was considered better because it could reuse a lot of the functionality of the existing Hardware Controller. Note that the view on the current design shows a complete architecture, while it is also a set of design decisions. The resulting design (figure 3.5) is visualized with the two design decisions taken: the *Updater* and the *SongDatabase*.

3.4.3 Design decisions

In the example of section 3.4.2, the software architecture of the CD player is the set of design decisions leading to a particular design, as depicted in 3.4. In the classical

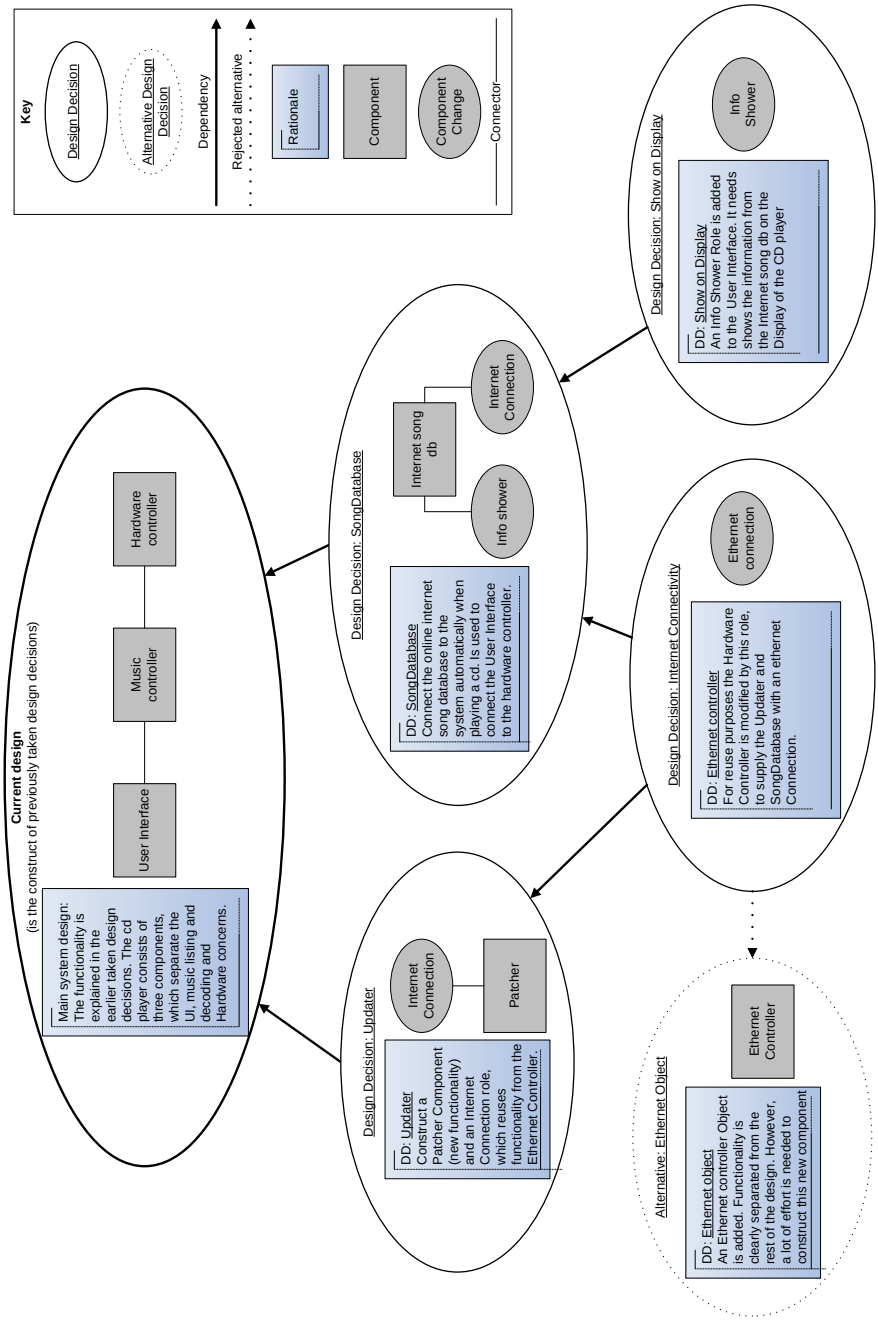

Figure 3.4: The architecture of a CD player with extended functionality

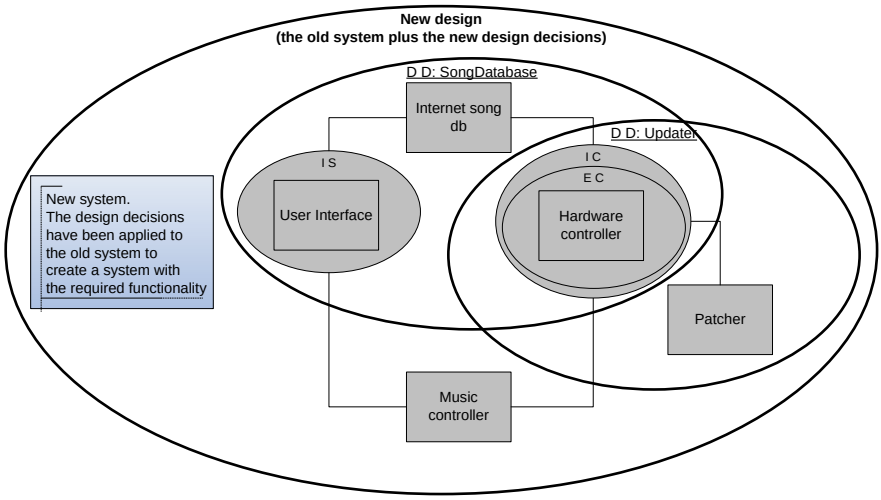

Figure 3.5: The result of the design decisions of figure 3.4

notion of system design only the result depicted in figure 3.5 is visible while not capturing the design decisions leading up to a particular design.

Although the term architectural design decision is often used [11, 29, 67], a precise definition is hard to find. Therefore, we define an architectural design decision as:

> A description of the choice and considered alternatives that (partially) realize one or more requirements. Alternatives consist of a set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements.

We detail this definition by describing the used elements:

- The *considered alternatives* are potential solutions to the requirement the design decision addresses. The *choice* is the decision part of an architectural design decision; it selects one of the considered alternatives. For example, figure 3.4 contains two considered alternatives for the connectivity design decisions. The Ethernet Object alternative is not selected. Instead, the Internet Connectivity is selected.

- The *architectural additions, subtractions*, and *modifications* are the changes to the given architecture that the design decision makes. For example, in figure 3.4 the Song Database design decision has one addition in the form of a

new component (the Internet Song Database), and introduces two modifications to components (info shower and internet connection).

- The *rationale* represents the reasons behind an architectural design decision. In figure 3.4, the rationale is briefly described within the design decisions.
- The *design rules* and *constraints* are prescriptions for further design decisions. As an example of a rule, consider a design decision that is taken to use an object-oriented database. All components and objects that require persistence need to support the interface demanded by this database management system, which is a rule. However, this design decision may require that the complete state of the system is saved in this object-oriented database, which is a constraint.
- Timely fulfillment of *requirements* drives the design decision process. The requirements not only include the current requirements, but also include requirements expected in the future. They can be either explicit, e.g. mentioned in a requirements document, or implicit.
- A design decision may result in *additional requirements* to be satisfied by the architecture. Once a design decision is taken, new insights can lead to previous undiscovered requirements. For instance, the design decision to use the Internet as an interface to a system will cause security requirements like logins, secure transfer etc.

The given *architecture* is a set of earlier made design decisions, which represent the architectural design at the moment the design decision is taken.

Architecture design decisions may be concerned with the application domain of the system, the architectural styles and patterns used in the system, COTS components and other infrastructure selections as well as other aspects described in classical architecture design. Consequently, architectural design decisions can have many different levels of abstraction. Furthermore, they involve a wide range of issues, from pure technical ones to organizational, business, political, and social ones.

3.4.4 Designing with design decisions

Existing design methods (e.g. [11] and [19]) describe ways in which alternatives are elicited and trade-offs are made. An architect designing with design decisions still uses these design methods. The main difference lies in the awareness of the architect, to explicitly capture the design decisions made and the associated design knowledge.

Section 3.2.3 presented key problems in software architecture. Designing with design decisions helps in handling these problems in the following way:

- **Design decisions are cross cutting and intertwined.** When designing with design decisions the architect explicitly defines design decisions, and the relationships between them. The architect is made aware of the cross cutting and intertwining of design decisions. In the short term, if the identified intertwining and cross cutting is not desirable, the involved design decisions can be reevaluated and alternative solutions can be considered before the design is further developed. In the long term, the architect can (re)learn which design decisions are closely intertwined with each other and what kind of problems are associated with this.
- **Design rules and constraints are violated.** Design decisions explicitly contain knowledge about the rules and constraints they impose on the architecture. Adequate tool support can make the architect aware about these rules and constraints and provide their associated rationale. This is mostly a long term benefit to the architect, as this knowledge is often forgotten and no longer available during evolution or maintenance of the system.
- **Obsolete design decisions are not removed.** In evolution and maintenance, explicit design decisions enable identification and removal of obsolete design decisions. The architect can predict the impact of the decision and the effort required for removal.

Designing with design decisions requires more effort from the architect, as the design decisions have to be documented along with their rationale. In traditional design, the architect forms the bridge between architecture and rationale. In designing with design decisions, this role is partially taken up by the design decisions.

Capturing the rationale of design decisions is a resource intensive process [42]. To minimize the capture overhead, close integration between software architecture design, rationale, and design decisions is required. The following section presents an example of an approach that demonstrates this close integration.

3.5 Archium

The previous section presented a general notion of architectural design decisions. In this section, a concrete example realization of this notion is presented: Archium [77]. First, the basic concepts of Archium are presented, after which this approach is illustrated with an example.

3.5.1 Basic concepts of Archium

Archium is an extension of Java, consisting of a compiler and run-time platform. Archium consists of three different elements that are integrated with each other. The first element is the architectural model, which formally defines the software architecture using ADL concepts [107]. Second, Archium incorporates a decision model, which models design decisions along with its rationale. Third, Archium includes a composition model, which describes how the different concepts are composed together.

The focus in this subsection is on the design decision model. For the composition and architectural model see [77]. The decision model (see figure 3.6) uses an issue-based approach [101]. The issues are problems that the solutions of the architectural design decisions (partially) solve. The rationale part of the decision model focuses on *design decision rationale* and not *design rationale* in general (see the 'DRL' section in chapter 1 of [42]).

Archium captures rationale in customizable rationale elements. They are described in natural text within the scope of a design decision. Rationale elements can explicitly refer to elements within this context, therefore creating a close relationship between rationale and design elements.

The motivation and cause elements provide rationale about the problem. The choice element chooses the right solution and makes a trade-off between the solutions. The choice results in an architectural modification.

To realize the chosen solution in an architectural design decision, the components and connectors of the architectural model can be altered. In this process, new elements might be required and existing elements of the design might be modified or removed. The architectural modification describes these changes, and thereby the history of the design. These architectural modifications are explicitly part of design decisions, which are first-class entities in Archium. This makes Archium capable of describing a software architecture as a set of design decisions [77].

Rationale acquisition (see chapter 1 of [42]) is a manual task in Archium. The approach tries to minimize the intrusiveness of the capturing process by letting the rationale elements of the design decisions be optional. The only intrusive factor is the identification and naming of design decisions.

The rationale elements are to a certain extend similar to that of DRL [101]. The *Problem* element is comparable to a *Decision Problem* in DRL. A *Solution* solves a *Problem*, likewise *Alternatives* do in DRL. The *Motivation* element gives more rationale about the *Problem* and is comparable to a supportive *Claim* in DRL. A

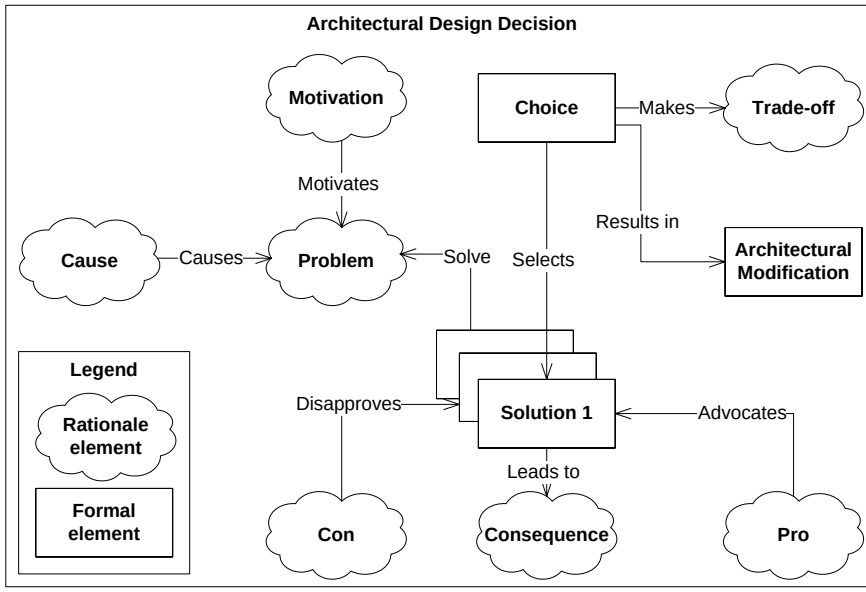

Figure 3.6: The Archium design decision model

Cause can be seen as a special instance of a *Goal* in DRL. The *Consequence* element is like a DRL *Claim* about the expected impact of a *Solution*. The *Pro* and *Con* elements are comparable to supporting and denying DRL *Claims* of a *Solution* (i.e. a DRL *Alternative*).

3.5.2 Example in Archium

An example of a design decision and the associated rationale in Archium is presented in figure 3.7. It describes the *Updater* design decision of figure 3.4. Rationale elements in Archium start with an @, which expresses rationale in natural text. In the rationale, any design element or requirement in the scope of the design decision can be referred to using square brackets (e.g. [iuc:patcher]). In this way, Archium allows architects to relate their rationale with their design in a natural way.

A design decision can contain multiple solutions. Each solution has a realization part, which contains programming code that realizes the solution. A realization can use other design decisions or change existing components. In the *InternetUpdate solution* the realization contains the *InternetUpdateChange*, which defines the *Patcher* component and the component modifications for the *Internet Connection* (see figure 3.4). The *IUCMapping* defines how the *InternetUpdateChange* is mapped onto the current *design*, which is an argument of the design decision.

```
design decision Updater(CurrentDesign design) {
  @problem {# The CD player should be updatable.[R4] #}
  @motivation {# The system can have unexpected bugs or require
                 additional functionality once it is deployed. #}
  @cause {# Currently this functionality is not present in the
            [design], as the market did not require this functionality
             before. #}
  @context {# The original [design]. #}

  potential solutions {
    solution InternetUpdate {
      architectural entities {
        InternetUpdateChange iuc;
        IUCMapping iucMapping;
      }
      @description {# The system updates itself using a patch, which
                      is downloaded from the internet. #}
      realization {
        iuc = new InternetUpdateChange();
        iucMapping = new IUCMapping(design,iuc);
        return design composed with iuc using iucMapping;
      }
      @design rules {# When the [iuc:patcher] fails to update, the
                       system needs to revert back to the previous state. #}
      @design constraints {# #}
      @consequences {# The solution requires the system to have a
                       [iuc:internetConnection] to work. #}
      pros {
        @pro {# Distribution of new patches is cheap, easy, and fast #}
      }
      cons {
        @con {# The solution requires a connection to the internet to
                work. #}
      }
    }
  /* Other alternative solutions can be defined here */
  }
  choice {
    choice InternetUpdate;
    @tradeoff {# No economical other alternatives exist #}
  }
}
```

Figure 3.7: The Updater design decision in Archium

To summarize, the architectural design decisions contain specific rationale elements of the architecture, thereby not only describing how the architecture has become what it is, but also the reasons behind the architecture. Consequently, design decisions can be used as a bridge between the software architecture and its rationale. The Archium environment shows that it is feasible to create architectures with design decisions.

3.6 Related work and further developments

This section describes related and future work. The related work focuses on software architecture, as the related work about rationale management is explained in

more depth in previous chapters of this book. After this, subsection 3.6.2 describes future work on design decisions.

3.6.1 Related work

Software architecture design methods [11, 19] focus on describing how the right design decisions can be made, as opposed to our approach which focuses on capturing these design decisions. Assessment methods, like ATAM [11], asses the quality attributes of a software architecture, and the outcome of such an assessment steers the direction of the design decision process.

Software documentation approaches [29, 67] provide guidelines for the documentation of software architectures. However, these approaches do not explicitly capture the way to and the reasons behind the software architecture.

Architectural Description Languages (ADLs) [107] do not capture the road leading up to the design either. An exception is formed by the architectural change management tool Mae [132, 161], which tracks changes of elements in an architectural model using a revision management system. However, this approach lacks the notion of design decisions and does not capture considered alternatives or rationale about the design.

Architectural styles and patterns [138] describe common (collections of) architectural design decisions, with known benefits and drawbacks. Tactics [11] are strategies for design decision making. They provide clues and hints about what kind of design decisions can help in certain situations. However, they do not provide a complete design decision perspective.

Currently, there is more attention in the software architecture community for the decisions behind the architectural design. Kruchten [93], stresses the importance of design decisions, and creates classifications of design decisions and the relationship between them. Tyree and Akerman [158] provides a first approach on documenting design decisions for software architectures. Both approaches model design decisions separately and do not integrate them with design. Closely related to this is the work of Lago [99], who models assumptions on which design decisions are often based, but not the design decisions themselves.

Integration of rationale with the design is also done in the design rationale field. With the SEURAT [24] system, rationale can be maintained in a RationaleExplorer, which is loosely coupled to the source code. This rationale has to be added to the design tool, to let the rationale of the architecture and the implementation be maintained correctly. DRPG [9] couples rationale of well-known design patterns

with elements in a Java implementation. Likewise SEURAT, DRPG also depends on the fact that the rationale of the design patterns is added to the system in advance.

3.6.2 Future work

The notion of design decisions as first-class entities in a software architecture design raises a couple of research issues. Rationale capture is very expensive, so how can we determine which design decisions are economically worth capturing? So far, we have assumed that all the design decisions can be captured. In practice, this would often not be possible or feasible. How do we deal with the completeness and uncertainty of design decisions? How can we support addition, change, and removal of design decisions during evolution?

First, design decisions need to be adapted into commonly used design processes. Based on this, design decisions can be formalized and categorized. This will result in a thorough analysis of the types of design decisions. Also, dependencies need to be described between the requirements and design decisions, between the implementation and design decisions and between design decisions among themselves.

Experiments by others have already proven that rationale management helps in improving maintenance tasks. Whether the desired effects outweigh the costs of rationale capturing is still largely unproven. The fact that most of the benefits of design decisions will be measurable after a longer period when maintenance and evolution takes place complicates the validation process. We are currently working on a case study which focuses on a sequence of architectural design decisions taken during evolution. Additional industrial studies in different domains are planned in the context of an ongoing industrial research project, which will address some of the aforementioned questions.

3.7 Summary

This chapter presented the position of rationale management in software architecture design. Rationale is widely accepted as an important part of a software architecture. However, no strict guidelines or methods exist to structure this rationale. This leaves the rationale management task in the hands of the individual software architect, which makes it hard to reuse and communicate this knowledge. Furthermore, rationale is typically kept separate from architectural artifacts. This makes it hard to see the benefit of rationale and maintaining it.

Giving design decisions a first-class representation in the architectural design creates the possibility to include problems, their solutions, and the rationale of these decisions into one unified concept. This chapter described an approach in which decisions behind the architecture are seen as the new building blocks of the architecture. A first step is made by the Archium approach, which illustrated that designing an architecture with design decisions is possible. In the future, we think that rationale and architecture will be used together in design decision-like concepts, bridging the gap between the rationale and the architecture.

Acknowledgements

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

"Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled."

– Eoin Woods [145]

CHAPTER 4

SOFTWARE ARCHITECTURE AS A SET OF ARCHITECTURAL DESIGN DECISIONS

Published as: Anton Jansen, Jan Bosch. Software Architecture as a Set of Architectural Design Decisions, Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005), pp. 109-119, November 2005.

Abstract

Software architectures have high costs for change, are complex, and erode during evolution. We believe these problems are partially due to knowledge vaporization. Currently, almost all the knowledge and information about the design decisions the architecture is based on are implicitly embedded in the architecture, but lack a first-class representation. Consequently, knowledge about these design decisions disappears into the architecture, which leads to the aforementioned problems. In this paper, a new perspective on software architecture is presented, which views software architecture as a composition of a set of explicit design decisions. This perspective makes architectural design decisions an explicit part of a software architecture. Consequently, knowledge vaporization is reduced, thereby alleviating some of the fundamental problems of software architecture.

4.1 Introduction

Software architecture [119] has become a generally accepted concept in research and industry. The importance of stressing the components and their connectors of

a software system is generally recognized and has led to better control over the design, development, and evolution of large and increasingly dynamic software systems [11].

Although the achievements of software architecture are formidable, still some problems remain. The complexity, high costs of change, and design erosion are some of the fundamental problems of software architecture. We believe these problems are partially due to knowledge vaporization. Currently, almost all the knowledge and information regarding the design decisions on which the architecture is based (e.g. results of domain analysis, architectural styles used, trade-offs made etc.) are implicitly embedded in the architecture, but lack a first class representation.

The current perspective on software architecture lacks this notion of architectural design decisions, although architectural design decisions play a crucial role in software architecture, e.g. during design, development, evolution, reuse and integration of software architectures. In design, the main concern is which design decision to make. In development, it is important to know which and why certain design decisions have been taken. Architecture evolution is about making new design decisions or removing obsolete ones to satisfy changing requirements. The challenge is to do this in harmony with the existing design decisions. Reuse of software architecture is the use of earlier tried and tested combinations of design decisions (e.g. design patterns or components). In the integration of systems, the main concern is the unification of the design decisions and their assumptions.

To address this, we propose a new perspective on software architecture: we define software architecture as the composition of a set of architectural design decisions. This reduces the knowledge vaporization of design decision information, since design decisions have become an explicit part of the architecture.

The contribution of this paper is threefold. First, the problems with the current perspective on software architecture are presented. Second, it develops the notion of software architecture as the composition of a set of explicit architectural design decisions. Third, various views are presented for visualizing this new architecture perspective.

The remainder of this paper is organized as follows. The concept of architectural design decisions is presented in section 4.2. In section 4.3, the problems of software architecture with respect to architectural design decisions are explained in more depth. The next section introduces Archium: our approach for describing software architecture as a set of architectural design decisions. The approach is applied to a case and illustrated with various views on design decisions in section 4.6. After this, related work is discussed. The paper concludes with future work and conclusions in section 4.8.

4.2 Architectural design decisions

Although the term "architectural design decision" is often used [11, 29, 67], a precise definition is hard to find. Therefore, we define an architectural design decision as:

> A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture.

With the definition of architectural design decisions using the following elements:

- **Rationale** The reasons behind an architectural design decision are the rationale of an architectural design decision. It describes *why* a change is made to the software architecture.
- **Design rules** and **design constraints** are prescriptions for further design decisions. Rules are mandatory guidelines, whereas constraints limit the design to remain sound.
- **Design constraints** Design constraints describe the opposite side of design rules. They describe what is not allowed in the future of the design, i.e. they prohibit certain behaviors.
- **Additional requirements** A design decision may result in additional requirements to be satisfied by the architecture. These new requirements need to be addressed by additional design decisions.

An architectural design decision is therefore the outcome of a design process during the initial construction or the evolution of a software system. Architectural design decisions, among others, may be concerned with the application domain of the system, the architectural styles and patterns used in the system, COTS components and other infrastructure selections as well as other aspects needed to satisfy the system requirements.

We propose to view a software architecture as a set of explicit architectural design decisions. In this perspective, the software architecture is the *result* of the architectural design decisions made over time.

4.3 Problems of software architecture

The current perspective on software architecture lacks a clear view on why the architecture looks as it does [21, 93]. In the current notion of a software architecture, the results of the design decisions underlying the architecture are implicitly embedded within the architecture. Consequently, knowledge about the design decisions underlying the architecture is lost [158]. This vaporization of design decision information leads to a number of problems associated with software architecture:

- **Design decisions are cross cutting and intertwined:** Design decisions are often intertwined with each other, as they work in close relationship together. Furthermore, they typically affect multiple parts of the design simultaneously. This leads to the situation that the design decision information is fragmented across various parts of the design, making it hard to find and change the decisions. Both effects increase the overall complexity of the software architecture, as numerous seemingly unrelated relationships (e.g. dependencies) between architectural entities are introduced.
- **Design rules and constraints are violated:** During the evolution of the system, designers can easily violate design rules and constraints arising from previously taken design decisions. Violations of these rules and constraints lead to architectural drift [119] and its associated problems (e.g. increased maintenance costs). As design rules and constraints influence future design decisions, they have a steering influence on the future direction of the architecture.
- **Obsolete design decisions are not removed:** When obsolete design decisions are not removed, the system has the tendency to erode more rapidly. In the current design practice, removing design decisions is avoided, because of the effort needed, and the unexpected effects this removing can have on the system.

As a result of these problems, the developed systems have a high cost of change, and they tend to erode quickly. Also, the reusability of the architectural artifacts is limited if design decision knowledge vaporizes into the design. These problems are caused by the focus in the software architecture design process being on the resulting artifacts, instead of the decisions that lead to them. Although the effects of the decisions made are present in the design, the decisions themselves are not visible. Clearly, design decisions currently lack a representation in software architecture designs.

Defining software architecture as a set of architectural design decisions is a step forward in solving the aforementioned problems. This would also help the architect with:

- **Guarding the conceptual integrity** of the software architecture. The design decisions describe the rules and constraints that should be obeyed. In current practice, software engineers and architects are often unaware of the conceptual integrity that they break of the architecture. Explicit design decisions help in creating the necessary awareness and reference points for these constraints and rules.
- **Explicit design space exploration** helps the architect in preventing obvious mistakes. It forces the architect to reflect on the software architecture. Furthermore, it enables communication of the explored design space with others.
- **Analysis** of both the software architecture and the design process. For example, in evolution the architect wants to play "what if" scenarios of considered design decisions in the context of existing ones.
- **Improved traceability** of the design decisions and their relationship to features, design aspects, concerns, and among themselves. This helps the architect with obtaining a better understanding of the software architecture.

However, the following requirements need to be satisfied to realize this:

- **First class architectural design decisions** are required to describe a software architecture as a set of design decisions. Furthermore, first class design decisions can be communicated, related and reasoned about. This provides information about the architecture, which is currently often missed.
- **Explicit architectural changes** form the bridge between the first class architectural entities and the architectural design decisions. This is needed to have a well-defined relationship between the proposed solutions of an architectural decision and the involved architectural entities.
- **Support for modification, subtraction, and addition** changes are required to have sufficient expressiveness. The characteristic types of change often distinguished are the corrective, perfective, and adaptive types. However, the focus of this classification is on the reasons behind the change, not on the effect of the changes.
- **Clear, bilateral relationship between architecture and realization** Viewing a software architecture as a set of design decisions, makes evolution an inherent part of the description of an architecture. Changes in the architecture will have an effect on the realization of the system and vice versa. It is therefore important to have a bilateral relationship between the software architecture and the realization.
- **First class architectural concepts** As software architecture deals with abstractions, it is important to define these abstractions in a first class way. Abstraction choices are very subjective and greatly influence the resulting architecture.

Expressing these abstractions in a consistent and uniform way is therefore essential for software architectures.

4.4 Archium

The aforementioned problems of section 4.3 clearly show that the notion of an architectural design decision is an important one. Currently, no models for representing architectural design decisions exist [76]. Some general design decision models [128] do facilitate the description of abstract elements of an architectural design decision model, but these approaches fail to satisfy most of our requirements [76].

This is mainly due to the ill-defined relationship between these design decision models and software architectures. Therefore, we have developed an approach called Archium, which tries to define this relationship. Archium maintains this relationship during the complete life-cycle of the system. In this paper, the focus is on the software architecture aspects of Archium. The approach is based on a conceptual architectural design decision model, which describes the elements of architectural design decisions and their relationships in greater detail than the abstract design decision models. In the remainder of this section, this conceptual model is presented.

4.4.1 Architectural design decision model

Figure 4.1 presents our conceptual model for architectural design decisions. At the heart of the model is the *Problem* element, which together with the *Motivation* and *Cause* elements describes the problem, a *Motivation* as to why the problem is a problem, and the *Cause*s of this problem. The *Problem* is the goal the architectural design decision wants to solve. The solutions element contains the *Solution*s that have been proposed to solve the problem at hand. A *Decision* is made, which solution should be used, resulting in an *Architectural modification* that changes the *Context*.

To solve the described *Problem*, one or more potential *Solutions* can be thought up and proposed. For each of the proposed solutions, we define the following elements (which are not shown in the figure 4.1) :

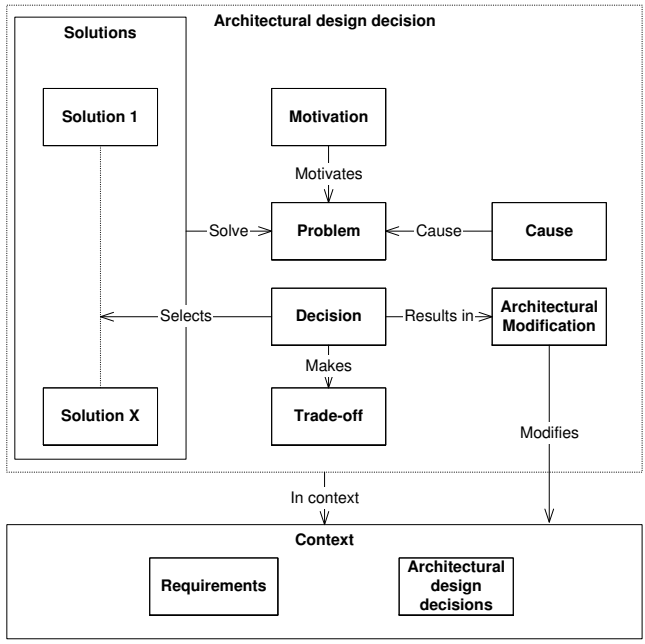

Figure 4.1: Model for architectural design decisions

- **Description** The description element describes the solution being proposed. The needed modifications are explained and rationale for these modifications is provided.
- **Design rules** A potential solution can have one or more design rules. Design rules define partial specifications to which the realization of one or more architectural entities has to conform. This allows a solution to define parts of how it should be realized in order to have a solution that solves the problem.
- **Design constraints** Besides design rules, a solution can have design constraints. Design constraints define limitations on the further design of one or more architectural entities. These constraints have to be obeyed for the potential solution to solve the problem at hand.
- **Consequences** The consequences element is a description of the expected consequences of the solution on the architecture. The element should provide additional rationale behind the pros and cons of the solution.
- **Pros** This model element describes the expected benefit(s) from this solution to the overall design and the impact on the requirements.
- **Cons** Solutions can also have a downside. The expected negative impact on the overall design is as important as the positive side.

Translating the conceptual model into concrete model(s) is a big challenge. Our earlier investigation [76] revealed a gap between design decisions and software architecture models. Therefore, the rest of this paper focuses on the interaction between architectural design decisions and software architecture. Specifically, we discuss how the *decision*, *solution*, *architectural modification*, *software architecture*, and *architectural design decisions* conceptual model elements are modeled and formalized to describe a software architecture as a set of design decisions.

4.5 Archium meta-model

In Archium, the functionality of the architectural modification is expressed as a *change* in functionality. New functionality is regarded as the change of nothing to something. In this perspective, Archium is fundamentally different to most other design methods, as it does not promote design for or with change, but rather designing *using* change.

A software architecture in Archium is described as a set of changes, which together form the software architecture. To be more precise, in Archium a $software\ archictecture = dd_1 + dd_2 + ... + dd_n$, where dd_x is a design decision. The exact elements required to achieve this are described in the rest of this section.

The Archium approach is based on a meta-model, which describes the central concepts of our approach and their relationships. Figure 4.2 presents this meta-model. The model consists of three sub-models: an architectural model, a design decision model, and a composition model. The architectural model defines software architecture concepts, which are similar to the concepts used in existing architecture models [107]. The design decision model contains design decisions as a first class concept. The composition model introduces the model elements needed to unite the two previous sub-models. Each sub-model is explained in more detail in the remaining part of this section.

A (trivial) running example of a subsystem of a measurement system exemplifies the different concepts. The measurement system acquires certain properties of a physical item that enters the system for measurement. The architecture of this system is visualized in the top of figure 4.4.

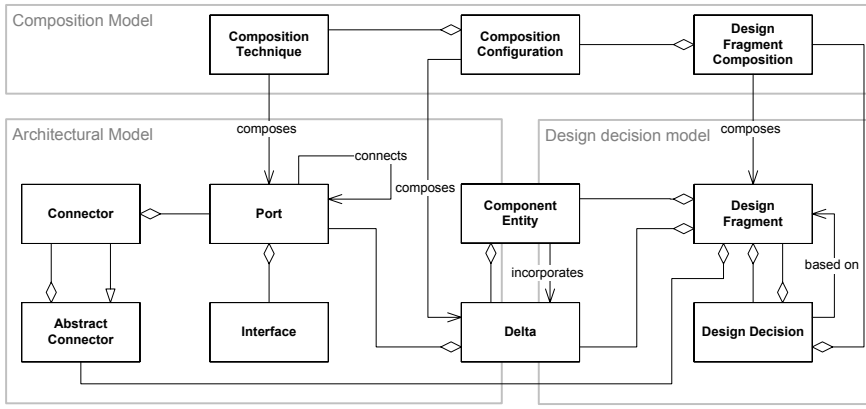

Figure 4.2: Meta-model of a software architecture with first class design decisions

4.5.1 Architectural Model

The architectural model part of the Archium meta-model uses concepts of the Component & Connector view [29]. The relationships of these concepts is visualized in figure 4.2. Following is a more in-depth description of these concepts and their relationships:

Component Entity is an abstraction of a component. A component entity describes the decomposition aspect of a component. The functionality of a component entity is not defined in the component entity itself, but in the deltas related to the component entity. A component in Archium is a specific *instance* of a component entity with known functionality, i.e. the deltas incorporated in the component entity instance are known.

For example, in the measurement system (see top of figure 4.4) a decomposition has been made in two parts, which are made up of the *Measurement Item* and the *Sensor* component entities. The *Measurement Item* component entity represents the object to be measured and the *Sensor* measures some properties of the measured object.

Delta is the first-class representation of a change to the behavior of a component entity, which is provided by the deltas already incorporated in the component entity. A component entity incorporating a delta includes the modification of the delta to its behavior. The merging of the behavior of different deltas is performed using the elements of the composition model (see section 4.5.3).

In the example, the functionality of the *Measurement Item* and *Sensor* components are defined in the *SensorDelta* and *MIDelta* deltas. The *SensorDelta* contains the

functionality to measure an item and the *MIDelta* has the functionality to store these measurements. These changes are not visible in figure 4.4, as they are being incorporated into the components.

Interface A definition of a collection of method signatures, representing a specific named semantic.

Port An external visible interface required or provided by a delta or connector. A port represents the provided or required "service" of a delta or connector. Deltas and connectors are only allowed to communicate with others through their defined ports. Ports of a delta and connector can be connected together, to form a connection, thereby creating a specific configuration of deltas and connectors.

In the measurement system (see figure 4.4), two ports are defined: a provided port for the *Sensor* and a required port for the *Measurement Item*. Neither port is defined in the component entities, but instead is part of the deltas incorporated in the components.

Connector defines the "glue" between one or more deltas, i.e. a connector is a first class representation of the interaction or communication between these deltas. The ports of a connector can be bound to the provided and required ports of deltas, thereby forming the "glue" between them. A connector therefore defines the specific functionality used for the communication between connected ports.

In the example, the communication between the *Sensor* and the *Measurement Item* is defined in the connector *CMISensor*.

Abstract Connector is an abstraction from a Connector, as it does not have an interface associated with it. It defines the communication type (i.e. synchronous, asynchronous) between two or more deltas. In addition, it defines a set of connectors that actually communicate between these deltas.

The abstract connector used in our example defines that the *SensorDelta* and *MIDelta* communicate using method invocation with each other. In addition, the abstract connector contains the connector *CMISensor* connecting the two deltas.

4.5.2 Design Decision Model

In this subsection, the design decision model part of Archium is presented. The relationship between architectural design decisions and the architectural concepts is defined using the concept of a design fragment. Following is a more in-depth description of both concepts:

Design Fragment is an architectural fragment defining a collection of architectural entities. An architectural entity can be part of multiple design fragments. A

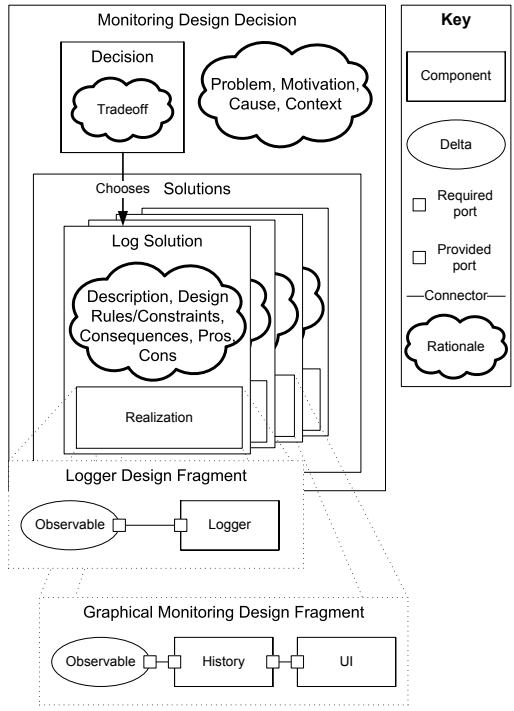

Figure 4.3: Example of a design decision

design fragment is a boundary-less container for maintaining traceability. The primary use is to define the scope of a solution of a design decision. For example, a design fragment can contain deltas and their configuration of (abstract) connectors, component entities incorporating certain deltas, and other design fragments, which describe a particular solution. A design fragment is therefore a (partial) description of the system, which can include explicit change (modeled as deltas) and structure, i.e. component entities and (abstract) connectors.

In Archium, the concept of a software architecture and a design pattern are specializations of the design fragment concept. The first class concept of a design fragment makes these two concepts elements in the Archium model. A software architecture is a design fragment describing the system as a whole. This description consists of the component entities and (abstract) connectors and their configuration, which is a specialized subset of the architectural entities a design fragment can contain.

For example, the architecture of the measurement system itself is a design fragment. Figure 4.4 visualizes this, with the architectural entities (*Measurement Item*, *CMISensor*, *Sensor*) being enclosed within the *SensorFragment*.

Design patterns [49] are often seen as predefined design decisions, which is not completely true. They define predefined *parts* of design solutions, which can be reused. However, design patterns still need to be instantiated and configured in a design decision to be of use in a specific architecture.

Design Decision is a first class concept in Archium. It defines the solutions considered and the one decided upon (i.e. the decision) to solve a described problem (see figure 4.3). A software architecture (i.e. a design fragment) describes the context in which this design decision is made. The considered potential solutions consist of one or more design fragments, which act upon this context design fragment by changing it according to the selected solution.

Figure 4.3 presents an example of a design decision. It consists of the rationale described in section 4.4.1, a decision element, and one or more solution elements. Each solution contains its own rationale and a realization part. The realization is a design fragment, which is mapped onto a design fragment representing the architecture. The mapping is explained in more detail in the next subsection.

A design decision is regarded as a change function within Archium. It has optional parameters, which are the design fragments describing the context that the design decision changes. Applying a design decision on these context design fragments results in a new design fragment, which includes the design decision it originated from. This explains the mutual relationship between a design fragment and design decision in the Archium meta-model (figure 4.2).

An example of the application of a design decision is provided in figure 4.4. In this case, the measurement system needs to be changed to allow monitoring of the activities within the system. The design decision is made to change the *SensorFragment* to include a logger. This design decision is presented in figure 4.3. The logger logs the actions of the *Measurement Item* on the *Sensor*. This modification is defined in the *LoggerFragment*, which is another design fragment. The composition of the design fragments, as a means to change the measurement system, is described in the next subsection.

4.5.3 Composition Model

The composition model is responsible for relating the changes of the design decision model to the elements of the architectural model. It defines the way in which a delta interacts with other composed deltas. In Archium, the following first class citizens are concerned with describing this:

- **Composition Technique** describes the way in which a delta changes a port of a component entity. For example, it can define that a delta introduces a new

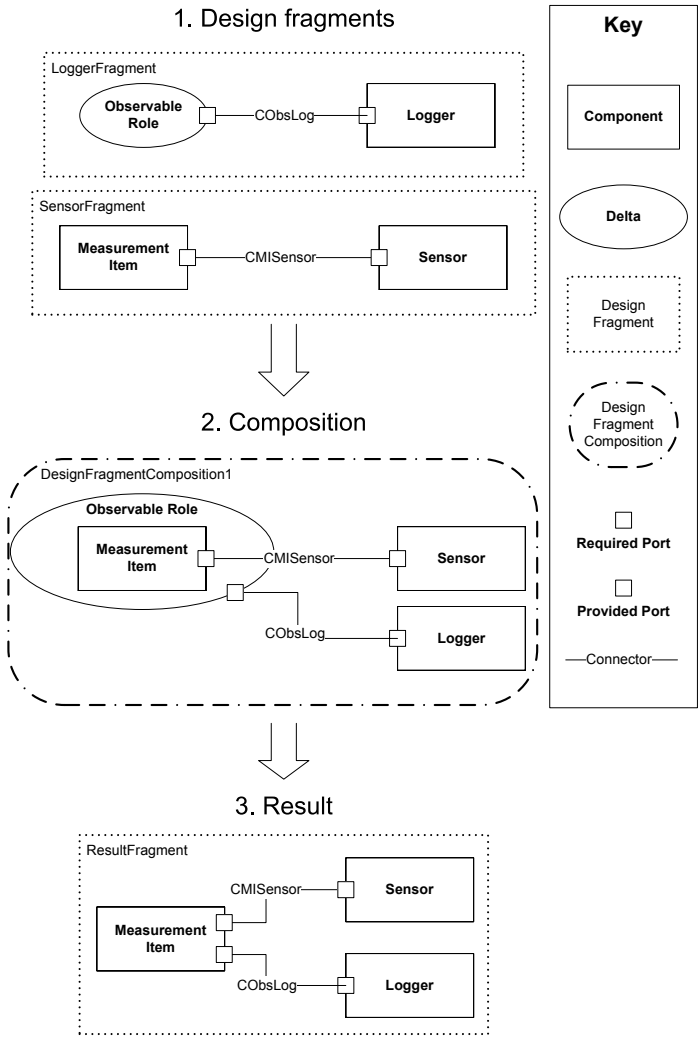

Figure 4.4: Example of the composition of two design fragments

port to the component entity, subtracts, or modifies an existing one. Examples of composition techniques include: Inheritance, Delegation, Replacement, and Adapter (adapt a port interface).

For example, in figure 4.4 a composition technique is used to describe how the provided port of the *ObservableDelta* reacts on the activities on the required port of *MeasurementItem*.

- **Composition Configuration** specifies how a component entity incorporates a certain delta. The composition configuration uses composition techniques to specify the change on a per-port basis. In this sense, the composition configuration is nothing more than a set of composition techniques to describe the way in which a delta changes a component entity.
- **Design Fragment Composition** is used to define how a design fragment can change another design fragment. It uses composition configurations and design fragments to relate architectural entities of one design fragment to another, thereby creating a new, changed design fragment.

  For the example of figure 4.4, the design fragment composition composes the *LoggerFragment* and *SensorFragment*. It uses a composition configuration to relate the *ObservableDelta* with the *MeasurementItem*.

4.6 Athena case

The previous section introduced the Archium meta-model and illustrated parts of it in a trivial example. In this section, we validate our approach by applying it to a case. First, the case is introduced, after which two design decisions are presented in more detail.

4.6.1 Introduction

Athena is a submission system for (automatic) judgement, review, manipulation, and archival of computer program sources. The primary use is supporting students in learning programming. To develop the programming skills of a student, he or she has to practice a lot. Small programming exercises are often used for this end. However, providing feedback on these exercises is laborious and time-consuming. Athena helps students (and teachers) by testing their solutions to functional correctness and provides feedback (e.g. test results, test inputs, compilation information etc.) on this.

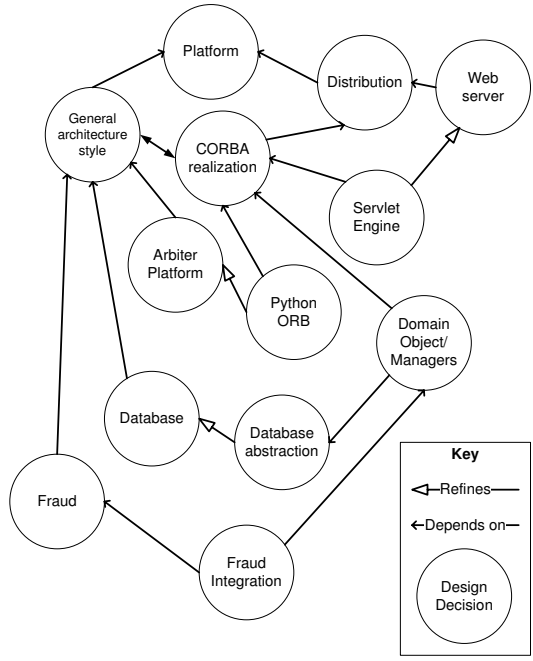

Figure 4.5: Design decisions of Athena

The architecture of Athena (called "Original design") is illustrated on the top of figure 4.6. Athena uses a three tier architectural style consisting of *Database*, *Middleware*, and *Client* components. The *Middleware* component consists of a *Connection Broker*, which provides database abstraction and connection handling. A *Domain Object* represents the different specific domain objects used in Athena (e.g. Student, Submission, Assignment etc.). A *Manager* provides query and instantiation services for the *Client* and *Domain Object* components. In the *Client* component, students submit their work with the help of the *Submission Client*. A *Arbiter* tests this work and students can view the results with the *Student Web Interface*. Teachers and their assistants configure Athena with the help of a *Management Tool*.

4.6.2 Design decisions

The software architecture of Athena is the result of multiple design decisions. Figure 4.5 presents a part of these design decisions in a design decision dependency view. An architect would like to navigate between this view and other views on the

architecture. The view allows for the management of the dependencies among design decisions. For example, "what if" scenarios can be played, where the impact of potential design decisions is examined.

The focus in the remainder of this section is on one dependency between two design decisions, as space constraints hinder a more elaborate description. Based on the different elements of Archium's design decision model (see section 4.4.1) the *Fraud* and *Fraud Integration* design decision are described. Both design decisions are made after the initial deployment of the system. They are described as follows:

Fraud design decision

Problem Some of the students don't create solutions for the exercises themselves, but rather copy the work of their fellow students. **Motivation** A result of this is that the students don't obtain an adequate programming experience, which is required for more advanced courses. **Cause** The large number of students (100+) in courses where Athena is used leads to anonymity and a small chance to get caught. On top of this, the high pressure resulting from the strict deadlines imposed by the system increases the temptation to commit fraud. **Context** The original design of the Athena system, as depicted on the top of figure 4.6.

Potential solutions

- ***Moss***

  Description Moss [134] is an anti-fraud system, which employs various code finger printing techniques to detect plagiarism. The Moss system uses a client-server architecture. The client consists of perl script, which communicates with the Moss Internet server over TCP/IP. The client provides the user with an URL pointing to the results of the analysis.

  Design rules For each assignment it should be clear whether it should be scanned for fraud or not.

  Design constraints Moss works in a batch oriented way; all the data to be tested for fraud should be delivered at the same time and increments are not possible.

  Consequences The Athena middleware becomes dependent on the Moss server.

  Pros +Good, confident fraud detection. +Can ignore base frameworks provided to students +Support multiple programming languages +Free to use

  Cons -Integration and archival of the analysis results can be difficult.

- ***JPlag***

  Description JPlag [103] is a plagiarism detection system similar to Moss. JPlag parses the submitted files and searches for similarities in their parse trees. The JPlag architecture uses a client-server architecture. The Java client sends the

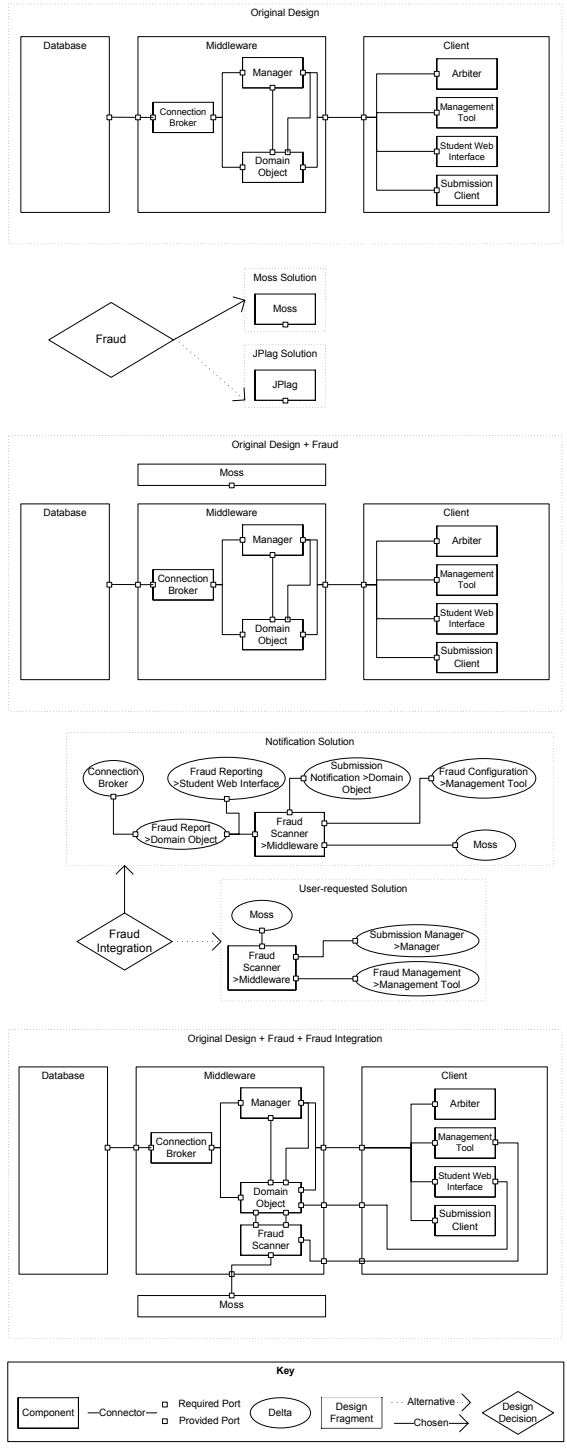

Figure 4.6: Two design decisions of Athena

files for scanning to the server. The results of the analysis can be viewed through a web interface.

Design rules For each assignment it should be clear whether it should be scanned for fraud or not.

Design constraints JPlag works in a batch oriented way, all the data to be tested for fraud should be delivered at the same time and increments are not possible.

Consequences The Athena middleware becomes dependent on the JPlag server.

Pros +Free to use

Cons -Supports a relatively small number of programming languages -No demo available

Decision The Moss solution is chosen, as it supports more programming languages and can ignore base frameworks provided to the students. **Architectural modification** The architectural modification of this design decision is depicted in figure 4.6.

Fraud integration design decision

Problem The Moss Internet server should be integrated with the Athena system. **Motivation** The Athena users should use the anti-fraud functionality in a transparent way. **Cause** The need for a fraud system, as described in *Fraud* design decision. **Context** The design of the Athena system, as depicted in figure 4.6 under the title "Original Design + Fraud".

Potential solutions

- ***Notification***

  Description The *Fraud Scanner* with the help of the *Fraud Configuration* and the *Moss* server keeps the *Fraud Report* for an assignment up-to-date. The *Fraud Reporting* uses the *Fraud Report* to inform the users.

  Design rules The Domain Object responsible for the processing of the student submissions should notify the *Fraud Scanner*, when a new submission for an assignment is made.

  Design constraints The availability of the Moss server may not interfere with the submission process.

  Consequences Every submission by a student leads to a new *Fraud Report*.

  Pros +The data for *Fraud Reporting* is instantly available +Allows for immediate feedback on the detection of fraud

  Cons -Heavy load induced on the Moss server

- ***User-requested***

  Description The user initiates a fraud analysis. The *Fraud Scanner* delivers the analysis using the *Moss* server.

Design rules The *Submission Manager* should provide the student submissions for a fraud analysis for the *Fraud Scanner*.
Design constraints Fraud analysis should be only performed when a user requests for this information in the *Management Tool*.
Consequences The result of the fraud analysis is not stored in the Athena system, but by *Moss*.
Pros +Relatively easy to develop +Light load induced on the Moss server
Cons -Automatic fraud feedback to students is not available.

Decision The decision is made to use the Notification solution, which provides a more active feedback from the system to the users. **Architectural modification** The architectural modification is presented in figure 4.6.

Figure 4.6 presents a view on these two design decisions, which visualizes part of the history of the Athena architecture with the help of design decisions. The top displays the architecture on which the *Fraud* design decision is based. The realization and choice part of the *Fraud* design decision is shown together with the "resulting" architecture below them. The same is done for the *Fraud Integration* design decision.

Note that in the view of design decision (figure 4.6) the mapping of the change elements onto the architecture is visualized in the change elements themselves. For example, in the *Notification* of the *Fraud Integration* design decision the *Submission Notification* delta is mapped onto the *Domain Object*. This mapping is defined with the help of the composition model (see section 4.5.3). However, the visualization of this mapping is not visible in this view. Instead, when the mapping is not clear from the delta name, the $>$ symbol followed by the target of the delta is used to clarify the mapping.

From both design decisions emerge a number of additional requirements. For example, in the *Fraud* design decision an Internet connection to the Moss server is now required for the Athena system to function completely. The same happens with the *Fraud integration* design decision, where requirements are needed about the expected feedback of the fraud system.

Note that the description of the design decisions itself was sufficient to describe the software architecture evolution and its reasons. For example, the Archium model contains all the information needed to explain why the *Fraud Scanner* component is part of the system. Usually with the term "architecture", the latest incarnation of a design is intended. In this case this would be the architecture illustrated on the bottom of figure 4.6. However, as shown, this architecture is the result of a number of design decisions.

4.7 Related work

Archium employs concepts from the field of software architecture [119]. Important concepts in this field are components and connectors, which are believed to lead to better control over the design, development and evolution of large and increasingly dynamic software architectures [11]. Software architecture documentation approaches [29, 67] try to provide guidelines for the documentation of software architectures. However, guidelines for design decisions are absent in these approaches, whereas Archium does provide them.

Architectural Description Languages (ADLs) [107] describe software architectures in a formal language that supports first class architectural concepts. Whereas ADLs try to describe an architecture, Mae [161] and Archium try to describe the evolution leading to an architecture. Mae [161] is an architectural change management tool that tracks changes to an architecture definition by a revision management system. However, it lacks the notion of a design decision and delta. Therefore, it can only track arbitrary changes and not the dependencies between design decisions that the architect is interested in.

Component languages like ArchJava[2] and Koala [170] are programming languages supporting some architectural concepts as first-class entities to various degrees. Archium shares some the concepts of these component languages, but differs as design decisions and architectural change are first-class citizens.

AOP[88] with its implementations like AspectJ[87] and genVoca [13] are approaches using different techniques to achieve multiple separation of concerns. Traditional use of AOP focuses on the code level, on the other hand Archium focusses on the cross-cutting concerns of design decisions at the architectural level.

A design pattern is a special type of design decision. Design patterns [49] are sets of predefined design decisions with known functionality and behavior. The rationale of these decisions, as documented in the description of a design pattern, can be related to the realization [9]. Archium differs from [9] as it keeps design patterns first class in the realization.

Knowledge systems [128], like IBIS [31], model decision processes and try to capture the rationale or knowledge used in these processes. Design decision models [128] are a special type of knowledge system, as they try to capture rationale of design decisions. Archium expands these decision models, as it integrates the decision model with an architectural model.

4.8 Conclusion

Architectural design decisions play an important role in the design, development, integration, evolution, and reuse of software architectures. However, the notion of architectural design decisions is not part of the current perspective on software architectures. We have identified several problems due to this, including high costs of change, design erosion, and limited reuse, which are primarily caused by the vaporization of these design decisions into the architecture.

To address these problems, we propose a new perspective on software architecture, where software architectures are described as set of design decisions. The presented Archium approach is centered around this idea. Archium models the relationship between software architecture and design decisions *in detail* for the first time. It uses a conceptual model consisting of the notions of deltas, design fragments, and design decisions to describe a software architecture. The different concepts were exemplified with the Athena case.

Ongoing and future work on Archium includes the development of tool support (see [5]) facilitating experimentation of the various concepts. In addition, we intend to add support for multiple views on the architecture as different views show different concerns about the architecture [29, 67].

This paper presented a first step in modeling the perspective of software architectures as a set of design decisions. Many research challenges remain in this perspective. For example, how can we distinguish important design decisions? What are interesting relationships between design decisions? What are the crucial factors influencing a design decision?

CHAPTER 5

TOOL SUPPORT FOR ARCHITECTURAL DECISIONS

Published as: Anton G. J. Jansen, Jan van der Ven, Paris Avgeriou, Dieter K. Hammer, Tool support for Architectural Decisions, Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), January 2007.

Abstract

In contrast to software architecture models, architectural decisions are often not explicitly documented, and therefore eventually lost. This contributes to major problems such as high-cost system evolution, stakeholders miscommunication, and limited reusability of core system assets. An approach is outlined that systematically and semi-automatically documents architectural decisions and allows them to be effectively shared by the stakeholders. A first attempt is presented that partially implements the approach by binding architectural decisions, models and the system implementation. The approach is demonstrated with an example demonstrating its usefulness with regards to some industrial use cases.

5.1 Introduction

Current research trends in software architecture focus on the treatment of architectural decisions [93, 96, 158] as first-class entities and their explicit representation in the architectural documentation. From this point of view, a software system's architecture is no longer perceived as interacting components and connectors, but rather as a set of architectural decisions [77]. This paradigm shift has been initiated in order to alleviate a major shortcoming in the field of software architecture: *Architectural Knowledge Vaporization* [21, 163].

Architectural decisions are one of the most significant forms of architectural knowledge [163]. Consequently, architectural knowledge vaporizes because most of the architectural decisions are not documented in the architectural document and cannot be explicitly derived from the architectural models. They merely exist in the form of tacit knowledge in the heads of architects or other stakeholders, and inevitably dissipate. Note that this knowledge vaporization is accelerated if no architectural documentation is created or maintained in the first place.

Architectural knowledge vaporization due to the loss of architectural decisions is most critical, as it leads to a number of problems that the software industry is struggling with:

- **Expensive system evolution**. As the systems need to change in order to deal with new requirements, new architectural decisions need to be taken. However, the documentation of existing architectural decisions that reflect the original intent of the architects is lacking. This in turn causes the adding, removing, or changing of decisions to be highly problematic. Architects may violate, override, or neglect to remove existing decisions, as they are unaware of them. This issue, which is also known as *architectural erosion* [119], results in high evolution costs.
- **Lack of stakeholder communication**. The stakeholders come from different backgrounds and have different concerns that the architecture document must address. If the architectural decisions are not documented and 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.
- **Limited reusability**. Architectural reuse cannot be effectively performed when the architectural decisions are implicitly hidden in the architecture. To reuse architectural artifacts, we need to know the alternatives, and the rationale behind each of them, so as to avoid making the same mistakes. Otherwise the architects need to 're-invent the wheel'.

The complex nature and role of architectural decisions requires a systematic and partially automated approach that can explicitly document and subsequently incorporate them in the architecting process. The development of such an approach is explained in a bottom up fashion in this paper. The explanation starts with the notion of architectural decisions and continues from the point of view of sharing and using these decisions by relevant stakeholders.

We have worked with industrial partners to understand the exact problems they face with respect to loss of architectural decisions. We demonstrate how the system

stakeholders can use architectural decisions with the help of a use-case model. We then introduce a first attempt on implementing the proposed approach, a research prototype entitled Archium that aims primarily at capturing architectural decisions and weaving them into the development process. The Archium tool is demonstrated through an example. The contribution of this paper is therefore a first attempt of putting the theory of architectural knowledge into practice.

The rest of the paper is structured as follows: in Section 2 the notion of architectural decisions is presented. Section 3 gives an overview of the proposed approach. Section 4 presents the implementation of this approach in the Archium tool, followed by the tool demonstration through an example in Section 5. Related work is discussed in Section 6. The paper concludes with conclusions and future work in Section 7.

5.2 Architectural Decisions

To solve the problem of knowledge vaporization and to attack the associated problems of expensive system evolution, lack of stakeholder communication, and limited reuse we need to effectively upgrade the status of architectural decisions to first-class entities. However, first we need to understand their nature and their role in software architecture. Based on our earlier work [21, 77, 163], we have come to the following conclusions on architectural decisions so far:

- They are cross-cutting to a great part or the whole of the design. Each decision usually involves a number of architectural components and connectors and influence a number of quality attributes.
- They are interlaced in the context of a system's architecture and they may have complex dependencies with each other. These dependencies are usually not easily understood which further hinders modelling them and analyzing them (e.g. for consistency).
- They are taken to realize requirements (or stakeholders' concerns), and conversely requirements must result in architectural decisions. This two-way traceability between requirements and decisions is essential for understanding why the architectural decisions were taken.
- They must result in architectural models, and conversely architectural models must be rationalized by architectural decisions. This two-way traceability between decisions and models is essential for understanding how the architectural decisions affect the system.

- They are derived in a rich context: they result from choosing one out of several alternatives, they usually represent a trade-off, they are accompanied by a rationale, and they have positive and negative consequences on the overall quality of the system architecture.

The exact properties and relationships of the architectural decisions [21, 95] are still the topic of ongoing research. Currently, some properties [77, 110, 158] and relationships [96] have been identified. In this paper, the definition from [163] is used for architectural decisions:

> A description of the choice and considered alternatives that (partially) realize one or more requirements. Alternatives consist of a set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements.

A description of an architectural decision can therefore be divided in two parts: a description of the choice and the associated alternatives. The description of the choice consists of elements like: problem, motivation, cause, context, choice (i.e. the decision), and the resulting architectural modification. The description of an alternative include: design rules and constraints, consequences, pros and cons of the alternative [163]. For a more in-depth description how architectural decisions can be described see [158, 163].

5.3 A knowledge grid for architectural decisions

5.3.1 Introduction

In order to support and semi-automate the introduction and management of architectural decisions in the architecting process an appropriate tool is required. In specific, this tool should be a Knowledge Grid [180]: “an intelligent and sustainable interconnection environment that enables people and machines to effectively capture, publish, share and manage knowledge resources”. A knowledge grid for architectural decisions should be a system that supports the effective collaboration of teams, problem solving, and decision making. It should also use ontologies to represent the complex nature of architectural decisions, as well as their dense interdependencies.

To resolve the problem of knowledge vaporization, this system must support architects to add, remove, analyze, and view the architectural decisions made in the

architecting process. It must effectively visualize architectural decisions from a number of different viewpoints, depending on what the stakeholders wish to look at.

Furthermore, it must enable easy sharing of decisions between stakeholders. It must be integrated with the tools used by architects, as it must connect the architectural decisions to documents written in the various tools or design environments. Through this, it enables traceability of these decisions to the artifacts of the architecting process. The system must also help the architect trace the architectural decisions to the requirements on one side and the implementation on the other side.

The Griffin research project [56] is currently working on tools, techniques and methods that will perform the various tasks needed for building this knowledge grid. The project has achieved so far two main contributions: a use case model [162] that describes the required usages of the envisioned knowledge grid, and a domain model [45] that describes the basic concepts and their relationships for storing, sharing, and using architectural decisions. In the next subsection, the use case model is briefly sketched while in Section 5.4 we present an implementation of this knowledge grid: the research prototype Archium.

5.3.2 Use Cases of Industrial Relevance

In the context of the Griffin project, we have done a thorough investigation on the demands for sharing architectural decisions in the architecting process. First, the demands of the project industrial partners were investigated. Interviews were held with several employees from these partners; architects as well as architecture reviewers were interviewed. The industrial partners were from different domains, namely embedded systems, information systems, and radio astronomy. The reports of the interviews were sent back to the interviewee, and suggested corrections were processed.

The results of the first investigation are presented as a set of 27 use cases [162]. Some use cases cannot directly be linked to these interviews and/or current daily practice. They are either use cases that emerged from related work [96], or from the Griffin research team. The use cases have been described according to the UML 2.0 [160] specification.

The use cases present a wide area of issues concerning the problems discussed in the introduction. To narrow this list, we ranked the use cases on the basis of the number of occurrences in the interview reports. This ranking reflects the importance of the use cases for the industrial partners. A list of the nine most important use cases follow, while a more elaborate description is given in Section 5.4:

(UC1) Retrieve architectural decision

(UC2) Add an architectural decision

(UC3) Check for consistency

(UC4) Validate the set of architectural decisions against the requirements

(UC5) Check implementation against architectural decisions

(UC6) Get consequences of an architectural decision

(UC7) Check for completeness

(UC8) Detect patterns of architectural decision dependencies

(UC9) Check for superfluous architectural decisions

According to this ranking, the basic add and retrieve functionality for architectural decisions is the most wanted. Note that modifying an design decision is seen as adding another design decision, which changes the effect of an earlier decision. The more sophisticated evaluation and checking mechanisms are considered somewhat less important, but are desired.

The problems identified in the introduction section are clearly reflected in the use cases. The **expensive system evolution** is tackled by checking the consistency (UC3) and completeness (UC7), but the requirement validation (UC4) and implementation check (UC5) are also very useful in this context. Adding (UC2) and retrieving (UC1) of architectural decisions is essential to alleviate the **lack of stakeholder communication**. Getting the consequences of an architectural decision (UC6) helps in increasing the insight in the effects of a change when the system is evolving. This use case is also the main functionality used to improve the **limited reusability** of architectures. The detection of patterns (UC8) and check for superfluous decisions (UC9) give more insight into the specific architecture, and the potential problems with this architecture.

The following section discusses how the Archium tool, an implementation of the knowledge grid for architectural decisions, fulfils the described use cases.

5.4 The Archium tool

5.4.1 Introduction

The Archium tool [5, 77] is a prototype implementation of the envisioned knowledge grid presented in the previous section, and realizes a part of the Griffin project

use cases [162]. The Archium prototype is a *specialized version* of the envisioned knowledge grid, as it provides a more pragmatic approach to the usage of architectural decisions: it links the architecting process with the system implementation through transformations.

The Archium tool integrates an architectural description language (ADL) [107] with Java. This language allows an architect to describe the elements from a component & connector view [29], and to express architectural decisions, design fragments [77], and rationale. Archium combines the above by modeling the relationship between architectural decisions and the architectural entities (e.g. components and connectors) in detail. In our earlier work [77], we described how these two aspects are integrated with each other. In this paper, the focus is on *how* these concepts are used. Instead of focussing on *what* these concepts are, as we did in our earlier work [77, 163].

The remainder of this section presents how Archium realizes the Griffin use cases. For each use case, the use of Archium is explained, which if followed by a description of the traceability that is used to achieve many of these use cases. The section concludes, with an overview on how Archium tool is realized.

5.4.2 Use case realization

Retrieve architectural decision

Use-case: Given the architectural model (or a part of it), trace back to the architectural decision it is based on. Provide the architectural drivers and the rationale of the decisions.

Archium: The visualization of the Archium tool, allows the architect to select a component or connector. This will cause the relevant architectural decisions to appear on the screen, while the architectural decision responsible for the existence of the element is highlighted with a separate color. Hovering above an architectural decision reveals the relevant rationale of this decision in a tool-tip. An example of this is presented in figure 5.1.

Add an architectural decision

Use-case: Add an architectural decision. Prior to adding a decision, certain prerequisites should be satisfied, i.e. questions that need to be answered in order to be able to take the decision. Also, the evaluated alternatives and rationale about the decision can be recorded.

Archium: In Archium, architectural decisions are considered as first class entities. Adding an architectural decision is therefore the definition of the corresponding element in the Archium language, which allows for the description of the alternatives and rationale of a decision. In this sense, the Archium language acts as a formalized template for architectural decisions.

Check for consistency

Use-case: Check if the current set of architectural decisions is internally consistent. Check if the chosen alternatives have inconsistent consequences on the architectural model.

Archium: The Archium compiler and run-time environment do numerous checks to ensure consistency. For example, the compiler ensures communication integrity [2] by checking whether a component refers to architectural elements that are not defined in the components required interface. Another example is the check Archium makes on whether the functional dependencies of an architectural decision are satisfied before an architectural decision is applied.

Validate the set of architectural decisions against the requirements

Use-case: Trace the requirements to the decisions. Check if the requirements are all sufficiently covered by the decisions that are taken.

Archium: Archium supports the tracing of requirements to architectural decisions and can check whether all requirements are addressed in one or more architectural decisions. The Archium compiler warns about requirements that are not addressed.

Check implementation against architectural decisions

Use-case: At a certain moment in time, the architect would like to see to what extent the implementation effort of the development team is in line with the architectural decisions. Consequently, the architect wants to know where in the development process people ignore or disregard the made decisions.

Archium: The Archium compiler includes a code transformation process, which analyzes the architectural elements (e.g. components, connectors) and transforms them where applicable into Java classes. If the implementation team ignores or disregards the architectural decisions made, either the compiler or run-time environment will warn and prohibit violations of the architectural decisions. Consequently,

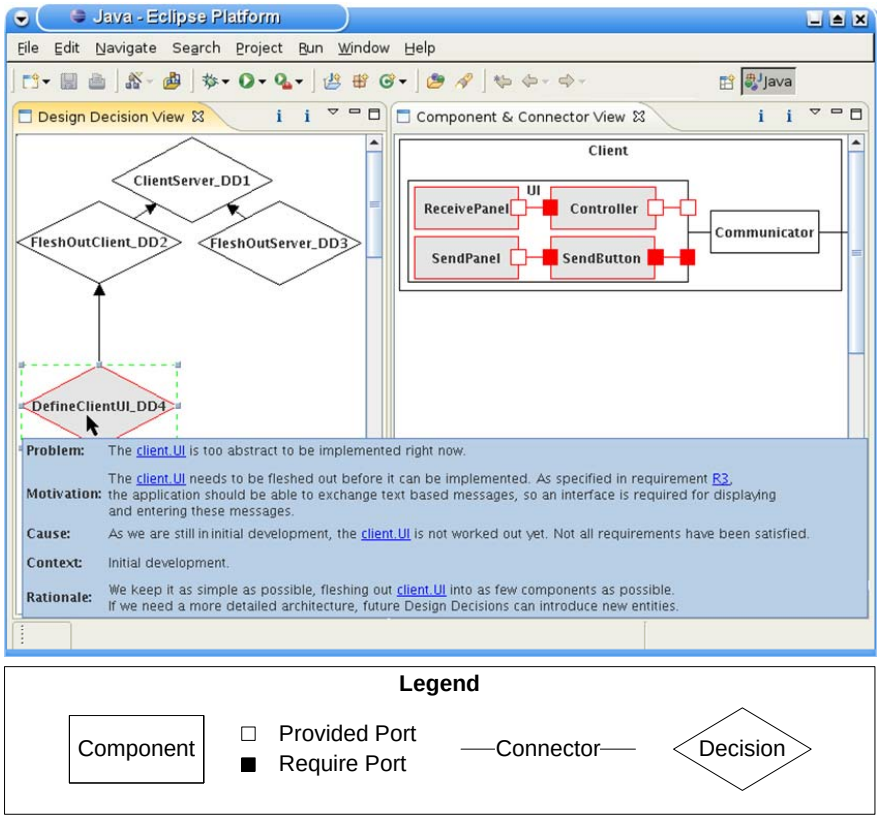

Figure 5.1: Impact of an architectural decision

either the implementation team will have to realign their implementation with the architecture, or define new architectural decisions to adapt it.

Get consequences of an architectural decision

Use-case: The main consequences of a decision are the changes in the model when a decision is executed. Furthmore, new decision topics can be introduced as a consequence of an architectural decision. This use case involves getting insight into the consequences of the decision.

Archium: Archium provides a visualization of an architectural decision by a dependency graph, which gives an indication of the consequences of the decision. An architect can assess the consequences of an architectural decision in the visualizer by hovering over the dependency relationships, and see in a tool-tip what archi-

tectural entities are responsible for a dependency. This helps the architect with evaluating the consequences of an architectural decision. The architect can also select an architectural decision and see its impact on the architecture (see figure 5.1). This is possible, as the Archium tool explicitly traces the change of an architectural decision to the architecture [77].

Check for completeness

Use-case: Check if all the decision topics are covered sufficiently in the architectural decisions taken.

Archium: The Archium compiler tries to check and warn when one or more rationale elements (e.g. motivations, causes, and problems) are missing. Furthermore, it provides errors when no implementation is provided for a chosen alternative, or no alternative is chosen for an architectural decision.

Detect patterns of architectural decision dependencies

Use-case: In order to be able to check the soundness of the architecture, it is needed to analyze the decisions taken, and the dependencies between these decisions. This includes identifying patterns in the graphs of decisions that can lead to guidelines for the architects. For example: decisions being hubs ("Godlike" decisions), circularity of a set of decisions, and decisions that gain weight over time and are thus more difficult to change or remove.

Archium: Archium offers a visualization that provides a view on the functional dependencies between architectural decisions. This is not an automatic pattern detection (a difficult task by definition), but visualizing the dependencies does support the architect in identifying such patterns.

Check for superfluous architectural decisions

Use-case: The architect wants to know if there are superfluous decisions. This can occur in two situations. Decisions can overlap (i.e. are redundant), e.g. parts of the decisions describe the same, or decisions do not affect the current architectural model at all (i.e. are unnecessary).

Archium: Archium supports the architect in identifying *one* class of superfluous decisions: unnecessary architectural decisions that do not have a function within the architecture anymore. These architectural decision do not have a dependency relationship to the main architectural decisions of the application.

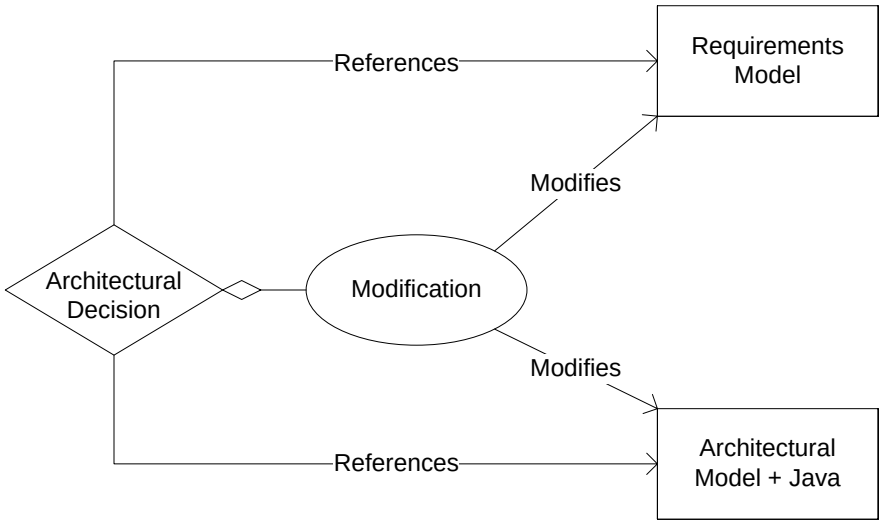

Figure 5.2: Traceability in Archium

5.4.3 Traceability

Archium can support many of the use cases due to its ability to provide traceability among different concepts. The traceability helps one to get a better understanding of the design. Figure 5.2 presents how this works within Archium. Central to the traceability of Archium is the concept of an *Architectural Decision*. It includes a *Modification* part, which alters the *Architectural Model/ Implementation* and the *Requirements Model*. Note that the architectural model includes other architectural decisions. Note as well that the architectural decision also includes alternative modifications that have not been chosen. The different relationships Archium supports between model elements can be classified in two distinct types:

Formal relationships are relationships that are defined in the Archium meta-model. Archium provides explicit language constructs to express these relationships. The Archium tooling (see also section 5.4.4) checks and constrains these relationships. Furthermore, the tooling uses these relations to satisfy some of the use cases. For example, most of the modification relationships (see figure 5.2) are formally defined and used by the Archium tool to determine the impact of an Architectural Decision and relate it to the affected components in the architectural model.
Informal links are relationships that are defined in the textual descriptions of various Archium concepts (e.g. a motivation or problem of a design decision). They work similarly as hyperlinks and allow the expression of a relationship between

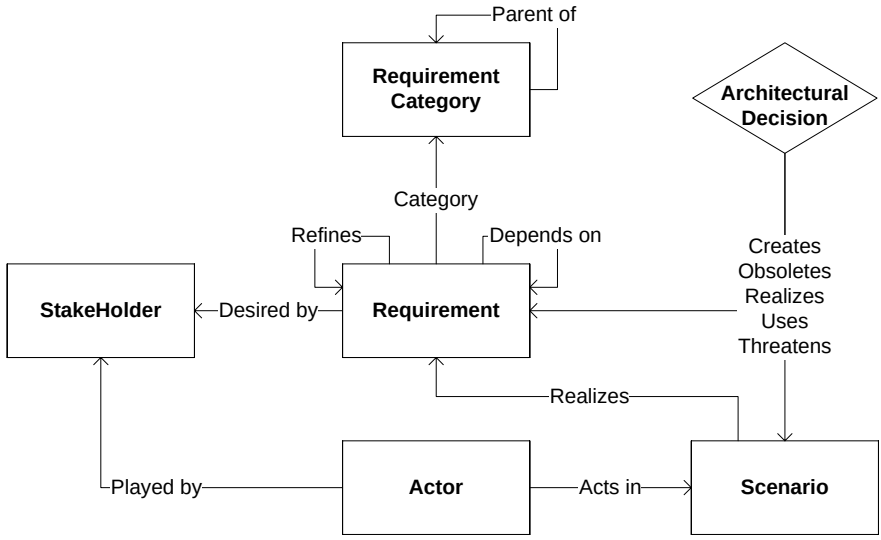

Figure 5.3: Requirements Model of Archium

two model elements. However, these links do not (formally) define the semantics of this relationship. Informal links are defined by putting a reference to a model element between square brackets (e.g. "[ComponentX]") in a textual description. To make it easier to relate elements, the informal links are context aware, i.e. they follow the naming scope of the surrounding model element in which they are used.

The formal relationships between architectural decisions and the requirements model of Archium is presented in figure 5.3. In our previous work [77], we already presented how Archium relates (and therefore provides traceability) architectural decisions with an architectural model. Therefore, the focus in this paper is on the traceability between requirements and architectural decisions.

The requirements model (see figure 5.3) is relatively small, as the primary focus of Archium is on the design part. The model defines five different relationships between an architectural decision and a requirement or scenario. The *creates* relationship is used in the refinement process and traces which requirements came forth of which architectural decision. This is important, as often requirements or scenarios become apparent after an architectural decision is made, as a lot of requirements and/or scenarios only make sense after particular decisions.

The *obsoletes* relation describes the opposite situation, due to new insights certain requirements or scenarios may become obsolete and no longer relevant for

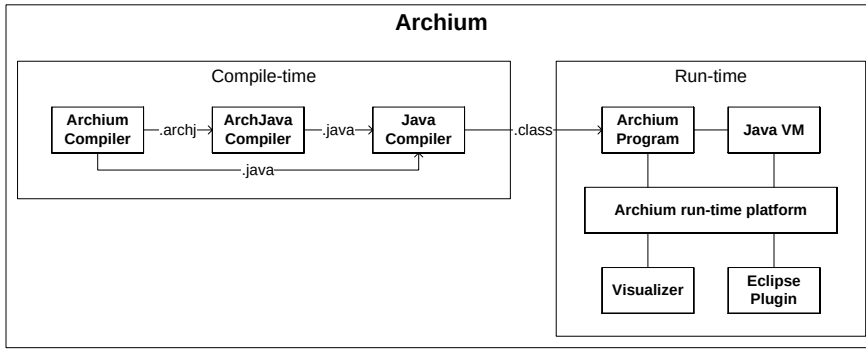

Figure 5.4: The Archium tool architecture

the design. The *uses* relationship denotes that an architectural decision uses a requirement or scenario in its rationale to decide between multiple alternatives. The *realizes* relationship denotes that an architectural decisions tries to achieve (a part of) a requirement or scenario. The *threatens* relationship has the opposite semantic, i.e. an architectural decision makes the achievement of a particular requirement or scenario harder. Note that the existence of an relationship between an architectural decision and a requirement is a confirmation of the fact that a requirement is architectural significant.

5.4.4 Architecture of the Archium tool

Both the requirements model and the informal links are implemented as part of the Archium language. This language includes an ADL (integrated with Java), design decisions, and the requirements model presented in the previous section. The concepts behind the design decisions and ADL part have been explained in earlier work [77]. The language is implemented and used in the Archium tool. Figure 5.4 presents the architecture of this tool. The tool consists of two main parts: a compile-time part that transforms Archium code into Java classes; and a run-time part that performs run-time analysis and support, and executes Archium programs.

The compile-time part works as a pipes-and-filters system [138]. It commences with the *Archium Compiler*, which transforms Archium code (described in the Archium language) into ArchJava [2] and Java code. The *ArchJava* and *Java Compiler* subsequently transform the latter into Java classes. From a user perspective, this compile pipeline is completely transparent. The *Archium Compiler* invokes

the *ArchJava* and *Java compiler* and provides the user with feedback (e.g. compile errors and warnings) in terms of the input Archium code.

The Java classes generated by the compile pipeline constitute the *Archium Program* component, which is executed by the run-time part of the Archium tool. The *Archium Program* uses the *Archium run-time platform*, among other things, for composition of components and reconfiguration of the connections made by connectors. The *Archium run-time platform* also provides services to the *Visualizer* and the *Eclipse Plugin*, in order to allow architects to inspect the architecture and receive notifications of changes made to it.

The frontend of the *Archium Compiler* has been created with the help of the Java Compiler Compiler (JavaCC) [81], which is a parser generator and abstract syntax tree builder. The input for JavaCC is generated by our ArmPrep tool[5]. ArmPrep is a program that semi-automatically merges two JavaCC grammar specifications. This tool is used to merge the Archium language with the Java language.

The semantic analyzer of the *Archium Compiler* analyzes the generated abstract syntax tree for the following constraints:

- Type checking for various constructs in the Archium language, e.g. whether the interface of a connector is compatible with a port of a component.
- Naming convention checking, as the Archium language consists of several different concepts, compliance to the naming conventions is particularly important.
- Relation checking, especially the relations between the architectural decisions rationale and the architectural entities.

The other constraint checks, like communication integrity [2] and Java constraints, are handled by the ArchJava and Java compiler respectively.

The backend of the *Archium Compiler* consists of a template code generator. The code generator uses Velocity Templates [171] to generate ArchJava and Java code for the various Archium ADL concepts like components, connectors, and architectural decisions. The generated code, that is the *Archium Program*, uses the *Archium run-time platform* to create and maintain a run-time representation of the architectural model, requirements, and the architectural decisions. This representation enables the user to explore and make use of the traceability provided within the Archium model. The *Archium run-time platform* provides a service to this representation using Java Remote Method Invocation (RMI). Both the *Eclipse plugin* and *Visualizer* use this service to analyze, trace, and visualize the Archium model. For

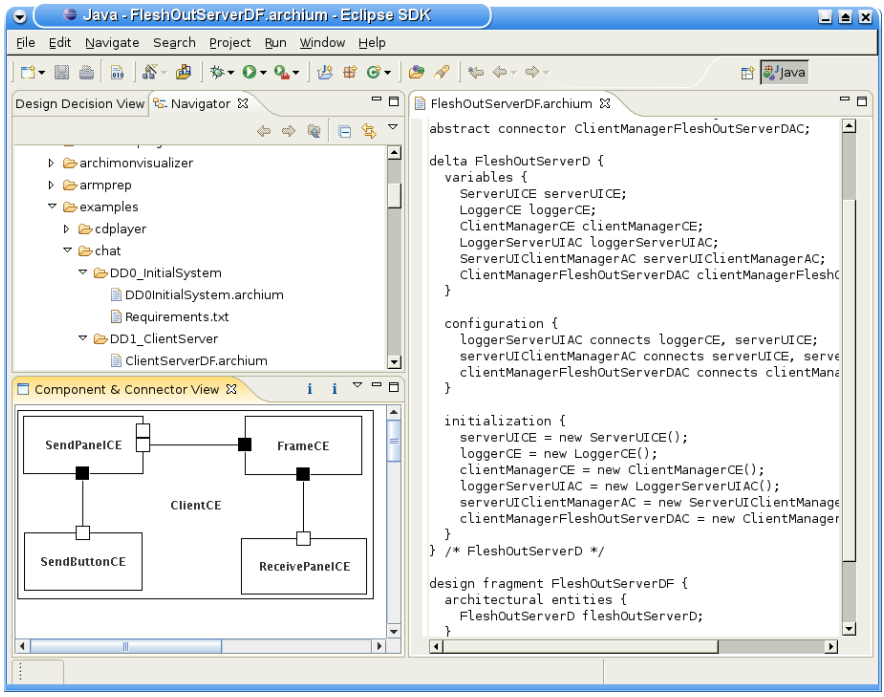

Figure 5.5: The Archium Eclipse plugin

the visualization, both utilize the JGraph toolkit [82] to render and automatically layout the architectural decisions and components & connectors.

5.5 Chat example

5.5.1 Introduction

To illustrate the Archium tool, an example of a chat program is presented in this section. The example consists of nine architectural decisions that define the architecture of the chat program. The architectural decisions are numbered chronologically, and marked with the term AD (e.g. AD1 is the first Architectural Decision).

Writing programs in Archium can be done with the Archium Eclipse plug-in. Figure 5.5 shows a screenshot of this IDE. On the bottom left, a component & connector view of the architecture of the (running) application is visualized. The boxes in this figure represent components; the squares are the ports (black squares are

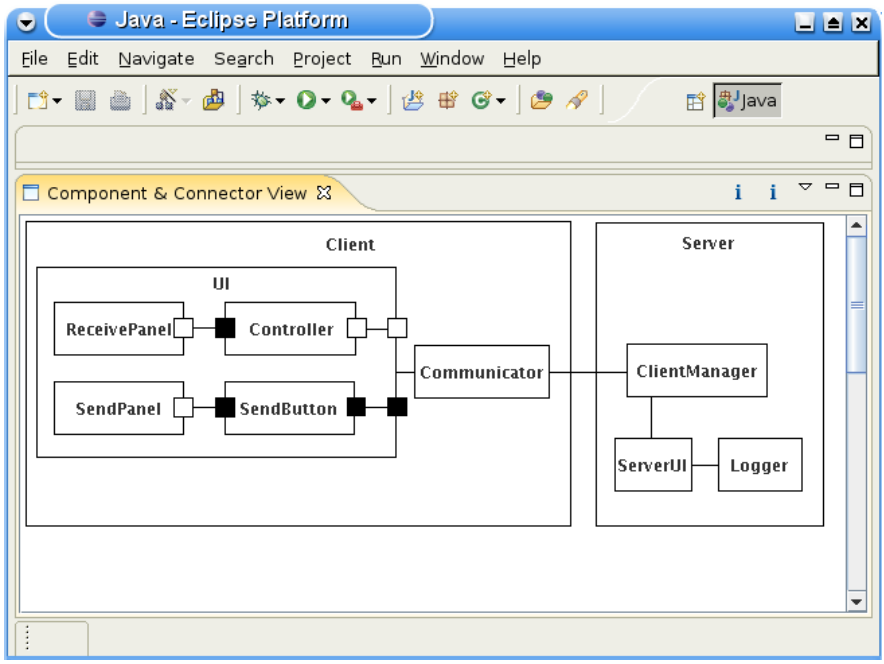

Figure 5.6: Chat example architecture after AD4

required ports, white provided), and the lines represent connectors. On the right, the main editor for the Archium code is shown. In the remainder of this section, two situations are discussed to illustrate what Archium can do.

5.5.2 Usage scenarios

After the first four architectural decisions have been made (AD1-AD4), the architecture has been decomposed in a *Client/Server* style (AD1). The *Client* component consists of a *UI* and *Communicator* (AD2). The *UI* handles the interaction with the user, while the *Communicator* takes care of the communication with the *Server*. Architectural decision AD3 concerns the structure of the server, but this is out of scope for this example. Figure 5.6 shows the state of the component and connector view after AD1-AD4. The boxes in this figure denote components, the lines connectors, and the little squares are provided ports (white square) and required ports (black square).

While implementing the *UI* (AD4), it seemed that the *Communicator* became redundant. The communication could easily be handled by the components used

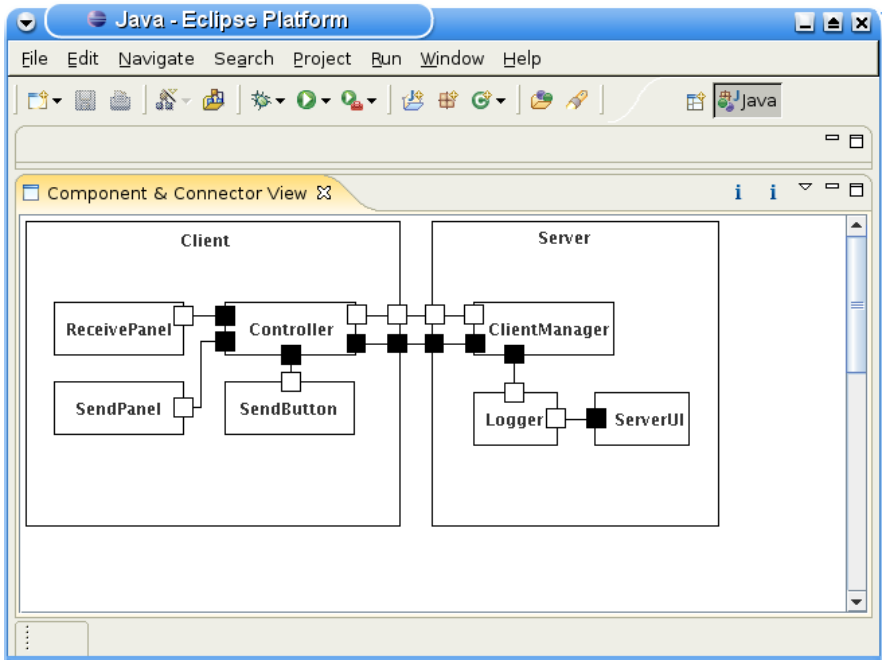

Figure 5.7: Chat example architecture after AD8

in the *UI*. Before deciding to remove the *Communicator*, the decisions dependant on the *Communicator* are checked. This revealed that AD2 is only affected. An architectural decision (AD5) is made to remove the *Communicator* (UC5 and UC2: Check implementation against architectural decisions, add a decision). The Archium compiler is rerun and the chat program is executed to check for any problems (UC3: Check for consistency).

The second situation arises when the requirement for the user interface changes: multiple user interfaces should be supported. The current state of the architecture is presented in figure 5.7. The architect wonders what the consequences of this change in requirements are and traces the original requirement to AD2, where an initial decomposition of the *Client* is made in an *UI* and *Communicator* (UC6: Get consequences of an architectural decision).

However, the *Communicator* is no longer part of the architecture. Tracing the architectural decision dependencies the architect finds the place where the *Communicator* was removed, AD5. As described above, AD5 unfolds the *UI* and removes the *Communicator* component. The rationale of this architectural decision is that the responsibility of the *Communicator* has been relocated to the *Controller* to allow

for easy integration with the *UI* components.

This knowledge leads the architect to think up two alternatives to deal with the changed requirement:

- Reintroducing the *UI*. This should contain all the components in the client with exception of the *Controller*. The *Controller* can be reused with different implementations of the *UI*.
- A new user interface is regarded as a new implementation of the *Client*, thereby creating a specific *Client* for each user interface.

From the rationale of AD5, the architect knows that separating the *UI* and the *Controller* will not be easy and the last alternative is probably the easiest to achieve.

5.6 Related work

Software architecture design methods [11, 19] focus on describing how sound architectural decisions can be made. Architecture assessment methods, like ATAM [11], assess the quality attributes of a software architecture, and the outcome of such an assessment steers the direction of the decision-making process. Our approach focuses on providing ways to capture these architectural decisions, and in the case of Archium, explicitly couples them to the implementation.

Software documentation approaches [29, 67] provide guidelines for the documentation of software architectures. However, these approaches do not explicitly capture the way to take architectural decisions and the rationale behind those decisions.

Architectural Description Languages (ADLs) [107] do not capture the decision making process in software architecting either. There are two notable exceptions. One is formed by the architectural change management tool Mae [161], which tracks changes of elements in an architectural model using a revision management system. However, this approach lacks the notion of architectural decisions and does not capture considered alternatives or rationale about the architectural model. The second exception is the domain specific ADL EAML[133], which models architectures for enterprise applications. In EAML, architectural decisions justify rationale, which provides the architecture description that in turn influences the architectural decisions. However, EAML does not describe *how* architectural decisions influence the architecture description, which is something Archium does.

Architectural styles and patterns [25, 138] describe common (collections of) architectural decisions, with known benefits and drawbacks. Tactics [11] are similar, as

they provide clues and hints about what kind of techniques can help in certain situations. However, they do not provide a complete architectural decision perspective, as presented in this paper.

Currently, there is more attention in the software architecture community for the decisions behind the architectural model. Kruchten [93], stresses the importance of architectural decisions, and presents classifications of architectural decisions and the relationship between them. Tyree and Akerman [158] provide a first approach on documenting design decisions for software architectures. Both approaches model architectural decisions separately and do not integrate them with the architectural model. Closely related to this is the work of Lago and van Vliet [99], who model assumptions on which architectural decisions are often based, but not the architectural decisions themselves.

Integration of rationale with design is also done in the field of design rationale. The SEURAT [24] system maintains rationale in a RationaleExplorer, which is loosely coupled to the source code. This rationale has to be added to the design tool, to let the rationale of the architecture and the implementation be maintained correctly. DRPG [9] couples rationale of well-known design patterns with elements in a Java implementation. Just like SEURAT, DRPG also depends on the fact that the rationale of the design patterns is added to the system in advance. The importance of having support for design rationale was emphasized by the survey conducted by Tang et al. [149]. The results emphasized the current lack of good tool support for managing design rationale.

From the knowledge management perspective, a web based tool for managing architectural knowledge is presented in [110]. They use tasks to describe the usage of architectural knowledge. These tasks are much more abstract then the use cases defined in this paper (e.g. architectural knowledge use, architectural knowledge distribution). They do propose a framework for capturing architectural knowledge, by using techniques for enquiring knowledge from human sources, and by mining used patterns. They provide templates for noting down the knowledge. However, they do not integrate the AK with the design process, instead they distil it, thus it remains separated from the design artifacts.

Finally, another relevant approach is the investigation of the traceability from the architecture to the requirements [173]. Wang uses Concern Traceability maps to reengineer the relationships between the requirements, and to identify the root causes. Methods similar to the proposed ACCA method could be used to generate architectural decision information for Archium.

5.7 Conclusions & Future work

We believe that the field of software architecture will make significant progress when architectural decisions are treated with the same importance as architectural models. This paper presented nine use cases that described the benefits of architecting with architectural decisions treated as first-class entities. The presented use cases covered the basic functionality for a support tool: adding and retrieving decisions, verification of the relationship with requirements as well as the implementation, and visualization of the relationship of architectural decisions with each other.

The Archium tool is a first attempt of realizing the presented use cases and is aimed at the later stages within design. It weaves architectural decisions into architectural models and connects them to the implementation. Archium supports architects in maintaining the architectural decisions taken in the architecting process. We have described the functionality of Archium by explaining how it fulfils the use cases. Specific instances of the use cases in Archium have been explained with an example. Because Archium is able to store the architectural decisions explicitly as artifacts of the architectural model, it decreases the effects of knowledge vaporization.

The Archium tool has not been tested yet in an industrial setting, so empirical verification data is not yet available. This will take place in the scope of the GRIFFIN project, where currently four industrial case studies are being conducted at four different industrial companies. The cases concern different aspects of managing and sharing of architectural decisions. Special attention is given to the integration with the tools currently used by architects (e.g. Microsoft Word, System Architect, Rationale Rose) in the architecting process. These case studies are in the exploratory phase, and help us to validate the use cases [162] and the domain model [45]. In the future, we will evolve Archium according to the outcome of these case studies. Furthermore, we plan to do some experiments to investigate the balance between the effort of capturing architectural knowledge and its benefits. We are also thinking about integrating Archium with other tools, support for more architectural views, and support for team work.

Acknowledgements

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

about architectural knowledge. We would like to thank the people from the GRIFFIN project for the cooperation in creating the use cases.

CHAPTER 6

EVALUATION OF TOOL SUPPORT FOR ARCHITECTURAL EVOLUTION

Published as: Anton Jansen, Jan Bosch, Evaluation of Tool Support for Architectural Evolution, Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), pp. 375-378, September 2004

Abstract

Evolution of software architectures is, unlike architectural design, an area that only few tools have covered. We claim this is due to the lack of support for an important concept of architectural evolution: the notion of architectural design decisions.

The absence of this concept in architectural evolution leads to several problems. In order to address these problems, we present a set of requirements that tools should support for architectural evolution. We evaluate existing software architecture tools against these architectural requirements. The results are analyzed and an outline for future research directions for architectural evolution tool support is presented.

6.1 Introduction

Software architecture [119, 138] has become a generally accepted concept in research as well as industry. The importance of stressing the components and their connectors of a software system is generally recognized and has lead to better control over the design, development and evolution of large and increasingly dynamic software systems.

Although the achievements of the software architecture research community are admirable, architectural evolution still proves to be problematic. Most research and tool development in the area of software architecture has focused on the careful design, description and assessment of software architecture. Although some attention has been paid to evolution of software architecture, the key challenge of the software architecture community has been that software architectures need to be designed carefully because changing the software architecture of a system after its initial design is typically very costly. Many publications refer to changes with architectural impact as the main consequence to avoid.

Interestingly, software architecture research, almost exclusively, focuses on this aspect of the problem. Very little research addresses the flip side of the problem, i.e. how can we design, represent and manage software architectures in such a way that the effort required for changes to the software architecture of existing systems can be substantially reduced. As we know that software architectures will change, independently of how carefully we design them, this aspect of the problem is of particular interest.

To reduce the effort in changing the architecture of existing software systems it is necessary to understand why this is so difficult. Our studies into design erosion and analysis of this problem, i.e. [168] and [78], have led us to believe that the key problem is knowledge vaporization. Virtually all knowledge and information concerning the results of domain analysis, architectural styles used in the system, selected design patterns and all other design decisions taken during the architectural design of the system are embedded and implicitly present in the resulting software architecture, but lack a first-class representation.

The design decisions are cross-cutting and intertwining at the level at which we currently describe software architectures, i.e. components and connectors. The consequence is twofold. First, the knowledge of the design decisions that lead to the architecture is quickly lost. Second, changes to the software architecture during system evolution easily cause violation of earlier design decisions, causing increased design erosion [72, 119, 168].

We believe that architectural design decisions are a key concept in software architectures, especially during evolution. Consequently, capturing design decisions is of great importance as it addresses some fundamental problems associated with architectural evolution. Effective support for architectural design decisions requires tools support, but this currently largely lacking. However, existing tools do provide parts of the necessary solution. The aim of this paper is therefore to evaluate existing software tools with respect to their ability to represent aspects of architectural design decisions.

The contribution of this paper is threefold. First, it develops the notion of explicit architectural design decisions as a fundamental concept during architectural evolution. Second, it states a set of requirements that tooling needs to satisfy in order to adequately support evolution of architectural design decisions. Finally, an analysis is presented of existing tool approaches with respect to the stated requirements.

The remainder of this paper is organized as follows. The concept of architectural design decisions and the problems associated with the architectural evolution are explained in more depth in section 6.2. In section 6.3, the requirements for tool support of architectural design decisions are formulated. The section after that presents the tools and the evaluation on the requirements stated in the proceeding section. A discussion of the evaluation results is presented in section 6.5. The paper concludes with future work and conclusions in section 6.6.

6.2 Architectural Design Decisions

Architectural evolution is closely related to architectural design decisions. In our experience, architectural evolution is fundamentally the addition, change and removal of architectural design decisions. Classifications such as [14, 27, 109] can all be viewed as the consequences of architectural design decisions.

Due to new and changed requirements, new design decisions need to be taken or existing design decisions need to be reconsidered. Consequently, the architectural design decision is the central concept in the evolution of software architectures. We define an architectural design decision as:

> A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture.

This definition of architectural design decisions uses the following elements:

- **Architectural change** Describes the addition, subtraction, and modification of the architectural entities used in the architecture.
- **Rationale** Design decisions are often made with a reason, these reasons behind an architectural design decision are part of the rationale of an architectural design decision.
- **Design rules** Architectural design decision can prescribe rules for architectural change.

- **Design constraints** Besides design rules, certain design possibilities can be invalidated by an architectural design decision. Design constraints therefore describe the opposite side of design rules, i.e. they prohibit certain architectural changes.
- **Additional requirements** Design decisions can give rise to additional requirements. Since designing is often an exploratory process, new requirements will be discovered once a design decision has been made and the architectural change is developed further.

An architectural design decision is therefore the result of a design process during the initial design or the evolution of a software system. This process can result in changes to architectural entities, e.g components & connectors, that make up the software architecture or design rules and constraints being imposed on the architecture (and on the resulting system). Furthermore, additional required functionality (a form of constraining) can be demanded from the architectural entities as a result of a design decision. For instance, when a design decision is taken to use an object-oriented database, all components and objects that require persistence need to support the interface demanded by the database management system.

Architectural design decisions may be concerned with the application domain of the system, the architectural styles and patterns used in the system, COTS components and other infrastructure selections as well as other aspects needed to satisfy all requirements.

However, current approaches to the description of software architectures leave the notion of architectural design decisions implicit and provide no mechanism for representing this concept. During the system's evolution, the software architect is forced to reconstruct the design decisions underlying the current architecture that are relevant for the required changes. This leads to a number of problems that are insufficiently addressed:

- **Lack of first class representation** Architecture design decisions lack a first class representation in the software architecture. Once a number of design decisions are taken, the effect of individual decisions becomes implicitly present, but almost impossible to identify in the resulting architecture. Consequently, the knowledge about the what and how of the software architecture is soon lost [22, 89] Some architecture design methods [29, 67] stress the importance of documenting architecture design decisions, but experience shows that this documentation often is difficult to interpret and use by individuals not involved in the initial design of the system.

- **Design decisions cross-cutting and intertwined** Architecture design decisions typically cross-cut the architecture, i.e. affect multiple components and connectors, and often become intimately intertwined with other design decisions.
- **High cost of change** A consequent problem is that a software architecture, once implemented in the software system, is sometimes so expensive to change that changing the architecture is not economical viable. Due to the lack of first-class representation and the intertwining with other design decisions, changing or removing existing design decisions is very difficult and affects many places in the system.
- **Design rules and constraints violated** During the evolution of software systems, designers, and even architects, may easily violate the design rules and constraints imposed on the architecture by earlier design decisions.
- **Obsolete design decisions not removed** Removing obsolete architecture design decisions from an implemented architecture is typically avoided, or performed only partially, because of (1) effort required, (2) perceived lack of benefit and (3) concerns about the consequences, due to the lack of the necessary knowledge about the design decisions. The consequence, however, is the rapid erosion of the software system, resulting in high maintenance cost and, ultimately, the early retirement of the system [72, 119, 168].

To solve the aforementioned problems, an important role is laid down for the notion of an architectural design decision. Only with proper tool support can this notion be realized. Consequently, requirements are needed for such tools in order to support this fundamental concept of architectural evolution. In the following section, the requirements for architectural design decisions are presented. The requirements in turn are used in section 6.4 to evaluate which parts of architectural design decisions are already supported in existing tools.

6.3 Requirements

In this section, the requirements are presented that need to be satisfied to support architectural evolution in the form of architectural design decisions. The individual requirements have been grouped together into three groups: architecture, architectural design decisions, and architectural change.

Architectural design decisions deal with the evolution of software architecture. Consequently, tools should satisfy requirements regarding software architectures. The concept of architectural design decisions introduces some requirements as well, which are covered in the group architectural design decisions.

Architectural change is an important part of an architectural design decision. Requirements for the changing influence architectural design decisions have on the architecture are covered by the architectural change group of requirements.

The requirements for tools to provide architectural design decision support are the following:

6.3.1 Architecture

1. **First class architectural concepts** For software architecture to be of use, it is required to have a shared domain model between the stakeholders. The domain model should provide a common ground for the stakeholders involved about the concepts used in the software architecture. However, a shared domain model alone is not enough; the architectural concepts also need to be first class citizens. As software architecture deals with abstractions, it is very important to define these abstractions in a first class way. The way in which abstraction choices are made is very subjective and this greatly influences the resulting architecture.

   Expressing these abstractions in a consistent and uniform way is therefore essential for software architectures. Only when the architectural concepts become first class can abstraction choices be explicitly communicated. Consequently, misinterpretations about the used abstractions are reduced. This effect not only eases negotiations about the entities of an architecture, but also communication and reasoning about the evolution of the architecture becomes easier.

2. **Clear, bilateral relationship between architecture and realization** In the view of evolution, it is important to have a bilateral relationship between the software architecture and the realization [106]. During evolution changes in the architecture will have an effect on the realization of the system and vice versa. Knowing the relationship between the architectural entities and their realization is essential for reasoning about the effects of change (caused by architectural design decisions) have on the architecture or realization.

3. **Support multiple views** Software architectures have different views for different concerns [29, 67, 92]. Architectural design decisions often influence multiple concerns simultaneously, because they try to strike a balance in the effects they have on different concerns in their changes. Consequently, for architectural evolution it is important to know these different concerns. Hence, multiple views on the architecture should be supported.

6.3.2 Architectural design decisions

4. **First class architectural design decisions** Architectural design decisions in our vision are the architectural changes evolving the architecture, as already stated in the introduction (see section 6.1). A first class representation of design decisions is required to make evolution explicit. First class design decisions can be communicated, related and reasoned about. For example, an important relationship is the dependency between design decisions. If a design decision is undone, it is important to know which other design decisions might be invalidated. The dependency relationship can provide this information. However, this can only be achieved if there is a first class notion of an architectural design decision in the first place.

5. **Under-specification and incompleteness** An important property of design decisions is the ability to specify incompleteness. The proposed solutions in a design decision can only give a general and incomplete description of how the problem at hand could be solved. Consequently, design decisions are often made on the basis of incomplete information. Knowing which part of the chosen solution is not known is important meta-knowledge. Since designing is (partly) an exploratory process, these incomplete parts are often the sources for additional design decisions and invalidations of implicit assumptions earlier made. Hence, they are important for an architect to evolve an architecture.

6.3.3 Architectural change

6. **Explicit architectural changes** A well defined relationship between the proposed solutions of an architectural decision and the architectural entities involved is essential. The proposed solutions describe design alternatives for solving the problem of a design decision.The decision part of a design decision decides which of the proposed solutions will be realized. The realization of these solutions will change the architecture. The resulting architectural changes define great parts of the semantics of the solution. As the exact semantics cannot be known in advance without much effort. Consequently, the architectural change required to realize the solution is an inherent part of the solution. Hence, an explicit representation of these architectural changes is required to express this important part of design decisions. Architectural changes therefore form the bridge between the first class architectural entities and the rationale part of architectural design decisions.

7. **Support for modification, subtraction, and addition type changes** The characteristic types of change of evolution often distinguished are the corrective, perfective, and adaptive types [51]. However, this classification focuses on the reasons behind the change, not on the effect the changes have on the system. Software architectures primarily deal with the global structure of the system. Consequently, architectural evolution has its main focus on the structural type of changes:modification, subtraction, and addition [109, 115]. Replacement is not a basic change type, as it can be accomplished using subtraction followed by addition in one atomic action. Tools need to support these three basic change types on architectural entities if they want to support architectural evolution. Otherwise, not all types of changes to the architecture can be managed by the tools.

6.4 Evaluation

In this section, six tools are evaluated against the requirements stated in section 6.3. For each tool, an overview description is provided, followed by a short description of the support the tool provides for each requirement. The evaluation presented will be used in section 6.5 to discuss the general tool performance on the three groups of requirements.

6.4.1 ArchStudio 3

6.4.1.1 Description

ArchStudio 3 [106, 115] is an architecture-driven software development environment, i.e. software development from the perspective of software architecture. ArchStudio supports a particular architectural style: the C2 architectural style. C2 [154] supports the typical architectural concepts as components, connectors and messages. The C2 architectural style is a hierarchical network of concurrent components linked together by connectors (or message routing devices) in a layered way. C2 requires that all communication between C2 components is achieved through message passing.

ArchStudio uses the C2 architecture expressed in xADL[37] as a way to communicate with external tools. The sister application Ménage [161], used for software product family architectures (SPF) [19], has an architectural change management support tool called Mae [161]. The Mae tool is a change management tool for the

C2 specifications supporting the evolution of the architecture definition by revision management.

6.4.1.2 Evaluation

- **Architecture**
  - R1 The underlying C2 part of ArchStudio supports first class architectural concepts as architecture, configuration, components, connectors and messages.
  - R2 ArchStudio only supports a one-way, relationship from the software architecture to the realization (Java).
  - R3 The architectural concepts supported by ArchStudio are all part of the Component & Connector view [29] on the software architecture. ArchStudio does not support other architectural views.
- **Architectural design decisions**
  - R4 The concept of an architectural design decision is not supported by ArchStudio.
  - R5 Since the concept of an architectural design decision is not supported, there is no support for under-specification and incompleteness. However, ArchStudio does support inconsistent architectural models, which is part of the required incompleteness.
- **Change**
  - R6 Explicit architectural changes are only partially supported by ArchStudio. The tool supports the basic operations and does not have a first class representation of a change in itself. However, Mae, the change management tool, which is no part of but related to ArchStudio, has support for the notion of revisions. Consequently, explicit architectural changes are only supported by an external tool (Mae) and not by ArchStudio itself.
  - R7 ArchStudio supports the basic operations of change, apart from the modification type. However, replacement as in an atomic subtraction and addition operation is not supported. Mae claims to support replacement of components by encapsulating the separate versions in one component and allowing run-time change between them.

6.4.2 ArchJava

6.4.2.1 Description

ArchJava [2, 3] is an extension to Java that aims to unify software architectural concepts with the implementation. It uses a type system to ensure that the implementation conforms to an architecture and focusses especially on communication integrity. Communication integrity defines that the components in a program can only communicate along declared communication channels in the architecture. ArchJava enforces this communication integrity for control flow: a component may only invoke an operation of another component if it is connected to the other component in the architecture.

6.4.2.2 Evaluation

- **Architecture**
  - R1 ArchJava supports the architectural concepts of connectors, components, configuration, and ports.
  - R2 The supported architectural concepts are implemented directly as first class entities defined as an extension to Java. Consequently, there is no division between the architecture and its realization.
  - R3 Only the architectural concepts of the component & connector view [29] are supported by ArchJava.
- **Architectural design decisions**
  - R4 The concept of an architectural design decision is not supported by ArchJava.
  - R5 Since the concept of an architectural design decision is not supported, the language of ArchJava does not support under-specification and incompleteness.
- **Change**
  - R6 Architectural change is not explicitly supported in ArchJava.
  - R7 ArchJava does not support the basic change types for evolution. An exception is formed by the ability to add a connector. A special connect statement allows new connections to be created at run-time between the components, but only if their type is a subtype of an earlier defined type.

6.4.3 AcmeStudio

6.4.3.1 Description

AcmeStudio is the tool used as a front end for Acme [50], which is an architectural description language. The development of Acme started back in 1995 as an ADL interchange language but has evolved to an ADL itself. Acme currently makes use of xADL [37]. AcmeStudio is a graphical editor for Acme architectural designs, which allows editing of designs in existing styles, or creating new styles and types using visualization conventions that can be style dependent. The integrated Armani constraint checker [148] is used to check the architectural design rules. The tool is implemented as an Eclipse plug-in for portability and extensibility.

6.4.3.2 Evaluation

- **Architecture**
  - R1 Acme, as used by AcmeStudio, supports the architectural concepts of components, connectors, configuration, and ports.
  - R2 AcmeStudio has a code generating ability to Java and C++, resulting in a one-directional relationship from the architecture to the realization.
  - R3 AcmeStudio only supports the Component & Connector view [29]. Although the tool does support multiple views on the architecture (called "representations"), only one type is implemented.
- **Architectural design decisions**
  - R4 The concept of architectural design decisions is not supported by AcmeStudio.
  - R5 Since the concept of an architectural design decision is not supported, underspecification and incompleteness are not explicitly supported.
- **Change**
  - R6 Architectural change is not supported.
  - R7 Supporting mechanisms for the change types are not available.

6.4.4 SOFA

6.4.4.1 Description

SOFA (SOFtware Appliances) [121] is a component model in which applications are viewed as a hierarchy of nested components. In SOFA, the architecture of a

component describes the structure of the component by instantiating direct subcomponents and specifying the subcomponents' interconnections via connectors. The use of connectors in SOFA is to provide separation between the application logic and the necessary interaction semantics that cover deployment dependent details [41]. To describe its components and architectures, SOFA uses the Component Definition Language (CDL). The CDL is loosely based on the Interface Description Language (IDL) as used in CORBA [114].

6.4.4.2 Evaluation

- **Architecture**
  - R1 In the accompanied CDL, SOFA has a first class representations for architecture, modules, and components (called frames [121]).
  - R2 SOFA creates the architectural infrastructure. The implementation is still defined in separate Java classes. Consequently, SOFA uses a one-way code generation approach.
  - R3 Although SOFA defines modules and component concerns, these concerns cannot be used orthogonal in SOFA. Furthermore, SOFA only has a textual interface to the Component Definition Language presenting one view to the component model. Hence, multiple views are not supported in SOFA.
- **Architectural design decisions**
  - R4 The concept of (architectural) design decisions is not supported in SOFA.
  - R5 SOFA does not satisfy the requirement for incompleteness and under-specification.
- **Change**
  - R6 Architectural change is not supported as a first class entity. However, explicit versioning is part of the component definition language model defining the component model.
  - R7 SOFA has support for the replacement of components at run-time. However, the basic change types are not supported.

6.4.5 Compendium

6.4.5.1 Description

Compendium [8, 135, 181] is (partly) a knowledge system based on gIBIS [32], which in turn uses the decision model IBIS[97]. A knowledge system, which uses a decision model, tries to capture the rationale behind the decisions made in a decision process. Architectural design decisions can be seen as a very specific kind of

decisions made in a specific process (architectural design phase), which makes this tool a candidate for evaluation.

Compendium centers on capturing design rationale created in face-to-face meetings of design groups, potentially the most pervasive knowledge-based activity in working life, but also one of the hardest to support well. The tool provides a methodological framework, for collective sense-making and group memory. Compendium excels in enabling groups to collectively elicit, organize and validate information and make decisions about them. In order to integrate this with pre/post-meeting design activities and artifacts, the created reasoning diagrams can be transformed into other document formats for further computation and analysis. The domain independence of Compendium's reasoning mapping technique is the tool strength and weakness.

6.4.5.2 Evaluation

- **Architecture**
  - R1 Compendium has no notion of first class architectural concepts.
  - R2 A relationship between architecture and realization can only be defined by relating the artifacts in Compendium. Consequently, there is only an indirect relationship between the artifacts of the architecture and realization.
  - R3 As Compendium has no notion of architectural concerns, multiple views are not supported.
- **Architectural design decisions**
  - R4 Compendium has a first class notion of a design decision (in the form of an issue). Furthermore, it supports rationale capturing about design decisions.
  - R5 The argument and position elements of the IBIS model [97] allows for under-specification of the design decisions and the increasing refinement of them.
- **Change**
  - R6 Compendium does not support architectural change, let alone a first class representation of it. It does support versioning of an artifact describing the architecture. Through this, Compendium could track architectural change.
  - R7 As the notion of a software architecture is not known in Compendium, the basic changes type are not supported.

6.4.6 Archium

Please note that this evaluation of Archium was not part of the original publication of this chapter, but was part of the publication chapter 5. However, this evaluation was left out in the final publication due to space constraints. To be complete in our evaluation, we present this short evaluation of Archium.

6.4.6.1 Description

For an in-depth description of Archium, we refer to chapter 4 on page 79 and chapter 5 on page 101 of this thesis.

6.4.6.2 Evaluation

- **Architecture**
  - R1 The Archium tool supports first class architectural concepts of the component & connector view, like components, connectors, ports etc.
  - R2 The relationship between the architecture and realization is very closely related in Archium, as they have been integrated with each other. Consequently, a bilateral relationship exists between the two
  - R3 Archium does support multiple views in the form of the component & connector view [29], and an architectural decision dependency view. Figures 5.5 on page 115 and 5.1 on page 109 give concrete examples of these views. However, important views like module and deployment views [29] are to be added later on.
- **Architectural design decisions**
  - R4 The Archium tool includes a first class representation of architectural decisions.
  - R5 Under-specification and incompleteness are only partially supported in the tool in the form of stubs and optional rationale elements. However, the Archium tool lacks the capabilities to explicitly express the incompleteness of its architectural decisions and architectural elements. For example, if an architectural decision requires more investigation and is put aside for the moment, Archium cannot express this state of an architectural decision.
- **Change**
  - R6 In various ways, explicit architectural change can be expressed in the Archium tool. Changes to components, connectors and configurations all can be expressed explicitly.

Table 6.1: Evaluation result of the examined tools

Approach	Architecture			A.D.D.		Change	
	R1	R2	R3	R4	R5	R6	R7
ArchJava	++	++	-	- -	- -	- -	-
ArchStudio	+	+/-	-	- -	- -	+/-	+/-
AcmeStudio	+	+/-	+/-	- -	- -	- -	- -
SOFA	+	+/-	- -	- -	- -	+/-	+
Compendium	- -	- -	- -	+	++	- -	- -
Archium	++	++	+/-	++	+/-	++	+

R7 Furthermore, the Archium tool supports all the three basic change types (i.e. addition, subtraction, and modification of architectural entities) for these explicit architectural changes. However, this support is limited to the architectural entities. An external version management system should be used to track changes to the architectural decisions themselves.

6.5 Discussion

In the previous section, six tools were evaluated with respect to their support for architectural evolution. Table 6.1 provides an overview of the results of the evaluation, by presenting the tools against the requirements. For each requirements group (architecture, architectural design decisions, architectural change), the results are discussed in more detail.

6.5.1 Software Architecture

A clear bilateral relationship between architecture and realization proves to be troublesome for most tools. Most of them (ArchStudio, AcmeStudio, SOFA) use a generative approach. The architecture is defined first in the tool, which than generates the implementation classes once. For evolution this means that two models have to be maintained, one for the realization, and one for the architecture. ArchJava and Archium are the approaches in the evaluation that do not suffer from this co-evolution problem. Since these approaches mix the architecture first-class with the rest of the realization.

Software engineers are notorious in neglecting maintenance of separate models, because they find it very hard to do and do not see clear benefits in maintaining them.

In many organizations, it is not uncommon to use an explicit architecture specification only during the initial design phase. The explicit architecture is primarily used to explore ideas for potential solutions. Once the (automated) translation to a detailed design and implementation is made, the architecture description is no longer updated. Consequently, the description of the architecture slowly starts diverting from the architecture of the realization, which degrades the usefulness of the description of the architecture. In the view of architectural evolution, this is definitely a subject for further research.

The idea of separate view points to an architecture [29, 67] is neither supported, nor implemented in any of the evaluated tools besides Archium. All the evaluated tools apart from Compendium and Archium concentrate on a single view: the Component & Connector (C&C) view [29]. This is probably due to the fact that the C&C view is conceptually the closest to the result the architect want to achieve. The other views, in the view of the tools, more or less "result" from this view as specific realization choices are made by the tool when generating the skeleton for the implementation. Compendium does not support architectural views at all. Archium covers both an architectural design decision view and a component & connector view, but still lacks important other views, especially those of the module and deployment viewtype [29].

6.5.2 Architectural Design Decisions

Apart from Compendium and Archium, the concept of (architectural) design decisions is not supported. Compendium does support design decisions, as issues elements from the IBIS model. The tool supports arguments and positions of stakeholders regarding the current problem that can be solved (partially) by a design decision. A major drawback of Compendium (and other knowledge systems) is that it does not explicitly support architectural concepts. The tool only allows for references to be made to artifacts describing the context of the problem (i.e. the architecture) and artifacts describing potential solutions. Apart from Archium, there are currently (as of 2004) no artifacts for software architectures that can express design solutions to a design problem in an unambiguous way.

The idea of (architectural) design decisions resulting from a design decision process, as used in the research community of knowledge systems, is not treated explicitly in the software architecture community. Consequently, the design rationale of an evolving software architecture is not captured in these tools. Finally, architectural design decisions are implicit, which results in the problems already stated in the introduction (see section 6.1).

6.5.3 Architectural change

Architectural change is an important part of architectural design decisions, as this change primarily affects the architecture and evolves the system. The architectural change forms the bridge between the other aspects of an architectural design decision and the architecture.

Explicit architectural change is at the heart of Archium and is therefore well supported. Of the other evaluated tools, only SOFA and ArchStudio provide some support for this change and the required change types. SOFA has explicit version management in its component language and does support the basic change types to some extent. However, the *change* of the first class entities is not explicit and grouped. Hence, SOFA cannot relate the change to the evolution of the system.

ArchStudio, with the help of Mae, has some support for architectural change. Revisions of components and connectors can be stored and retrieved with the help of the Mae tool. However, the change itself is not explicit in Mae, i.e. the change of the revisions is not grouped in an explicit entity. Furthermore, Mae does not support all the change types. Especially, the modification change seems to be lacking.

Change management tools [174], which are not evaluated, could also be used to track architectural change. For example, they could track the changes Archium cannot capture. However, these approaches require a system model that describes the entities that can change and their relationships. However, these concepts for architectural change are not explicitly defined for these tools. Consequently, tracking architectural change with these kind of tools is troublesome.

In general, architectural tool support (apart from Archium) does not view evolution as an inherent part and separate dimension of a software architecture. The focus of the majority of tools is on defining the architecture in the right way. The change of the architecture is often overlooked and not implemented and left over to change management tools. Consequently, tool support for architectural evolution is currently only partially realized.

Archium has addressed some of the issues other tools have struggled with. Compared to other approaches, Archium performs quite well according to the defined criteria, but needs improvements to alleviate its shortcomings. Most significantly, the tool in its current form is not very suitable for the early design phases. This is because the architect generally uses natural language in these early phases to express the architecture, as opposed to the formalized architectural description in the Archium tool.

6.6 Conclusion

The notion of software architecture has become an increasingly important concept in software engineering research and practice. The rationale for this is that an explicit software architecture facilitates the control over the design and development evolution of large and complex software systems.

Evolution of software systems and their associated software architecture are not equally well supported by existing state-of-the-art approaches. Although basic analysis of architecture evolution can be performed, the evolution of the key architecture concepts during evolution are not captured. The claim of this paper is that this is due to the lack of support for the notion of architectural design decisions. In our experience, architecture evolution is fundamentally the addition, change and removal of architectural design decisions.

The lack of support for architectural design decisions in software architecture representations and associated tool support leads to several problems, including high cost of change of already implemented design decisions, the easy violation of design rules and constraints and the cross-cutting and intertwining nature of design decisions.

In this paper, we have presented a set of requirements that tools for architecture evolution should support. The requirements have been arranged into three groups: architecture, architectural design decisions, and architectural change requirements. For the software architecture, the requirements are: support for architectural concepts, a clear bilateral relationship to the realization, and support for multiple architectural views on a system. The architectural design decisions requirements include a first class representation of this concept and support for dealing with the incompleteness and under-specification of these decisions. The architectural change requirement consists of support for explicit architectural change and the ability of the tool to support the fundamental different types of change (i.e. modification, subtraction, and addition) on the architectural entities.

Overviewing the results of the evaluation, it is concluded that there exists a gap between tools for capturing rationale, architectural change, and software architectures. This gap can only be closed if architectural evolution becomes an inherent dimension of the description of a software architecture. We believe that architectural design decisions are the missing concept in this perspective and that tools need to support this concept in order to support architectural evolution.

In our future work, we will try to make the concept of architectural design decisions more explicit. The integration of architectural design decisions, the resulting

architectural change, and the relationship to the software architecture will be important areas of work. Hopefully, with the right decisions we will evolve and mature the concept of architectural design decisions to tool support, making architectural evolution an inherent part of software architectures.

"The Wheel of Time turns, and Ages come and pass, leaving memories that become legend. Legends fades to myth, and even myth is long forgotten when the Age that gave it birth comes again. In one Age, called the Third Age by some, an Age yet to come, an Age long past, a wind rose in the Mountains of Mist. The wind was not the beginning. There are neither beginnings nor endings to the turning of the Wheel of Time. But it was ***a*** *beginning"*

– The Wheel of Time series, Robert Jordan

CHAPTER 7

DOCUMENTING AFTER THE FACT: RECOVERING ARCHITECTURAL DESIGN DECISIONS

Published as: Anton Jansen, Jan Bosch, Paris Avgeriou, Documenting after the fact: recovering architectural design decisions. Journal of Systems and Software (JSS) 81(4), pp. 536-557, Elsevier Science, April, 2008

Abstract

Software architecture documentation helps people in understanding the software architecture of a system. In practice, software architectures are often documented after the fact, i.e. they are maintained or created after most of the design decisions have been made and implemented. To keep the architecture documentation up to date, an architect needs to recover and describe these decisions.

This paper presents ADDRA, an approach an architect can use for recovering architectural design decisions after the fact. ADDRA uses architectural deltas to provide the architect with clues about these design decisions. This allows the architect to systematically recover and document relevant architectural design decisions. The recovered architectural design decisions improve the documentation of the architecture, which increases traceability, communication, and general understanding of a system.

7.1 Introduction

Software architectures represent the design of a software system and the decomposition of a system into its main components ([11, 119, 138]). Architectural design decisions underly the software architecture ([21, 93, 95, 158]). As such, software architectures can be seen as the result of a set of architectural design decisions ([21, 77]).

Software architectures are typically described in one or more software architecture documents. Architecture documentation approaches provide guidelines on which aspects of the architecture should be documented and how this can be achieved ([29, 67, 92]). However, these approaches document only partially what an architecture is, as they lack rationale, rules, constraints, and a clear relationship to the requirements ([158, 163]). This information is valued by practitioners ([149]) and helps in future design decision making ([44]). Consequently, not only the architecture should be documented, but also its underlying design decisions.

To document architectural design decisions, two problems need to be addressed. The first problem is that the current notion of an architectural design decision is rather vague. It is unclear what is part of an architectural design decision and what is not. Consequently, this vague notion leads to problems in documenting architectural design decisions, as it is unclear what exactly should be documented.

A second problem is that, in practice, software architecture documentation is often not well maintained or not created at all ([172]). Reasons for this practice are the perceived benefits of documenting the architecture or lack thereof, and time pressure ([30]). In this practice, two distinct cases can be discerned.

In the first case, the time pressure to deliver the software is perceived to be so great that documenting is perceived as overhead. Consequently, no effort is spent to document the architecture. If an organization cares for the quality of its software and documentation, then it usually reserves some cleanup time after a deadline to update the documentation accordingly.

In the second case, an organization does not create or maintain architecture documentation. The knowledge of the architecture resides in the head of the architect. The lack of documentation then becomes an issue when the architect is no longer available, e.g. has moved to another project or company. The tacit knowledge ([112]) of the organization is no longer sufficient to (completely) understand the system. To prevent this situation from happening, the architect is usually given the task to document the architecture before he or she becomes unavailable.

In both cases, the architect tries to guess the architectural design decisions *after* they were made. Therefore, these decisions are not readily available to the architect.

Consequently, there is a need to recover them. The key issue this paper addresses is how an architect in these cases can systematically recover architectural design decisions.

The main contribution of this paper is a recovery approach, which systematically recovers architectural design decisions. The approach uses a template based on a conceptual model to describe the recovered architectural design decisions.

The rest of this paper is organized as follows; section 7.2 introduces the notion of architectural design decisions and its conceptual model. Sections 7.3 and 7.4 present an approach for recovering architectural design decisions. In section 7.5, this approach is validated with a case study. The approach is evaluated in section 7.6. Section 7.7 presents related work and the paper concludes in section 7.8 with conclusions and future work.

7.2 Architectural design decisions

7.2.1 Introduction

A notion is needed of what architectural design decisions are before they can be recovered. This section introduces our notion of architectural design decisions and presents a process and conceptual model describing them. The process model describes architecting from an architectural design decision perspective, which describes the context in which architectural design decisions are created. Complementary to this is the conceptual model, which describes the concepts that make up these decisions. In the next section, the conceptual model is used to underpin our recovery approach for these decisions.

In this paper, the used definition of an architectural design decisions is taken from [163]. It is defined as a specialization of the Merriam Webster ([175]) definition of a decision, i.e. 'a report of a conclusion':

> A description of the choice and considered solutions that (partially) realize one or more requirements. Solutions consist of a set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements.

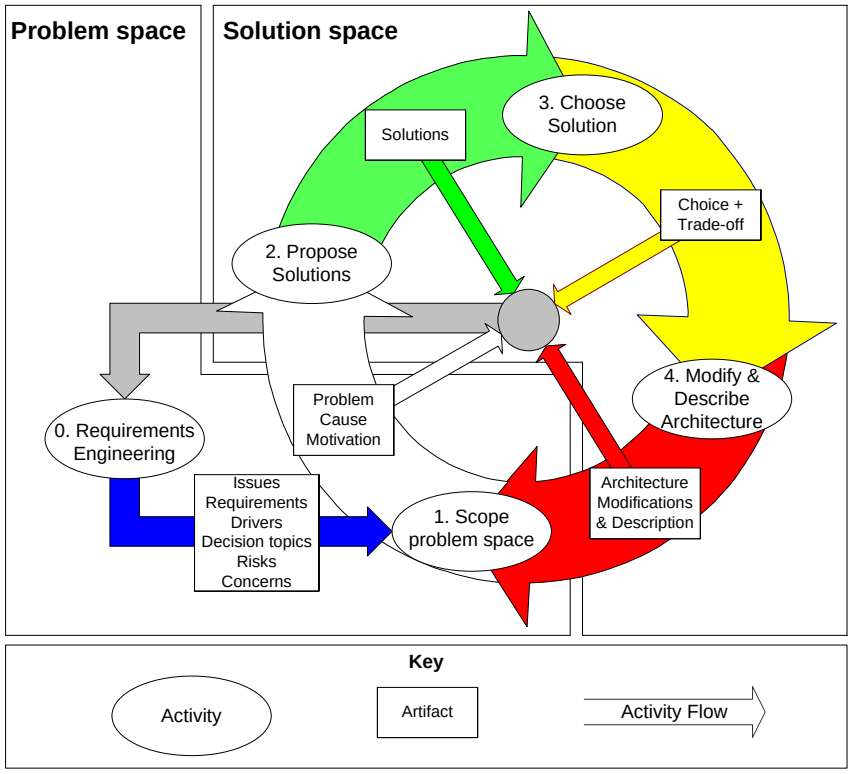

Figure 7.1: The architecting process from an architectural design decision perspective

An architectural design decision is therefore the result of a decision process that takes place while architecting. Figure 7.1 illustrates this process. It presents the context in which architectural design decisions are made. Furthermore, it visualizes the close interaction between architecting and requirements engineering, as every activity can lead to the *Requirements Engineering* activity. This is due to new insights acquired in the architecting activities, which lead to a better understanding of the problem domain. The architecting process consists of the following activities:

0. **Requirements Engineering.** Although the requirements engineering activity is not part of the architecting process, it closely interacts. Most importantly, it fuels the architecting process with different issues (e.g. requirements, drivers, decision topics, risks, and concerns) from the problem space. These elements form the main input for the activity of scoping the problem space.

1. **Scope problem space.** Based on the issues at play in the problems space the architect makes a scoping (and thereby a prioritization) of these issues and distills it into a concrete problem. To put the problem in perspective, a motivation and cause of the problem is described as well. This scoping is needed, as the problem space is usually so big that an architect is unable to address all the issues at once.

2. **Propose Solutions.** The existing architecture description and the problem of the previous step form the starting point from which the architect tries to come up with one or more solutions that might (partially) address the problem.

3. **Choose Solution.** The architect makes a choice between the proposed solutions, which can entail making one or more trade-offs.

4. **Modify & Describe Architecture.** Once a solution is chosen, the architecture description has to be modified to reflect the new status.

The architecting process presented here is a specialization of the generalized model of architecting by [66]. Their model discerns three main activities in architecting: architectural analysis, synthesis, and evaluation. From an architectural design decisions perspective, architectural analysis is the *scoping of the problem* (activity 1), architectural synthesis is *proposing solutions* (activity 2), and architectural evaluation is both *choosing a solution* and *describing the architecture* (activities 3+4).

7.2.2 A conceptual model

To refine and more formally define the definition of architectural design decisions, this section presents a conceptual model. The model describes the elements of architectural design decisions and their relationships in more detail. The presented model has been briefly presented before in earlier work ([163], [77]) for forward engineering purposes. However, in this paper the conceptual model is used for recovery purposes and presented in more detail than before. The conceptual model is used later in this paper to define the elements of an architectural design decision that should be recovered.

Figure 7.2 presents the conceptual model for architectural design decisions. For each concept of an architectural design decision, the corresponding activity of figure 7.1 is noted. At the heart of the model is the *problem* element, which together with the *motivation* and *cause* elements describes the *problem*. A *motivation* describes why the *problem* is relevant. The *cause* describes the causes of this problem.

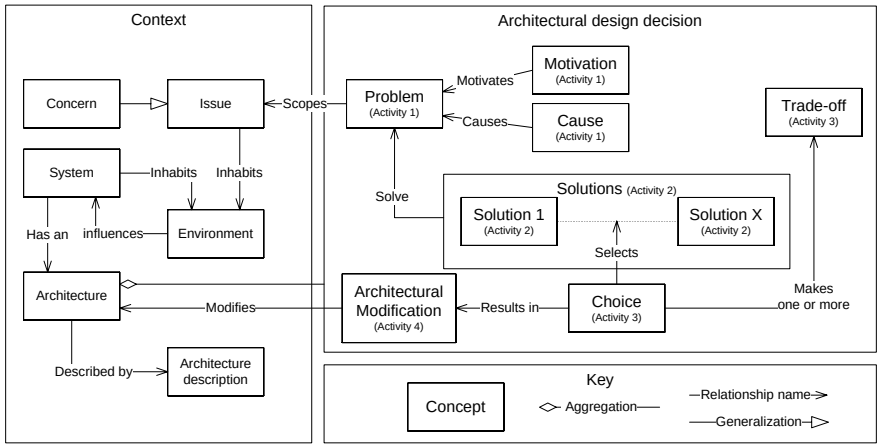

Figure 7.2: Conceptual model for an architectural design decision

Solving the *problem* is the goal the *architectural design decision* wants to achieve. The *solution*s element contains the solutions that have been proposed to solve the problem at hand. A *choice* is made as to which solution should be used and this results in an *architectural modification* of the architecture.

Most of the model elements are made in a specific *context*. Apart from the *issue* concept, all the *context* concepts (i.e. concern, system, environment architecture, architecture description) and the relationships among them come from the IEEE 1471 standard for describing software architectures ([70]). An extension is made with the new concept of an *issue*, which is a generalization of the *concern* element. This new concept is needed, as a *concern* is traceable to one or more stakeholders. For an *issue*, this does not have to be the case, as it might directly come out of the *environment*. For example, an issue might be a problematic technical constraint coming forth from an earlier design decision.

An architectural design decisions is related to the *context* in three different ways. First, the *problem* element is the scoping of various *issues* of the problem space. Second, an architectural design decision modifies with an *architectural modification* the *architecture*, which will lead to an update of the *architecture description*. Third, an architectural design decision is part of the architecture, as this is a set of design decisions ([77]).

In the remainder of this section, the concepts that make up an architectural design decision are explained in more detail. Following is a list of these concepts and their relationships:

- **Problem.** A design decision is made to solve a certain problem. For example, the problem can be how specific requirements can be met or how the design can improve on some quality aspects.
- **Motivation.** This element contains the rationale for why the problem needs to be solved. This element therefore describes the rationale behind the scoping of the problem, as it explains the importance (and thereby the prioritization) of the problem. Usually, this comes from the requirements stated for the system. Together with the *problem* element this element determines the architecture significant requirements (ASR). Since a requirement is, by definition, architectural significant if it is addressed by an architectural design decision.
- **Cause.** Often multiple causes for a *problem* do exist, this element describes them. This knowledge is important, as it decreases the chance that *solutions* are proposed that are inadequate to solve the *problem*. Causes can include technical limitations, changed requirements, limitations imposed by previous design decisions, symptoms of other problems, etc.
- **Current Architecture.** This element describes the architecture upon which the architectural design decision is made, i.e. the architecture before being modified by the decision. (see figure 7.2). The element is described by referring to the appropriate architecture description of the current version of the architecture.
- **Solutions.** To solve the (described) problem, one or more potential solutions can be thought up and proposed. For each of the proposed solutions, the following elements can be identified:

  - **Description.** This element describes the solution being proposed. The required modifications are explained and rationale for these modifications is provided.
  - **Design rules.** Design rules define partial specifications to which the realization of one or more architectural entities have to conform. This defines parts of how the solution should be realized.
  - **Design constraints.** They define limitations or constraints on the further design of one or more architectural entities. These limitations and constraints are to be obeyed by future decisions for this solution to work.
  - **Pros.** Describes the expected benefit(s) from this solution to the overall design and the impact on the requirements.
  - **Cons.** Describes the expected negative impact on the overall design, as opposed to the *Pros*.

- **Trade-off.** The different solutions have typically different impacts on the quality attributes and provided functionality of an architecture. Hence, a *Choice* should decide on one or more *trade-offs*. In some cases, these trade-offs can

be rather complex. This element describes the different quality attributes or functionalities a trade-off has to be made between.

- **Choice.** For a problem there are often multiple solutions proposed, but only one of them can be chosen to solve the described problem. The choice involves selecting different *trade-offs* using the *pros* and *cons* of the solutions as arguments to rationalize the selection of a particular solution.
- **Architectural Modification.** The chosen *solution* in the *decision* can affect one or more architectural entities and this element describes the changes to these elements.

7.3 Recovering architectural design decisions

This section introduces the **A**rchitectural **D**esign **D**ecision **R**ecovery **A**pproach (ADDRA). The approach is presented in two sections, this section presents the recovery steps that make up ADDRA, whereas the next section explains the knowledge externalization process that underly these steps.

ADDRA tries to recover architectural design decisions and documents them using the conceptual model presented in the previous section. The approach uses a combination of existing recovery techniques and the tacit knowledge of the original architect to recover architectural design decisions.

The basic idea of ADDRA is that changes in the architecture are due to architectural design decisions. Hence, these changes are a sort of clues for finding the decisions they came from. The tacit knowledge of the original architect is used to trace back from these changes to the decisions they originated from. The changes are used to focus the tacit knowledge of the architect and allow for a systematic approach of documenting this knowledge.

ADDRA consists of five steps, which are organized in an iterative process (see figure 7.3). Steps 1 - 4 are concerned with finding the changes in the architecture caused by architectural design decisions. ADDRA first recovers relevant software architecture views for different releases of the system (steps 1 - 3). The differences between the views of one release are then compared to another subsequent release forming the *architectural delta* between two releases (step 4). The architectural delta in turn is used to recover the architectural design decisions (step 5). An example of an iteration of these steps is presented in figure 7.4. Note that in the case of initial design, step 4 is skipped, as no earlier design exists and the recovered architecture views of step 3 are used as architectural delta instead.

ADDRA is performed iteratively and can be triggered at two points. The first point

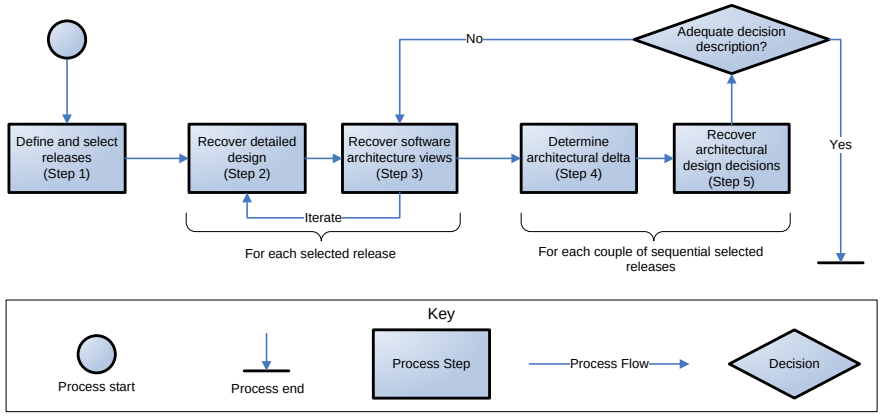

Figure 7.3: Overview of the steps of ADDRA

is in step 3 (see figure 7.3). When the information for recovering the software architecture views is inadequate, another iteration of step 2 takes place. The second point is after step 5, the the description of the recovered architectural design decisions is deemed inadequate. In this case, ADDRA returns to step 3 to sharpen the description of the software architecture views. This process continues until no more significant progress can be made.

The recovery steps of ADDRA are far from trivial. Steps 1 to 3 (and partially 4) are not completely solved yet and remain under research by the design recovery community (see also section 7.7.2). We briefly present these steps, point out potential tools, problems, and trade-offs to be made. The main focus of this paper is however on step 5; the recovery of architectural design decisions. This step is therefore presented in more detail.

7.3.1 Step 1: Define and select releases

The first step in the recovery method is to define and select the releases of the system under consideration. Releases are snapshots of the system and its associated artifacts at a specific moment in time. A selection is made to limit the amount of low-level information the architect has to deal with. In this selection process, the following concerns play a role:

Effort. Extracting design decisions from various artifacts is a *very* time-consuming operation. Therefore, a selection of the releases is made, which reduces the number of releases to examine. The exact number of releases to select is a balance between

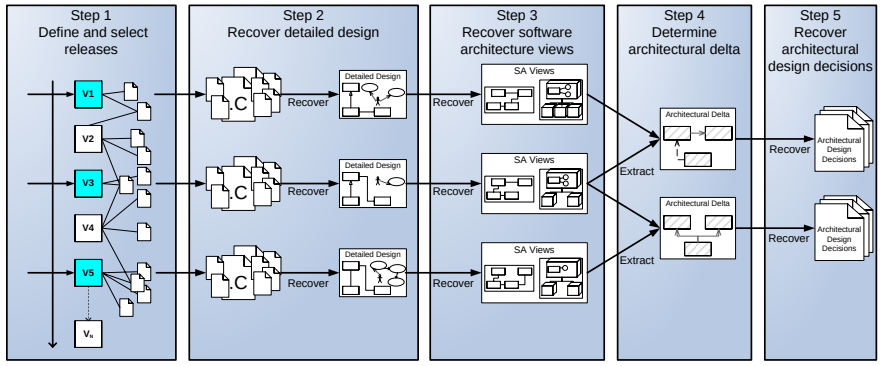

Figure 7.4: One iteration of ADDRA

accuracy and effort. Selecting more releases improves the detection of relevant changes and thus improves the accuracy of the recovery, but also takes more effort to realize.

Timing. Releases should have been an achievement target (e.g. a deliverable or a major release) in the past. If this is not the case, the release will likely be a snapshot of an unstable design situation. In this situation, design features are more likely to be in an undetermined state where a lot of design decisions are pending a decision. This will be reflected in the potential consistency and completeness among the artifacts, thus complicating recovery.

Scope. The changes between two releases (i.e. their delta), should be chosen in such a way that the scope of the delta is the right size. If the scope of a delta is too large, then the chance that multiple design decisions have been taken on top of each other becomes greater. This makes the extraction of individual design decisions more difficult. However, the opposite is also not desirable. If the scope of the delta is too small, a design decision can still be work in progress and only parts will be visible. This will obscure the results of a design decision, thus hampering the recognition of the design decision.

A rough estimation of this scope can be estimated by looking at changes in logs or log files, code metrics, and the development time between releases. The changes give an insight into the issues addressed and the scale of impact of the design decisions made between two releases of a delta.

Availability and accuracy. Both the availability and accuracy of the artifacts influence the usefulness of a particular release for the recovery process. The availability of artifacts (e.g. requirements, source code, and changelogs) for a release determines the potential value of the release. Accuracy is important as well, as inaccurate artifacts can easily lead to wrong conclusions about the system and therefore

to the recovery of irrelevant and inappropriate decisions. The different versions of the artifacts should be linked to releases, such that an overview is created of the available artifacts. After this, the accuracy of (relevant) artifacts can roughly be determined by a quick scan and use of past experiences.

An open research challenge is to determine useful heuristics that can guide the selection process. One could think of adapting existing cost prediction models (e.g. COCOMO II ([15])) to this end. However, to the best of our knowledge, no work exists in this area.

7.3.2 Step 2: Detailed design

For each selected release, the detailed design is recovered. The detailed design forms the basis for abstraction later on to recover the software architecture. Therefore, the goal is not to recover a complete detailed design, but merely a useful abstraction that can be used later on. The detailed design can be recovered with the help of one or more recovery tools. Although these tools help in supporting the recovery of the detailed design, they still require the expert knowledge and guidance of the architect to find the right abstractions ([164]).

Depending on the relevant software architecture views of the next step, one or more detailed design views might be of interest. These detailed design views are usually represented in the Unified Modeling Language (UML). The main interest is typically in recovering the class ([60]), object ([155]) diagrams, and the use-cases. The first two focus on the structural aspects of the system. Recovery tools, such as [60] and [155], can typically recover (parts of) these views. The use-cases provide hints for issues that are addressed in the system and not easy to recover with tools, although tools like the one from [125] can help in their recovery. If the dynamic aspect of the architecture is relevant, the processes, threads, and state-charts of the detailed design are of interest. Behavioral recovery tools like Discotect ([177]) are useful in these cases. More information regarding the manual reconstruction of different detailed design views and UML can be found in [17].

Although recovery tools are very helpful in recovering the detailed design, they do have some shortcomings:

- **Distributed and dynamic behavior.** Recovery tools have difficulty in recovering the dynamic and distributed behavior of programs. Therefore, the recovered detailed design by these tools is often not complete.

- **Language specific.** Most useful recovery tools are very language specific, but in practice many software systems use multiple languages (e.g. C#, C++, C). Integrating the results of the various recovery tools together is often cumbersome or impossible due to the different concepts they use.
- **Configuration.** Configuring a recovery tool for a specific system is often not trivial. Often, the knowledge of a domain expert is needed to filter out noise in the recovered results that is created by used third party components, frameworks, and the specific platform used. Furthermore, some recovery tools require extensive interaction with the architect to come to a correct configuration.

7.3.3 Step 3: Software architecture views

The third step consists of recovering one or more views on the architecture. This step is very difficult to perform and can only be achieved successfully by the original architect. The views are recovered for each selected release, thereby providing a description of the software architecture at specific moments in time. To reduce effort, only a selected number of views are recovered. For this selection, two factors are considered: the relevance of the view and the relationship to other releases. As with the selection of relevant releases, the number of views to select is a trade-off between effort and accuracy.

Relevant views are the views that describe parts of the solution(s) to the main concerns at hand. Identifying these views is based on the recovered detailed design, design documents about the architecture, and knowledge from the architect. These information sources help in determining relevant views in the following way:

- The recovery of the detailed design refreshes the memory of the architect, which can provide clues to relevant concerns. Furthermore, it provides a stable ground for abstraction to various views on the architecture, which eases the creation of these views. For example, package diagrams form a basis to abstract to a module view.

- Design documents are created with the intent to describe certain aspects of the architecture that were meant to address the main concerns at that time. Before a design document can be used, the consistency with the recovered detailed design should be checked. The document may not have been well maintained and may no longer be consistent with the design it describes. Identifying these inconsistencies provides valuable clues for design decisions later on.

- The guidance and knowledge of the architect remains the main factor in this selection process. In the end, the architect can make the best distinction between relevant and less relevant aspects and therefore mostly determines relevant views.

One set of views is needed for all the releases. In the next step of the recovery, similar views on the architecture are compared with each other. This requires that the same views be recovered for subsequent releases. Hence, the decision to create one particular view for a release influences the decision to create the same view for the previous and next selected releases. Furthermore, for these views to be comparable with each other they should conform to the same viewtype definition ([29]), i.e. use the same definitions, entities, relationships, naming, and notation. A natural way to achieve this is to use the recovered view of a previous release as a starting point for the next one and adapt it to conform to the associated recovered detailed design of step 2.

7.3.4 Step 4: Architectural delta

The architectural delta of two releases is the change in the architecture that is required to go from the architecture of one release to another. Since the identified changes result from architectural design decisions, they can be used later on for the reconstruction of the architectural design decisions (see section 7.3.5). Note, that this does include architectural design decisions that are unconsciously made. Even more so, they do not have to be made by the architect at all.

The architectural delta is determined by looking at the differences between the architectural representations of each release. The architectural views, representing the architecture, were reconstructed earlier on (see section 7.3.3). The architectural delta is view independent, but differences between sequential views are a part of the architectural delta. The differences between these views *together* form an approximation of the architectural delta. To create this approximation, each entity (e.g. component, connector, module, uses relationship, etc.) found in the views is classified in one of the following five categories (this is similar to the work of [113] for UML diagrams):

- **Unmodified** entities have not been modified during the evolution. These elements are not part of the architectural delta, but represent the stable part of a system during change.
- **Modified** entities have been modified due to architectural design decisions. However, the entities still exist in the same view of both releases.

- **New** entities are entities, which did not exist in the oldest of the two releases, but does exist in the newest of the two. These entities represent new elements introduced into the design by architectural design decisions.
- **Moved** entities are entities that have been deleted in the new design. However, the aspects they represented are still in the design, but are now represented by other entities. Typical unit operations ([10]) that result in entities being moved are abstraction, uniform decomposition, and compression of entities.
- **Deleted** entities represent aspects that are no longer relevant for the new design. These entities are not available anymore in the new design (not even moved), but did exist in the old design.

Inspecting the differences in the views determines to which category an entity belongs. For each couple of sequential releases the corresponding views are compared with each other. The identified differences between both views discriminates the entities in separate groups: *(un)modified, new, deleted* or *moved*. *(Un)modified* entities are entities available in both views. *New* entities are only available in the view of the latest release. On the other hand, *deleted* or *moved* entities are only available in the view of the first of the two releases.

Discriminating between *moved* and *deleted* entities is complicated and has no clear solution. To make this distinction we use the implicit knowledge of an architect. The architect can identify potential relationships between a *new* entity and a *deleted or moved* entity. *Deleted or moved* entities falling outside this group can be marked as *deleted*. Further detailed design or even code inspections are used to determine the true nature of the remaining *deleted or moved* entities.

The architectural delta can be visualized for each viewpoint the entities have been classified for. For each of the type of entities defined in the viewpoint five different representations are made, which represent their classification. To visualize the changing aspect of the architectural delta for a particular viewpoint, unmodified entities are left out of the visualization. An exception is made if these entities are needed for providing a context for other entities, e.g. changing relationships. Corresponding numbers on the entities express the relationship between the moved entities and the new or modified entities incorporating them. An example of this notation for the module view is presented in the bottom part of figure 7.9, which also demonstrates the use of the correspondence numbers.

7.3.5 Step 5: Architectural design decisions

In this last step, the architectural design decisions are recovered. The architectural design decisions are recovered using the template of section 7.2. This requires that

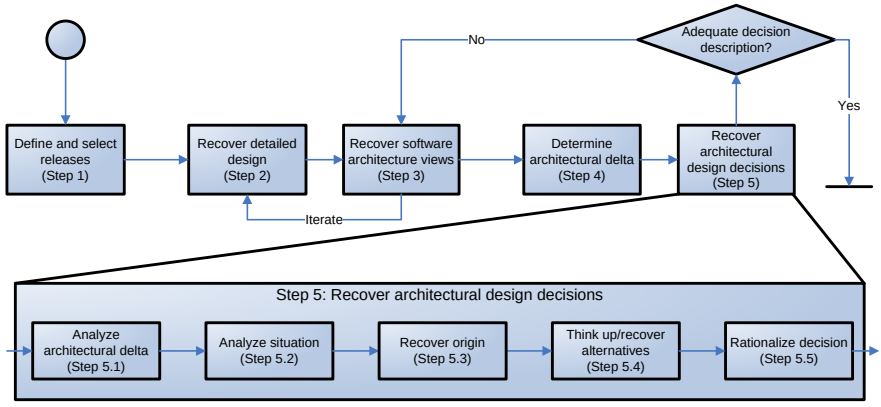

Figure 7.5: Overview of the externalization process of step 5

the knowledge of the architect is externalized into a documented form. This is achieved with the architectural delta and a supporting externalization process.

Figure 7.5 presents an overview of this externalization process. It illustrates the various activities and the order in which they are executed. The remainder of this section explains each step of this process. To execute these steps, several supporting knowledge transformations are needed, which are presented at the end of this section.

7.3.5.1 Step 5.1: Analyze architectural delta

The first step is to examine the architectural delta for clues, that is: find changes in the delta that can be related together to form an architectural modification of a design decision.

7.3.5.2 Step 5.2: Analyze situation

In the second step, the situation is analyzed and three intertwining and interfering activities occur: the definition of the context, describing the solution the architectural modification represents, and identifying which problem the modification tries to tackle. The last activity tries to recover the scope of the problem space that this decision addresses (see section 7.2). Often this will entail issues coming from the (partial) fulfillment of one or more requirements. Once these three activities are completed, a fitting name describing the design decisions is defined.

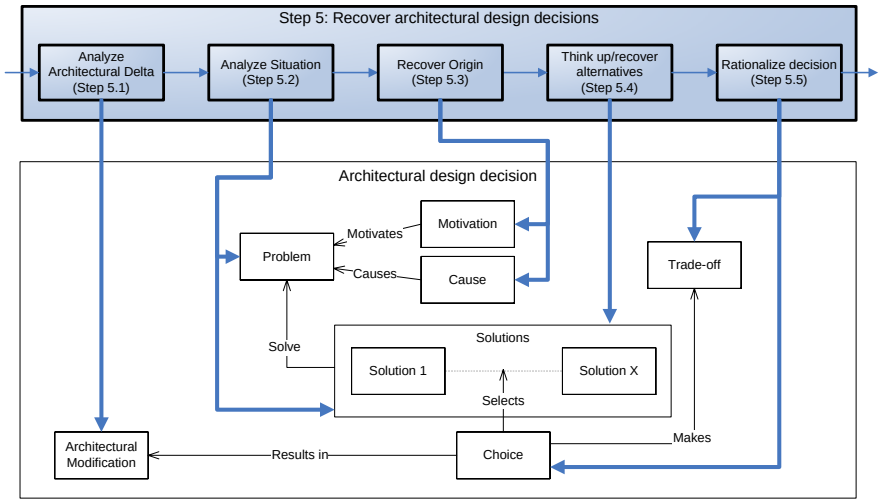

Figure 7.6: The externalization process in relationship to the conceptual model

7.3.5.3 Step 5.3: Recover origin

The origin of the decision is then recovered by determining the cause and the motivation of the problem at hand. Often a problem comes from the (partial) fulfillment of one or more requirements. In these cases, requirements form an excellent motivation of the problem. Hence, the motivation element creates traceability between the requirements and a part of the software architecture. Both motivation and cause provide a basis for the further refinement of the solution found, as the relationship with the problem space becomes clearer (and thereby the comprehension of the solution).

7.3.5.4 Step 5.4: Think up/recover alternatives

The next step is to think up or recover alternative solutions. Although several architectural solutions can be proposed in an architectural design decision, most of the time only one will actually be implemented (and therefore documented). For the reconstruction of architectural design decisions, this poses a problem, because information about the solutions not chosen is often unavailable in the artifacts. Consequently, both the rationale of the Decision element (see figure 7.2) and the alternative solutions are lost.

Exceptions are architectural design decisions in which earlier made choices are undone. Due to lost rationale, reverting a choice is often a costly operation. Con-

sequently, considerable effort is spent on motivating why the old choice is not appropriate anymore and a potential new one is. In this process, implicit knowledge of the design is made explicit.

A similar approach can be taken for the lost alternative solutions and rationale, but an alternative exists as well. *New* alternatives can be thought up and considered, as long as they fit in the current architecture and address the problem of the design decision. Since the goal of documenting the architectural design decisions is to make the architecture more comprehensible it does not matter where the rationale comes from ([118]). A discussion about the limitations that this imposes is presented in section 7.6.2.3.

Intertwined with the definition of alternative solutions is the description of the pros and cons of each solution. Often, this activity leads to insights into new potential alternative solutions. A good source for pros and cons is formed by the expected consequences a solution will have on the quality attributes of the architecture. The expected consequences can be based on application generic knowledge ([98]), which is typically documented in patterns ([64]).

7.3.5.5 Step 5.5: Rationalize decision

In the last step, the rationale of the decision element of the architectural design decision (see section 7.2.2) is determined. This rationale should describe the reasons for choosing a particular solution. The previous step recovered the pros and cons for each solution. Based on this, the different trade-offs are determined that are made between these solutions. The rationale of the choice is based on the trade-off(s) being preferred in the chosen solution. Hence, we have to identify these preferred trade-off(s). Usually, these trade-off(s) come from the pros of the selected solution, as a design decision is intended to improve the design. Note however, that the rationale might be different from the original used rationale due to a difference in the considered alternatives, but as pointed out in the previous step this is not a problem.

7.4 The knowledge externalization process

So far, we have explained the steps that need to be taken to recover architectural design decisions. However, the issue *how* the knowledge externalization process used in these steps should be performed has not been answered. To address this issue, we use Nonaka's theory of knowledge creation ([112]). This theory distinguishes two types of knowledge: tacit and explicit knowledge. Tacit knowledge is

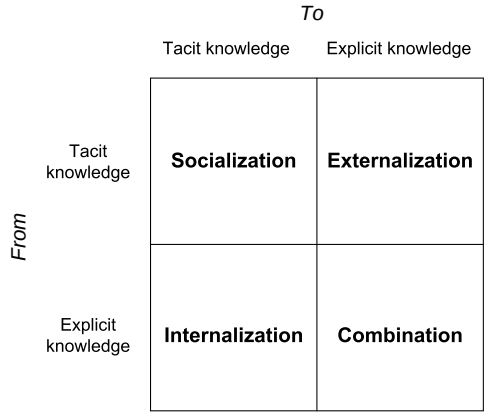

Figure 7.7: [111] four modes of knowledge conversion

the knowledge inside the heads of people, explicit knowledge is knowledge captured in documents, models, etc. One can convert one type of knowledge into another one by means of a knowledge conversion. Figure 7.7 presents the knowledge conversions Nonaka distinguishes. In short, these are the following:

- **Externalization** is the process of articulating tacit into explicit concepts, which is the main focus of this paper.
- **Internalization** is the process of embodying explicit knowledge into tacit knowledge.
- **Socialization** is the process of sharing experiences and thereby creating tacit knowledge, such as a shared mental model and technical skills
- **Combination** is the process of systemizing concepts into a body of knowledge (e.g. a document).

Based on these knowledge conversions, we identified four techniques for supporting the externalization process when recovering architectural design decisions. These techniques are often combined and used together in each of the different steps of the architecture design decision recovery process (see figure 7.5). The four techniques we identified are the following:

- **Introspection** is a form of externalization in which the architect asks him/herself questions and tries to come up with answers.
- **Inspection** is a combination of internalization and externalization. Inspection of various artifacts improves the understanding one has and allows one to express this knowledge.

- **Discussion** is a combination of socialization and externalization. During discussion tacit knowledge is shared, which is later made explicit.
- **Generalized domain knowledge** is a form of combination, where the situation at hand is combined with generalized domain knowledge from literature.

In the externalization process, the starting technique is often introspection by the architect. This can give rise to questions that cannot be directly answered. Hence, these questions require inspection or the use of generalized domain knowledge to be answered. The discussion technique is useful for validating the results of introspection. Furthermore, it can provide direction, similar to the generalized domain knowledge, when the knowledge of the architect is not sufficient. In the remainder of this section, each technique is explained in more detail.

7.4.1 Introspection. (Externalization)

The main technique for transforming tacit knowledge into explicit knowledge is introspection or reflection. The architect asks him/herself questions and tries to come up with answers (see section 7.6.2.1 for a discussion when the architect does not perform the recovery). Two distinct types of introspection can be discerned:

Knowledge of design under consideration. In this case, introspection examines the tacit knowledge of the architect about the design being recovered. This is primarily used to recover the line of reasoning once used or ton rationalize a new one. If certain questions cannot be answered, these questions are a starting point for inspection. Typical questions for the architect to ask him/herself in this case are:

- Why is this particular change made?
- What was the problem the change tried to resolve?
- Which requirements are involved?
- Which stakeholders were involved with this change/problem and what are their stakes?
- Are there other design decisions involved?

Knowledge of similar designs. An alternative form of introspection is to use past design experiences. Contrasting the situation at hand with earlier made designs can provide valuable knowledge. Typical questions to ask in this case are:

- Where does this situation deviate from similar past designed systems? Why are these deviations made?
- Where is this situation similar to past designed systems? What kind of decisions are always made in these cases? Why are these decisions made?

7.4.2 Inspection. (Internalization + Externalization)

Inspecting the architectural delta provides clues of architectural design decisions. These clues form the basis for introspection and discussion, as they provide a context and focussing point. From our experience, the following things are fruitful to inspect:

- Isolated changes often form a good starting point to unravel architectural entities that are affected by multiple changes. Inspection helps in determining whether the change is isolated or is a combined effect with other changes of the delta that are due to the same cause.
- Design decisions are typically made to refine earlier ones. Therefore, it is useful to inspect relationships between a change and other previous or further changes to the same and related architectural elements.
- Every change originated from a design decision. If there are still unaccounted changes, closer inspection of the commonalities and differences usually provides the required insight.

7.4.3 Discussion. (Socialization + Externalization)

Discussion entails explaining, defending, and arguing about design decisions. This provides a natural way to trigger the externalization process. Furthermore, the interaction with others decreases the chance on a narrow minded line of reasoning. A good basis for discussion includes the following points:

- Discussion about problems. Is the problem really the problem or are there underlying issues leading up to different problems?
- Brainstorming about potential alternatives, which is also done in ATAM ([11]).

7.4.4 Generalized domain knowledge. (Combination)

Another technique for supporting the externalization process is comparing and contrasting the situation at hand with generalized situations known in the domain, i.e. use application generic knowledge ([99]). This includes using the knowledge from:

Architectural styles and patterns. Styles and patterns can be seen as distilled generalized architectural design decisions ([64, 77]). If (part of) the delta can be identified as a derived result from a style or pattern, the documented knowledge of these styles and patterns is useful. This documentation contains further directions on issues and tradeoffs that need to be addressed when applying them. Consequently, this provides concrete guidelines on the decision making and therefore the architectural design decisions that the delta is a result off. Good guiding questions in this perspective are the following:

- How is this solution specialized from the general style?
- What is the documented rationale to use such a style or pattern?
- How are the future directions tackled of a style or pattern addressed?

Tactics Tactics describe architectural solutions for improving specific quality attributes of an architecture ([11]). This provides the opportunity to trace back an architectural solution to a problematic quality attribute. The question to answer in this case is:

- Does the architectural delta look similar to the architectural solutions proposed in a tactic?

Reference frameworks or platforms. For specific domains, reference frameworks or platforms (e.g. .NET, J2EE) are available. They provide standard solutions to common problems in these domains. The same issues are often addressed in the system under examination. Furthermore, reference frameworks and platforms often provide explicit variability for critical design decisions, e.g. the type of scheduler to use in a real-time operating system. The system under examination can be examined to see if it makes these critical design decisions as well.

7.5 Case study: Athena

ADDRA is validated in this section by applying the approach on a case called Athena. First, an introduction to the case is presented, followed by a description of the application of the different steps of ADDRA in subsequent subsections.

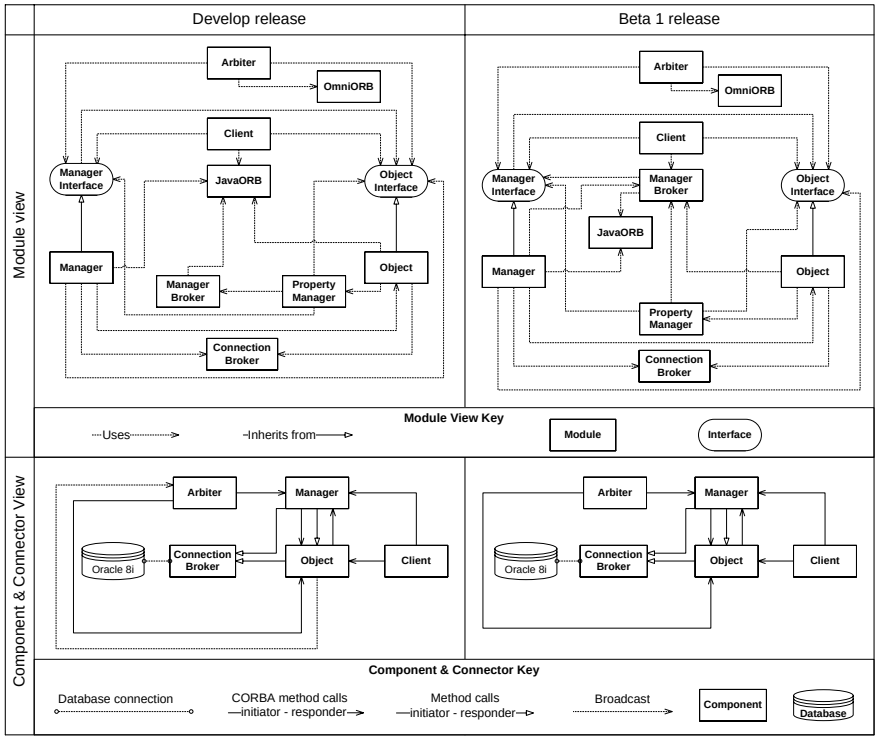

Figure 7.8: Two views of the `develop` and `beta 1` releases of Athena

Athena is a system for (automatically) judging, reviewing, manipulating, and archiving computer program sources. The primary use is supporting students to learn programming. To develop the programming skills of a student, he or she has to practice a lot. Small programming exercises are often used for this end. However, providing feedback on these exercises is a laborious and time-consuming effort.

Athena helps students by testing their solutions to functional correctness and provides feedback (e.g. test results, test inputs, compilation information etc.) on this. The system is capable of providing personal feedback to large groups of students in a relatively short time. Consequently, the speed and quality of learning is increased when using a supportive submissions system ([75]). For a more elaborate description of the case, we refer to [80].

7.5.1 Step 1: Define and select releases

The first step in the recovery approach is to identify and select relevant releases (see section 7.3.1). During the evolution of Athena, all documents and code artifacts were maintained in a version control repository. This enabled us to track the evolution of the Athena artifacts available in the repository and thus the software system itself.

To select appropriate releases in the Athena case, code metrics, log files, and release cycle time information were used. However, design documents were not available. Based on this information most of the design decisions seem to be taken during the initial development of Athena.

Although the case study is much bigger, the focus of this paper is on two releases in the initial development. The first is a prototype release named `develop`, the second is the first beta release called `beta 1`.

7.5.2 Step 2: Detailed design

For both releases, the software architecture was recovered based on the detailed design. A case tool was used to inspect the source code and the relationships of the various classes. While reconstructing the detailed design a big sheet of paper was used to record classes of interest and their relationships. Already recurring design patterns were discovered, which could be used to group detailed design entities together. In similar ways, abstract super classes, the relationship between internal and external provided modules, and the internal code organization delivered parts for the abstraction to the architectural views.

7.5.3 Step 3: Software architecture views

Figure 7.8 presents for both releases `develop` and `beta_1` the resulting module and component & connector views ([29]). The views contain the following relevant architectural entities:

- ***OmniORB, JavaORB*** general functionality needed for the CORBA communication
- ***Arbiter*** automatically tests and judges submissions of students.
- ***Client*** tools for sending in submissions, viewing results, and configuring the Athena system.

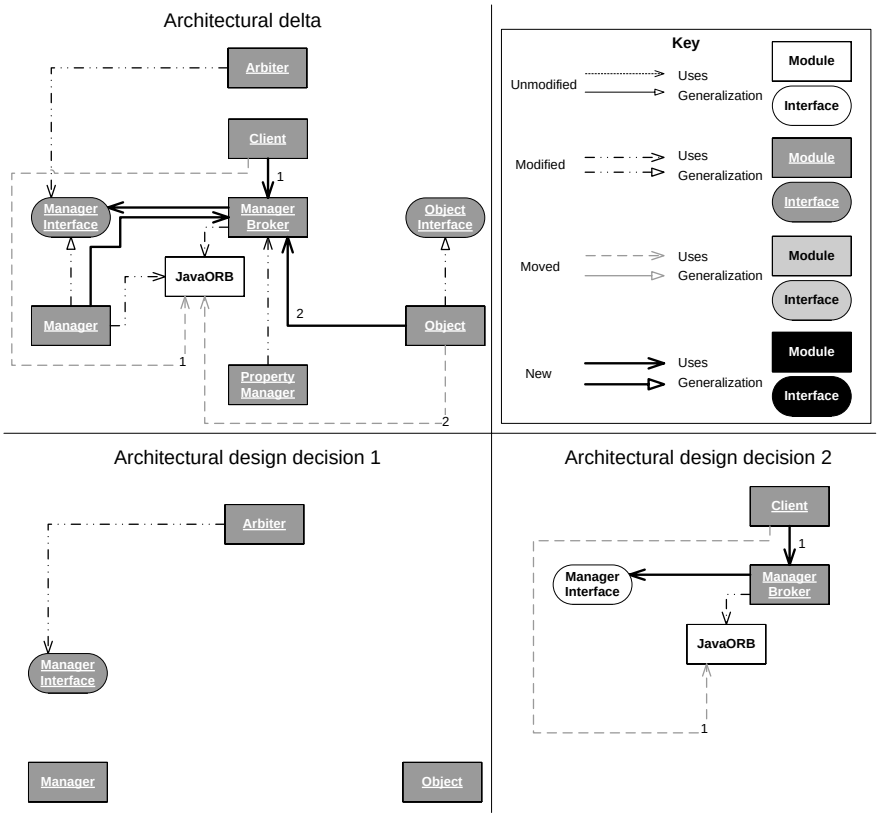

Figure 7.9: The architectural changes between releases `develop` and `beta_1` (top left) caused by architectural design decisions. Examples are the architectural module modifications of architectural design decision one (bottom left), and two (bottom right)

- ***Manager Interface*, *Object Interface*** interface between *Client* and server (*Manager*, *Object*)
- ***Manager*** handles queries, creation, and destruction of specific *Object*s
- ***Object*** representation of the domain objects, e.g. submissions, students, tests, test sets etc.
- ***Connection Broker*** Database abstraction and convenience layer to access the *Oracle 8i* database.

7.5.4 Step 4: Architectural delta

To determine the architectural delta between the two releases, the views on the architecture of Athena are used. First, the elements of the views are compared

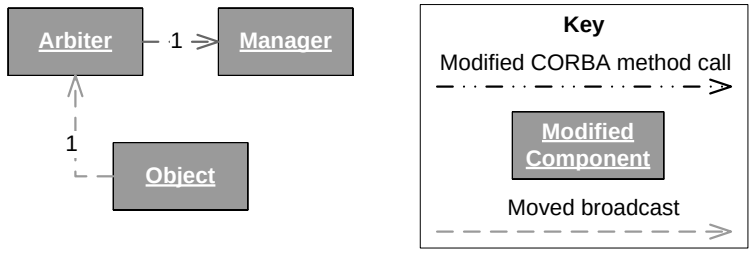

Figure 7.10: Architectural delta of the c&c view of releases `develop` and `beta_1`

for subsequent releases and classified in one of the following groups: *(un)modified*, *new*, or *deleted / moved*. For the module view (see figure 7.8) this is done by looking at the differences between the `develop` and `beta_1` releases. Inspection of both leads to the identification of two *deleted / moved* uses relationships, four new uses relationships, and the rest being classified as *(un)modified*.

The next step is to discriminate between *deleted* or *moved* entities. For example, the case of the uses relationship between the *Client* and *JavaORB*. Closer inspection of the *Client* reveals that this relationship has been *moved* to the *new* uses relationship with the *Manager Broker*.

The distinction between *modified* and *unmodified* is partly based on the previous *moved* uses relationship. Substantial changes to the *Client* and *Manager Broker* have been made. Furthermore, the four *new* uses relationships have significantly affected some of the involved modules and interfaces, which are therefore marked as *modified*. The other elements have not significantly changed and are therefore classified as *unmodified*. The resulting architectural delta of the module view visualized in figure 7.9. In a similar way, the architectural delta of the component & connector view is constructed, as illustrated in figure 7.10.

7.5.5 Step 5: Architectural design decisions

In this case study, a total of 15 different architectural design decisions have been recovered with ADDRA. Figure 7.11 presents an overview of these decisions. Between releases `develop` and `beta_1`, three different architectural design decisions were identified. Two of these decisions, the *Arbiter Job Control* and *Managing Manager References*, are presented here, while the third decision (the Domain Object Delegation decision) is left out due to space constraints. The other 12 architectural design decisions have been recovered from the initial *develop* release.

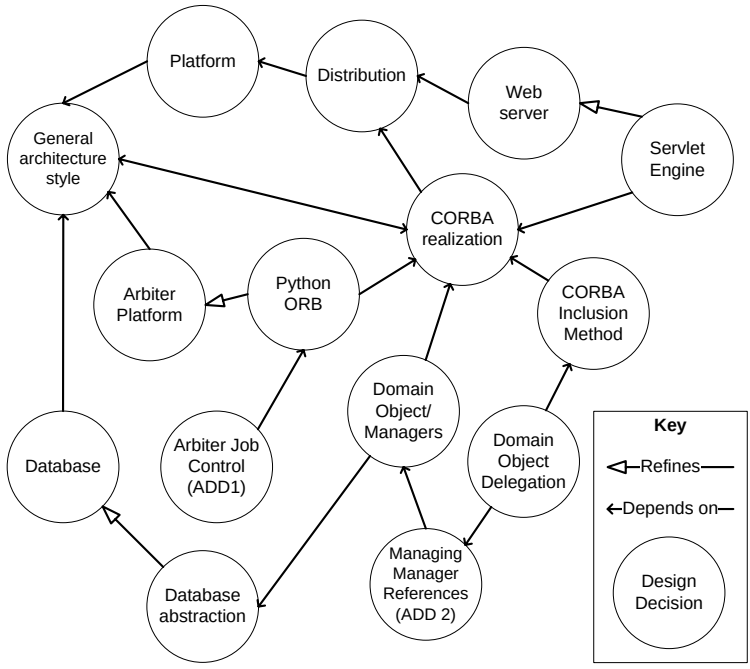

Figure 7.11: The recovered architectural design decisions in the Athena case study.

For the *Arbiter Job Control* decision we exemplify each of the steps of ADDRA for the recovery of this decision. An in-depth description of the *Arbiter Job Control)* and *Managing Manager References* decision is presented at the end of this section.

Step 5.1: Analyze Architectural Delta. *Inspection* of the architectural delta of the component and connector view (figure 7.10) shows a moved broadcast relationship between the *Object* and *Arbiter* to modified CORBA method calls between the *Arbiter* and the *Manager*. For the delta of the module view *inspection* reveals that the corresponding modules (i.e. *Arbiter*, *Manager (Interface)*, *Object*) have been modified as well. Next, the visible changed relationships (i.e. modified, moved, and new) of these modules are inspected, which uncovers only significant changes in the uses relationship between the *Arbiter* and *Manager Interface* interface that have to do with the moved communication between the *Object* and *Arbiter*. Hence, the **architectural modification** identified is displayed in the bottom left of figure 7.9 for the module view and in figure 7.10 for the component & connector view.

Step 5.2: Analyze Situation. Further *inspection* revealed that the moved broadcast communication was used to notify the *Arbiter* of new submissions. The new **solution** instead lets the *Arbiter* poll the *Manager* for new submissions.

The problem this solution addresses is found with the help of *introspection*. We remembered that we ran performance tests, which uncovered a problem with the response time of the system when new submissions were created by students. This did not scale well with the number of *Arbiters* being deployed, as at any moment in time only one *Arbiter* was judging submissions, whereas the other *Arbiters* were idling. The intention of the architecture in this context is to process the submissions in parallel, which does not take place.

Step 5.3: Recover Origin. The **motivation** for this problem is found in a requirement with *introspection*, which originates from *knowledge of similar systems*. Past experiences with ACM programming contests showed such systems to have problems with providing timely feedback due to performance issues. As such, a requirement was defined that a submission should be tested and reported within 10 seconds to provide timely feedback to the students.

The **cause** of this performance problem is found by *discussion* among the developers and *inspection* of the source code responsible for the notification of the *Arbiters*. It turns out that the CORBA broadcast notifications work synchronously although they are defined as asynchronous. Further *inspection* reveals the cause of this behavior, which is presented at the end of this section.

Step 5.4: Think up/recover alternatives. The polling solution that was recovered circumvents the problem by using a different communication strategy. An alternative **solution** that can be thought up is to address the cause of the problem, i.e. fix the CORBA broadcast implementation to work asynchronously. In this case, a different design strategy (a form of *generalized domain knowledge*) is used to think up this alternative.

Step 5.5: Rationalize decision. The last step is to rationalize the **choice** by making a **trade-off** between the pros and cons of the solutions. Comparing the quality attributes of the two solutions (polling versus notification) finds differences in performance, maintainability, and ease of implementation (a form of costs). Of these three there is a trade-off between cost versus performance between the two solutions, as the broadcast solution has a negative impact on the costs and a positive influence on the performance and the polling solution has this influence and vice versa. With the help of *introspection* we know that the ease of implementation was the deciding factor to choose for the polling solution, as the pressure was high to deliver on time.

The other decisions were recovered in a similar way as the *Arbiter Job Control* decision. To document these architectural design decisions, a template is used that is based on the conceptual architectural model (see figure 7.2). Note, that the template presents the various concepts in a logical order, whereas ADDRA recovers

these elements in a different order (see figure 7.6).

The template uses a condensed style, where each element of the conceptual model is typeset in bold, followed by an appropriate textual description of the element. An alternative style is to use a tabular approach, which takes up more space and is therefore not used in this paper.

Two noticeable adaptations are made to improve the template. First, the architectural modification is not documented as text, but uses the graphical notation for visualizing the architectural delta (see section 7.3.4). Second, the solutions rationalized afterwards (see section 7.3.5) are marked with an asterisk ($*$), thereby making an explicit distinction with recovered solutions. In the remainder of this section, the *Arbiter Job Control* and *Managing Manager References* decisions are presented in full detail using the template.

Architectural design decision 1: Arbiter Job Control

Problem: The response time of the system does not scale with the number of *Arbiter*s deployed. When running multiple *Arbiter*s only one judges submissions, the rest is waiting idle for work. **Motivation:** Multiple *Arbiter*s are needed to reach the performance requirement of a *10 second response time*. The current situation makes additional *Arbiter*s of no use. **Causes:** The CORBA broadcast facility that was used does not work as expected. The *Arbiter* is implemented in python, which in its core is a single threaded language. The *OmniORB* waits for the *Arbiter* to receive a broadcast message from a *Object*. Only after this has been done, the *OmniORB* will send an acknowledgment back to the *Object*. The broadcast is implemented in such a way that only after an acknowledgment is the next receiver of a broadcast notified. This results in the other *Arbiter*s having to wait in line to process the broadcast. All the while, they can be idle, waiting for new work to arrive. **Current Architecture:** This design decision is made on basis of the software architecture as represented by the views for the *develop* release in figure 7.8.

- **Polling Solution**
  Description: The *Arbiter* no longer receives broadcasts for new submissions from the *Object*. This solution uses a polling based approach instead. In this approach, the *Arbiter*s poll one of the *Manager*s for new work. If there is work, the work is returned as a pointer to the corresponding submission. **Design rules:** *Manager* provides a method for polling available submissions. **Design constraints:** None

Pros: + Relatively easy to implement.
Cons: - Load on the database increases.
- Scalability of the number of *Arbiter*s decreases.

- **Broadcast Solution***
  Description: The used CORBA broadcast facility is implemented in the *JavaORB*, which is a third party open source component. The solution is to modify this component in such a way it no longer waits on an acknowledgement before sending out the broadcast to the next receiver. Two threads are created, one that deals with sending out the broadcasts and one for handling the acknowledgements of these broadcasts. **Design rules:** The system should use the modified *JavaORB*. **Design constraints:** None

  Pros: + Very good response performance of the *Arbiter*s.
  Cons: - Maintenance increases; another component that evolves should be kept in sync.
  - Difficult to realize, as knowledge is required about the internal workings of the *JavaORB*.

Trade-off: Cost versus performance. **Choice:** The *Polling Solution* was chosen because it is easy to implement (i.e. has lower costs) and the expected negative impact on performance is not too great. The choice makes a trade-off in favor of costs (i.e. ease of implementation), as opposed to performance. **Architectural Modification:** From a module view perspective, the bottom left of figure 7.9 visualizes the changes and figure 7.10 does the same for the component & connector view.

Architectural design decision 2: Managing Manager References

Problem: The various *Client*s all use their own code to initialize the *JavaORB* and resolve the needed references to the *Manager*s. **Motivation:** Removing this duplicate code reduces the chance of errors and makes it easier to replace the CORBA implementation (*JavaORB*). Maintenance costs are also reduced, because only one instance of the code has to be maintained. **Causes:** There is currently no standard way to resolve references to the various *Manager*s. **Current Architecture:** The architectural design decision is made in the context of the architecture after the *Arbiter Job Control* had been made. The module view is therefore the original module view of the `develop` release (see figure 7.8), modified by the architectural modification of this decision as visualized in the bottom left of figure 7.9. The component & connector view of the software architecture is the one depicted in figure 7.8 for the *beta 1* release.

- **Caching Solution**
  Description: Remove the duplicate initialize code of the *JavaORB*. Add an extra layer in the form of the *Manager Broker*. This module can resolve references to the different *Manager*s without any CORBA related information for the top layer. The *Manager Broker* caches the reference of a *Manager* for further repeated use. The caching is introduced to minimize communication overhead between the clients and the naming service providing the location of the *Manager*s. **Design rules:** None **Design constraints:** All access to the *Manager*s has to be done through the *Manager Broker* for the *Client*s.

  Pros: **+** Removing duplication increases maintainability.
  + Caching increases performance.
  Cons: **-** Caching decreases availability

- **Layer Solution***
  Description: This solution is similar to the *Caching Solution*, but without the caching part. **Design rules:** None **Design constraints:** All access to the *Manager*s has to be done through the *Manager Broker* for the *Client*s.

  Pros: **+** Removing duplication increases maintainability.
  + Solution is easy to realize.
  Cons **-** Performance decreases due to extra layer.

Tradeoff: Performance versus availability. **Choice:** The *Caching Solution* was chosen, as performance is a major issue in this system and the unavailability of the system during upgrades is not regarded as problematic. The choice makes a trade-off preferring performance over availability. **Architectural Modification:** No modifications are visible in the component & connector view. For the module view, figure 7.9 visualizes the made architectural modifications.

7.6 Evaluation

The previous sections explained and exemplified the working of ADDRA. In this section, ADDRA is evaluated from three different perspectives. The first perspective is formed by the lessons learned while applying ADDRA on the Athena case. The second perspective presents the limitations of the approach. The third and final perspective outlines the benefits of ADDRA.

7.6.1 Lessons learned

7.6.1.1 Transitions between design decisions

The first lesson learned is the existence of transitions between design decisions. If an architectural design decision is undone and another (new) solution is chosen instead, a transition period exists. During this transition period, the design and the implementation will be incrementally adapted to the new (chosen) solution. It is not uncommon for these transition periods to last several months or even years, as business gives a higher priority to other decisions. If such a transition spans a period over one or more releases then it becomes part of the architectural delta of those releases. Consequently, there exists a transition period in which both solutions *coexist* and are partially realized, but as a *whole* realize the complete functionality.

Design decision models, including our own conceptual model, do not support these transitions, as these models view the transitions as atomic actions. To capture these transitions, an architectural design decision model should therefore support a more evolutionary model that tracks the transition phases between design decisions.

Transitions between design decisions hinder architectural design decision recovery, because they complicate the architectural delta. The architectural deltas of two releases can contain "leftovers" of design decision solutions, which are still in transition to a new solution. Choosing the releases relatively far apart from each other in step 1 of ADDRA partially negates this problem. However, due to the possible long durations of the transition periods this might not always help.

7.6.1.2 Architectural views are subjective views

The second lesson learned is the subjectivity of architectural views. This is not a specific problem for ADDRA, but a general problem for recovery. Architectural views are subjective views, because abstraction is used to construct them. When architectural views are (re-) constructed, abstraction choices are made. Although the concepts a view should visualize are defined, the clustering used while abstracting is not. This leaves space for different interpretations of the system, i.e. non-deterministic reconstructing results for various views ([164]).

The reason for architectural representations to be subjective is the absence of architectural entities as first class citizens in the lower abstraction levels. No explicit relationship exists between the architectural entities (and the representations of them) and the entities implementing these architectural entities. A (standardized) first class representation of architectural entities could remedy this situation.

The subjectivity of architectural views has two important consequences for ADDRA. First, the architectural views might not be fully comparable to each other, as for the same entities different abstraction choices might be made. This results in an *approximation* of the architectural delta, which in turn hinders the recovery of architectural design decisions. ADDRA tries to counter this effect by reusing the abstraction choices of previous releases (see section 7.3.3).

Second, the subjectivity of the abstraction choices makes the outcome of ADDRA partially subjective as well. This is because different people can make different abstraction choices, which can result in different architectural views on the selected releases. Therefore, they can recover a different approximation of the architectural delta and consequently can recover different architectural design decisions. ADDRA tries to counter this effect by employing the architect as the authoritative source for the abstraction choices. However, this does limit ADDRA to the availability of the architect (see also section 7.6.2.1).

7.6.1.3 Solutions are sketchy or incompletely defined

The third lesson learned is the incomplete and sketchy way in which solutions are defined in an architectural design decision. Solutions can only give a general and incomplete description of how the problem at hand should be solved, as limited resources (e.g. time) only allow for an abstract description. Later in the project the solutions that are actually used are further refined.

Most of the knowledge systems and design decision models have support for this refinement process (see also section 7.2). However, architectural representations lack the ability to represent this specialization process, because they focus on the *result* of the specialization process, not the individual steps. Some initial work dealing with this issue is the work of [132], which tracks the evolution of an architecture description. Combining such an incremental architecture view with design decisions has been first done in our own work ([77]) and has been refined by [26].

For ADDRA this refinement process is not a problem, as ADDRA is performed after the fact. In other words, most of the necessary refining design decisions have already been made. However, as ADDRA does not recover these refining design decisions in the lower abstraction levels, the description of the solution in the recovered architectural design decision remains sketchy and incomplete. Therefore, traceability to these refining solutions is limited, which makes the recovered solutions sketchy and incomplete.

7.6.2 Limitations

ADDRA is not without limitations. In this subsection, the foremost limitations of ADDRA are outlined together with potential strategies how they could be addressed.

7.6.2.1 Availability of the architect

One important limitation of ADDRA is the assumption that the architect himself or herself performs the recovery process. This assumption is important, as ADDRA explicitly employs the tacit knowledge of the architect in the recovery process. For example, in step 5, the tacit knowledge of the architect is used heavily to transform the architectural delta into recovered architectural design decisions (see section 7.4). However, in practice, an architect usually does not have the time to perform an elaborate recovery as outlined in ADDRA.

A potential solution is to use other people for the time consuming externalization and process. Although this reduces the externalization effort of the architect, additional socialization is needed to share the architectural knowledge of the architect with these people. Apart from the decision at which releases to look, steps 1 to 4 of ADDRA (see section 7.3) could be performed by others. However, this does pose a risk to successful recovery, as the architectural delta might be misleading due to the subjectivity of the underlying architecture views (see section 7.6.1.2).

The last step of ADDRA, step 5, requires the most tacit knowledge of the architect and is therefore hard to delegate to others. Supporting techniques (see section 7.4), used in this step that can be partially delegated are the inspection and use of generalized domain knowledge. For as both take explicit knowledge as input, which can be easily shared. The other two techniques, i.e. introspection and discussion, use tacit knowledge as input. Therefore these techniques require active participation of the architect, which makes it difficult to delegate them.

7.6.2.2 Selection of presented architectural views

In the Athena case study, ADDRA was presented using the module and component & connector views. However, ADDRA can be used for other architectural views as well. The idea behind the architectural delta (step 4, see section 7.3.4) and the architectural design decision recovery techniques based on this delta (step 5, see section 7.3.5) are view independent. In other words, they could also be used for other views. For example, a view from the allocation viewtype like the deployment view ([29]).

7.6.2.3 Lack of alternatives and trade-offs

ADDRA cannot always recover the considered alternatives and trade-offs of an architectural design decision. This is due to the absence of traces and identifiable effects of these decisions in the available explicit knowledge. Although ADDRA tries to recover these decisions by using tacit knowledge (see section 7.4), this recovery is not always successful. In these cases, the tacit knowledge of the architect is inadequate, which is mainly due to the forgetful nature of the human brain.

The solution used in ADDRA for these cases is to think up potential alternative solutions that could have been used (see also section 7.3.5.4). The trade-off made between the solution chosen and the thought up solutions can then be constructed by comparing the solutions with each other. Generally this strategy works rather well. However, a problem arises when in hindsight a superior alternative solution is thought up. The trade-off constructed in these cases then no longer aligns with the choice made in the decision. To solve this misalignment problem, the choice element should describe two parts. First, the element should describe the choice made among the inferior solutions. This is needed for the reader to understand the original choice. Second, it should describe the reason why the superior solution was not an option on the table. This can be due to two different reasons:

- When the choice was made, the superior solution was excluded based on an invalid assumption.
- The superior solution was not considered at the time.

Both reasons are important architectural knowledge and should be documented in the choice element. In the case of exclusion on an invalid assumption, making these flawed assumptions explicit helps prevent the same mistake from being made again in the future. For both cases, the superior solution(s) form the perfect starting point to improve the design of the architecture later on. Furthermore, they are helpful in recovery, as it might well be that such decisions are undone and improved upon in later decisions.

7.6.3 Benefits

Although ADDRA is not without limitations, it still has it merits. In short, the approach has the following three benefits:

- **Systematic and disciplined approach.** ADDRA offers a detailed process that describes which steps are needed and in which order they should be executed.

This process has two important benefits: (1) It steers the recovery process with clear goals and means, which prevents a recovery attempt getting "lost". (2) The systematic nature induced by this process reduces the chance of overlooking important architectural design decisions.

- **Explicit use of tacit knowledge.** Many architectural recovery approaches do not explicitly use tacit knowledge in their recovery. They focus more on the available explicit knowledge, e.g. source code, models and existing documentation. ADDRA is different, as it describes *how* the tacit knowledge of an architect can be used in conjunction with formal and documented knowledge. This gives ADDRA the benefit of finding more architectural design decisions than other approaches that do not use tacit knowledge.
- **Recovers the rationale behind the architecture.** ADDRA not only recovers what the architecture is, but also the underlying architectural design decisions of the architecture. The benefit of this is that these decisions make the architecture description more understandable, as the reader is supplied the rationale of how the architecture came to be.

7.7 Related work

The recovery of architectural design decisions is related to three research areas: software architecture, design recovery, and rationale management. Following is a description of each area and how it relates to recovering architectural design decisions.

7.7.1 Software architecture

Software architectures ([11, 119, 138]) describe the the results of the main design decisions made for a system. As such, they can be seen as a collection of architectural design decisions ([21, 77]). Software architectures can be represented and described with the help of two types of approaches. One type is the documentation approach ([29, 67, 92]), the other is an Architectural Description Language (ADL) ([107, 115]). Both provide the architect with a vocabulary to reason about the architecture.

Documentation approaches ([29, 67, 92]) use the concept of different views, i.e. a description of a particular aspect of the software architecture, to describe the software architecture as a whole. These approaches originally primarily focused themselves on the *result* of the decisions, not on the architectural design decisions

themselves ([158, 163]). Extensions to these approaches tie the views closer to the architecting process and architectural decisions ([12, 68]).

As opposed to the documentation approaches, Architectural Description Languages (ADLs) ([107, 115]) focus on a single aspect, e.g. the principle computation entities and their relationships. An ADL has exact semantics and describes allowed combinations of architectural entities and their relationships. Similar to the documentation approaches, the notion of architectural design decisions is also unknown in ADLs ([76, 77]).

There is a growing interest in architectural design decisions within the software architecture community. [158] presents a template for documenting architectural design decisions, which is tailored towards a forward engineering effort, as opposed to the template presented in this paper for recovery. [93] and [95] have created a classification for design decisions and identify common relationships among them, which may be used to further classify the recovered decisions and create explicit relationships among them.

ADDRA focusses on the recovery of architectural design decisions. For forward engineering purposes, we have developed another approach called Archium ([77, 79]). The architectural decision model of Archium is very similar to the one presented in this paper (see figure 7.2). However, Archium extends this model by combining it with a requirements, architecture, and implementation model into one single unified model. This allows architects to document architectural design decisions with traces to related elements (e.g. requirements, or parts of the implementation).

Another useful purpose of the recovered architectural design decisions is for change impact analysis ([149]), which allows for predictions about the effort required for undoing certain decisions. ADDRA can also be used to extend the pattern mining approach of [6], where architectural design decisions are linked to well-known architectural styles ([138]) and patterns ([25]), as is done before for design patterns ([49]) in [9].

7.7.2 Design recovery

Design recovery usually focusses on recovering a design that is suitable for reengineering. Different techniques for recovery can be used, each yielding a different result.

Cluster analysis ([47, 91, 100]) is about finding groups in source code or other data artifacts by computing distances or similarities between elements, these groups are

subsequently identified as architectural components. For example, ACDC of [159] is a clustering algorithm for C, which primarily uses the number of references made to the combination of a source and header file.

Instead of using computed distances or similarities, concern graphs ([130]) use a human expert to abstract from structural program dependencies. With automated tool support, like the FEAT tool ([131]), these kind of abstractions can easily be made without losing traceability to the source code where these abstractions originate from.

Concept analysis ([43, 54, 144, 167]) is a mathematical approach to building taxonomies, which can be used to recover a partial description of the architecture with respect to a specific set of related features. For example, [144] use concept analysis to optimize inheritance relationships of classes, based on the usage of the methods and properties of these classes.

Design pattern recovery ([71, 86]) tries to recover the design patterns ([49]) that exist in a given design. The identified relationships between classes are compared to well-known specified design patterns, thereby identifying potential instances of design patterns.

Object identification ([28, 167, 176]) is the search for candidate classes in a legacy system. The identified classes in turn can be used for reconstructing the design of the system.

Another recovery approach is to look at the dynamic behavior of the software and analyze the run-time events (e.g. method invocations) of a running program. Discotect from [177] matches method invocations against user defined patterns, which represent architectural constructs to recover a representation of the run-time architecture. X-ray of [108] also recovers the run-time architecture, but, unlike Discotect, uses a static analysis of the source code.

One or more of these recovery approaches can be used to recover the architecture in the steps 2 and 3 of ADDRA. The Symphony process ([166]) describes in detail how a number of these techniques could be used to recover certain architectural and detailed design views. However, none of these approaches recover software architectures along with their rationale ([86]), which is the focus of ADDRA.

7.7.3 Rationale management

Knowledge systems ([128]) model decision processes and try to capture the knowledge used in these processes. From a knowledge system perspective, making architectural design decisions is seen as a decision process that decides how the architec-

ture should change. Capturing knowledge of this decision process provides a basis for the justification, learning, and reuse of this knowledge for further decisions.

Design decision models ([101, 122, 147]) refine this general decision process, as designing a design is much more a goal-oriented process than a general decision process ([101]). These models explicitly model the goal the design decision process wants to satisfy, as well as the design decisions and their rationale. Design decision models provide a basis to capture, describe, and reason about design decisions made in a design process. One implementation of a design decision model is SEURAT of [24], which enables software engineers to use a decision model within the Java IDE Eclipse.

The conceptual model of section 7.2.2 is inspired on the conceptual design decision model presented by [147], which in turn is based on an abstraction of IBIS ([32, 97]), DRCS ([101]), REMAP ([40, 126]), Redux ([120]), and OCS Shell Core ([4]). Although design decision models give some indication of what an architectural *design decision* is and is not, they fail to relate software architectures to architectural design decisions ([163]).

7.8 Future work & Conclusions

7.8.1 Conclusion

Software architecture documentation should not only describe the architecture of a system, but also *why* this architecture looks the way it does. The software architecture design decisions underlying the architecture provide this why. In practice, software architectures are often documented after the fact, i.e. when a system is realized and architectural design decisions have been taken. This paper presented ADDRA, an approach to recover architectural design decisions in an after the fact documentation effort.

To understand what architectural design decisions are, this paper presented a conceptual model. The conceptual model is used as input to the template used in ADDRA to document the recovered architectural design decisions. ADDRA recovers the changes made to the architecture in selected releases and relates these changes back to the architectural design decisions they originated from.

The ADDRA approach has been applied on the Athena case study to recover architectural design decisions. The case study identified three complicating factors for ADDRA: transitions between design decisions, subjectivity of architectural views, and the incompleteness of solutions. In addition, the required effort for recovery

is a complicating factor, although recovery tools can lift some of the burden. Nevertheless, these tools cannot replace the tacit knowledge of the architect, which ADDRA uses to recover the rationale of architectural design decisions.

ADDRA is the first approach on recovering architectural design decisions explicitly. Hopefully, with the rising interest in architectural decisions in the software architecture community, ADDRA can be matured and adequate tool created that eases the recovery of architectural design decisions in after the fact documentation efforts.

7.8.2 Further work & validation

One direction for further work is to compare and contrast forward, recovery, and hybrid approaches for capturing design decisions. This could provide a good overview of the current state of the art. In addition, issues that not have been addressed so far can be identified, thereby forming a research agenda for the future.

Another direction for further work is the validation of ADDRA. The application of ADDRA in one case study proves that architectural design decision recovery after the fact is to some extent possible. However, it is still unknown how ADDRA performs compared to an ad-hoc approach for recovering architectural design decisions. Furthermore, it remains unknown to what extent ADDRA is capable of recovering all the significant architectural design decisions. Further work on validation is therefore needed to find out the relative performance and recall rate of ADDRA.

We plan to perform such validation by conducting a controlled experiment. In this experiment, several teams consisting of 2 architects develop in parallel the same application. During development, dedicated scribes capture the architectural design decisions these teams make. After several releases, the architects of each team are asked to document their architectural decisions. Of each team, one architect will use ADDRA, the other one uses his/her own ad-hoc approach. The captured decisions during the development form a baseline to judge the recall rate of both approaches against.

Such an experiment is not without challenges. A first challenge is how to deal with the sensitivity of the experiment for the involved subjects. Only when enough subjects (and therefore teams) are involved in the same project can significant statistical confidence can be reached. A second challenge is that the experiment requires a minimum project size for architectural decisions to play a role. This in turn implies that a realistic duration is needed for development, which complicates replication of the experiment.

A third challenge is to find suitable test subjects. A choice has to be made whether to use student or industrial teams as test subjects. Both types of teams have their benefits and drawbacks. Industrial teams typically do not work on the same (part of a) system. This makes it harder to compare results between teams, as the system can become a discriminating factor in the experiment. To counter this effect, more industrial teams are needed to statistically overcome the influence of the system factor. Student teams on the other hand do not have this problem, as they could work on the same system. However, the results from student teams will be less convincing then those of industrial teams, because they are less experienced and are not exposed to the pressures found in industry. Concluding, in-depth validation of ADDRA is possible and is an interesting challenge for further work.

7.9 Acknowledgement

The authors would like to thank Tom Mens and the anonymous reviewers for their comments on earlier versions of this paper.

CHAPTER 8

CONCLUSIONS

This chapter presents the conclusions of this thesis. First, the answers are presented to the research questions underlying this thesis. This is followed by a description of the contributions this thesis makes to the software engineering field. The chapter concludes with an outlook into future research directions and open research questions.

8.1 Research Questions & Answers

In section 1.5 on page 16, the research questions of this thesis were presented. For each of these questions, we present a short answer here based on the work presented in the previous chapters. We start with the detailed research questions and conclude with answering the two more general research questions.

RQ-1: What are architectural design decisions?

We have provided a definition (see sections 3.4.3, 4.2, 6.2) and a conceptual model (see sections 3.5.1, 4.4.1, 7.2.2) to answer this question. There are two interpretations of this concept: a narrow and a broad one. In the narrow interpretation, an architectural decision is a *choice* among several architectural alternatives. This thesis, however, follows the broader interpretation in which architectural design decisions are much more than this choice alone. The conceptual model of section 7.2.2 describes this broader interpretation. Besides the choice it defines the following concepts to be part of an architectural design decision:

- **Problem,** a way to scope the problem space.
- **Motivation,** the rationale of this scoping.
- **Cause,** an analysis of the things that have lead to this scoping and motivation.
- **Solutions,** the alternatives considered for the choice along with their pros, cons, rules, and constraints.
- **Trade-offs,** the balances offered by the various solutions among quality attributes and functionality.
- **Architectural modification,** the effect the choice has on the software architecture.

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

Features in an SPL can be seen as a special type of architectural design decisions (see 1.3.3). Before answering the question how architectural design decisions can be modeled (RQ-2), we answer this question for this specific type of architectural design decisions. In chapter 2, we presented a role based feature model for modeling features in an SPL. The influence a feature might have on a component is isolated in a role. In this way, features can be described independently from each other. However, this separation of concerns comes at the price of making it hard to combine individual features. The complicating factors are feature interactions and the dependencies they represent, which conflict with the aim of describing features as separate concerns. How to deal with these feature interactions is the topic of RQ-3.1. Similar to this, a research question has been posed how to deal with architectural design decision dependencies (RQ-3).

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

The Archium meta-model (see A.1 on page 195) presented in chapters 4 and 5 models architectural (design) decisions. The meta-model models the elements described in the conceptual model for architectural design decisions (see RQ-1), which is based on the broad definition of this concept.

Similar to the way the role based feature model tries to make features first class citizens (see chapter 2), the Archium meta-model tries to achieve this for architectural design decisions. In particular, the modeling of the architectural modification, i.e. the changes to the architecture design of an architectural design decision, is inspired by the feature model. The concept of a delta comes from the concept of a role from the feature model. The delta concept allows Archium to separate the architectural modification from the architectural design. Consequently, this supports the modeling of architectural design decisions as first class citizens in Archium.

Archium adopts the broad definition of architectural design decisions (see the answer to RQ-1). This implies that a rich context is modeled in the Archium meta-model. To this end, the meta-model uses five (sub)models, which not only model architectural design decisions, but their context as well. The five (sub)models the Archium meta-model consists of are the following:

- **Requirements model,** for modeling the concerns that come from various stakeholders, which influence architectural decisions.
- **Decision model,** for modeling the architectural design decisions themselves.
- **Architecture model,** for modeling the effects an architectural design decision has on the architecture.
- **Implementation model,** for modeling how the architecture relates to the implementation. In the case of the Archium tool, the Java programming language is used.
- **Composition model,** is used for combining the decision, architecture, and implementation models.

RQ-3.1:How to deal with feature interactions?

One way to deal with feature interactions is to use a flexible composition approach with sufficient expressive power. This holds for situations in which features are modelled in isolation, such as done in the role based feature model presented in chapter 2 (see also RQ-2.1). In this situation, feature interactions and the related problems become apparent when multiple features are combined during composition.

To resolve these problems, a powerful and flexible composition approach is needed. To have enough expression power, the approach should support the three basic solutions for dealing with composition problems: skipping, mixing, and concatenation (see chapter 2). In addition, it should have the flexibility to combine one or more of these solutions at different levels of abstraction. The composition model of the role based feature model tries to achieve these two aspects. Although it supports the three basic composition solutions, it is rather limited in terms of flexibility. Nevertheless, it still provides a powerful and sensible approach to deal with feature interactions.

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

To deal with architectural design decision dependencies, we provide a solution similar to the separation of concerns approach used for feature interactions. The

Archium meta-model (see chapter 4 on page 79 and A.1 on page 195) models architectural design decisions using a similar separation-of-concerns modeling approach to the one used for the role based feature model (see chapter 2). Thus, Archium suffers from at least the same kind of composition problems as the feature model, since the dependencies among architectural design decisions might be more extensive than for features in an SPL. To what extent this is the case is an open research question.

The composition model of Archium uses superimposition as its composition technique, thereby offering support for the three basic composition solutions (see chapter 2). In comparison with the feature model, Archium offers a lot more flexibility than the inheritance based composition technique used in the role based feature model. The basic composition solutions can be freely combined through the use of superimposition [18]. In addition, Archium offers a much wider range of granularity for the composition, which ranges from a single method invocation to an entire software architecture design.

Although the composition model of Archium provides a way to deal with architectural design decisions dependencies, the approach does not model these dependencies explicitly. Instead, Archium infers the dependencies between architectural design decisions. This inference is based on found composition dependencies and references in the textual elements of an architectural design decision to other decisions. Complicating matters is the run-time binding of architectural design decisions in the Archium tool. This late binding time approach also delays the verification of these implicit dependencies. In many cases, it would be more convenient if these dependencies could be verified at design or compile time, as to warn the user of potential problems in advance. However, this would come at the price of reducing the flexibility of the composition of the approach. The best approach is most likely a hybrid approach, in which an architect can choose the binding time herself. In conclusion, Archium provides one way to deal with the architectural design decision dependencies, but there is room for improvement, i.e. the verification of dependencies in advance.

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

Explicit architectural decisions improve the architecting process in various ways. Among others, they enable architects to make better architectural choices. They also improve communication about the software architecture among the various stakeholders. In addition, they allow stakeholders to evaluate various software ar-

chitecture options better. This is due to the following benefits of the concept of explicit architectural decisions (see also sections 3.3.2, 4.3, and 7.2):

- **Focus** the architecting process by explicit scoping and selecting a subset of the problems.
- **Guarding of conceptual integrity** as rules, constraints, and intent of previous decisions are known.
- **Improved understandability** as it provides a rationale for the choices made.
- **Explicit design space exploration** as it charts the explored design space by describing the alternative solutions considered. This prevents reconsideration of earlier rejected solutions and eases the identification of unexplored areas.
- **Support of trade-off reconsideration** as explicit architectural design decisions allow for revisiting those decisions that, in retrospect, involve trade-offs that are problematic. A first benefit is that the effort to locate these design decisions is reduced significantly. Second, as these decisions describe the considered alternatives, the effort to think up suitable alternatives is reduced. Third, the impact of changing a decision can be predicted [152] and a rough estimate can be made which part of the system should be redesigned.
- **Decision analysis support** as explicit decisions allow one to easily perform what-if scenarios and thus identify which decisions are important sensitivity and trade-off points.
- **Tracing support** among architectural elements and requirements. For example, tracing the impact a design decision has on various views of the software architecture provides a mechanism to relate the elements of different views.

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

The decision process of making architectural decisions is the driving force behind the architecting process. Both the decision circle (see section 7.2 on page 145) and the similarities between the architecting process with the rationale management process (see section 3.4 on page 64) illustrate this. Architectural analysis, synthesis, and evaluation are all activities for the purpose of making architectural decisions. Hence, the concept of architectural decisions is omnipresent in the architecting process. Consequently, there is not a single place in the architecting process where architectural decisions are made; rather, they drive the process in the background.

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

In this thesis, we addressed this research question in the context of architectural evolution. For a knowledge sharing perspective on this issues we refer to [46]. Rationale management tools (e.g. Compendium) provide support for explicit decision making. However, they do not support the notion of a software architecture, nor are they suited for coping with changes induced by architectural evolution. ADLs and component languages do. on the other hand, support the description of software architectures to a certain degree, but lack the notion of architectural (design) decisions. In addition, both ADLs and component languages (apart from SOFA [121]) have sub-optimal support for changes induced by architectural evolution. Consequently, there exists a gap between these tools and decision management tools. The evaluation of the Archium tool (see 6.4.6) demonstrates that it can bridge this gap. Nevertheless, there is room for improvement in the areas of support for architectural views and support for the refinement process. Archium does, however, address the issue of changes induced by architectural evolution, something the other evaluated tools have trouble with.

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

We have identified 27 use cases involving AK, based on interviews with software architects and project managers from industry [162]. For the nine most important use cases, we explain how the Archium tool can support them (see chapter 5). Thus, we present a way in which partial tool support for architectural decisions could be provided.

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

The Architecture Design Decision Recovery Approach (ADDRA) (see chapter 7) outlines the steps and techniques one can use to recover architectural decisions. The approach uses a combination of architectural recovery and knowledge externalization techniques. The tacit knowledge of the original architect plays a crucial role in this kind of recovery, as the choices for abstraction of the architectural design decisions are otherwise unrecoverable. We present various ways in which this dependency on the original architect can be minimized.

RQ: How to reduce the vaporization of architectural decisions?

The answer to this question depends on the knowledge management strategy of an organization (see also section 1.3.1). In this thesis, the focus was on organizations using an explicit codification strategy. To eliminate architectural decision vaporization in this case, we need to create, manage, and maintain explicit architectural decisions throughout the life-cycle. Achieving this is far from trivial and also expensive, as architectural decisions are typically intertwined with each other and cross-cut the software architecture.

For forward engineering purposes, this thesis presents Archium, which addresses the two aforementioned issues of intertwining and cross-cutting. In practice, however, creating explicit architectural decisions is not always feasible. Thus, for reverse engineering purposes we presented ADDRA to recover these decisions. Together, Archium and ADDRA help the architect to create, manage, and maintain explicit architectural design decisions during the life-cycle.

How to reduce the vaporization of architectural knowledge?

This thesis presented a small step towards an answer to this research question. Our focus was primarily on one type of AK: architectural design decisions. In addition, we focussed exclusively on finding solutions in the context of a codification knowledge management strategy. To provide a general answer to this research question, we need to investigate other kinds of AK (e.g. process and people related) for both the codification and personalization strategies.

8.2 Contributions

The previous section already presented some of the contributions of this thesis. In this section, we present an overview of them. In short, this thesis made the following contributions:

- **Role-based SPF feature model** Chapter 2 presented a role-based approach for modeling features in a SPF. This model allows for automatic derivation of products based on feature selections. In general, feature models for product derivation suffer from the feature interaction problem. We analyzed this problem and concluded that, for automatic product derivation, feature interactions boil down to composition problems. Three basic solutions to these composition problems have been identified: concatenation, skipping, and mixing. The use of one or

more of these solutions allows for the automatic derivation of products based on features in an SPF. The insights gained from this model are used in the Archium meta-model and tool.

- **The bridging function of design decisions** We presented in chapter 3 how the concept of (architectural) design decisions forms a bridge between rationale and software architecture. To understand this bridging function, we analyzed the software architecture design process. This process assumes that the software architecture is designed in iterations. In each iteration, an architectural modification is made and incorporated into the software architecture. This is followed by an assessment to determine whether the quality of the software architecture is sufficient and thus to determine whether more iterations are needed. Design decisions come into this process through the architectural modifications, as they capture both the rationale behind these modifications and the considered alternatives. Consequently, design decisions form a bridge between rationale and software architecture.
- **A conceptual model for architectural design decisions** The conceptual model (see sections 3.5.1, 4.4.1, 7.2.2) of architectural design decisions describes what this concept entails. Furthermore, it describes the concepts of the context in which the decisions are made. The conceptual model can be used as a basis for templates that describe architectural design decisions. For example, in chapter 7, this is done for recovered architectural design decisions. The conceptual model is also used as a conceptual basis for making architectural design decisions explicit. The Archium meta-model uses the conceptual model towards this purpose.
- **Archium meta-model** The Archium meta-model, presented in chapters 4 and 5, models how architectural design decisions can be given a first-class representation (i.e. an identifiable nameable object) in architectural design. It incorporates the aforementioned conceptual model to model architectural (design) decisions. It also models the context of these decisions in more detail. The meta-model describes the relationships various concepts have with architectural (design) decisions and defines semantics for these relationships. For example, the relationships that architectural decisions have with requirements or components.

  A novel perspective incorporated in the meta-model is to model a software architecture design as a set of architectural (design) decisions. To achieve this, the meta-model uses, among others, two new concepts; deltas and design fragments. The delta concept originates from insights gained from the role based feature model. The concept of a delta models the influence an architectural design decision has on a component. It has been inspired by the feature role

concept. The second novel concept, is the design fragment. This concept represents a piece of architecture, thereby making it possible to model concepts like architectural design decisions that work on parts of the architecture.

- **Archium tool** The Archium tool described in chapters 4 and 5 is one way to implement the Archium meta-model. The tool enables architects to create, manage, and use architectural (design) decisions. As its implementation model, the tool uses the Java language. This offers the architect a way to maintain a software architecture and its implementation, including the architectural design decisions. This can be done not only during the design phase, but also during the remaining part of the system life-cycle, as the decisions become an inherent and explicit part of the system.
- **New views and notations** To communicate the different concepts of the Archium meta-model, we have developed various views and notations. For example, figure 4.4 on page 91 graphically displays design fragment composition. Another example is figure 4.6 on page 95, which presents an evolutionary view on architectural design decisions. The Archium tool also offers a design decision impact view by combining a design decision dependency view and a component & connector view (see figure 5.1 on page 109).
- **Evaluation of tool support for architectural evolution** Chapter 6 presents a framework for the evaluation of tool support for architectural evolution. It allows one to judge tools with respect to their ability to support software architecture evolution. The framework consists of coarse grained criteria concerning the architecture, architectural design decisions, and architectural changes for evaluation. We have applied the evaluation framework on six different tools. From the results, we conclude that, in general, there exists a gap between software architecture tools on one side and knowledge management tools on the other side. The Archium tool bridges this gap, but can be improved in the area of support for multiple architectural views and facilities for expressing incompleteness by under-specification.
- **ADDRA** The Architectural Design Decision Recovery Approach (ADDRA) approach presented in chapter 7 offers the architect a means to recover architectural design decisions. The approach supports the recovery of decisions months or even years after they have been made. To this end, It uses a combination of architectural recovery techniques, differences in recovered architectures, and knowledge externalization strategies.

8.3 Open research questions and future work

Not all the research questions of this thesis were completely answered in section 8.1. In this section, we present these research questions that are still partially open together with possible directions for future work.

This thesis focussed on organizations using a codification knowledge management strategy for minimizing architectural knowledge vaporization. However, many organizations do not use a codification strategy, but use a personalization strategy. An open research questions is therefore how to minimize architectural knowledge vaporization in organizations using a personalization knowledge management strategy.

The idea of viewing design patterns as collections of reuseable design decisions is another direction for future work. For example, Harrison et al [64] explored how patterns can be used to capture architectural design decisions. Another interesting direction is to investigate the kind of design decisions made to integrate a pattern in a software architecture. This might provide the basis for offering better (tool) support for these kind of re-usable design decisions.

An other interesting open question is how architectural design decisions can be undone. Undoing architectural design decisions is a special case of modifying the architecture with the aim to remove constraints, rules, design modifications, or unwanted dependencies imposed by a decision. Of interest is undoing these decisions not only during design, but also during later phases of the life cycle. For example, in Archium, one could investigate how architectural design decisions could be reverted at run-time, including the implications this brings for the approach.

Another direction for future work is to look into the relationship an Archium like tool might have with software architecture documentation. If at all, formalization of the architecture, as supported by Archium, is usually done after a software architecture is textually described. Investigating how such a software architecture document could be refined and later transformed into a formal model might provide some insights into the problems with formalizing an architectural description. This knowledge in turn could be used to optimize the translation from an informal architecture description to a formal one.

The research question about how to deal with architectural design decision dependencies (RQ-3) was only partially answered. The answer from the role based feature model (see chapter 2) is incomplete and focusses on the role these dependencies play in a composition. However, the role the dependencies that are represented by textual references in Archium actually play are not dealt with. One direction for

future work is therefore to investigate the semantics of these textual relationships and the concepts they represent.

So far Archium has only been applied to small case studies. To further validate the Archium approach, bigger case studies are necessary. This comes not without challenges, as evolution forms an inherent part of the approach. Thus a part of the evolution of such a large case should be tracked, which complicates things and requires considerable effort.

The Archium tool is in many aspects a research prototype and needs to be matured. Especially, some of the technical limitations the underlying ArchJava language imposes (e.g. no support for run-time decoupling of connectors) should be resolved. Hence, maturing the Archium tool is another challenge. Another direction for further work is to investigate how the Archium meta-model could be expanded to include more architectural views (e.g. a module or deployment view) besides the component & connector view.

In chapter 6, various tools were evaluated with respect to the support they could offer for the evolution of software architectures and architectural (design) decisions. However, tools for managing AK should be evaluated from other perspectives as well, e.g. for sharing [46], creating, and evaluating AK. In addition, several new tools for managing AK have been published, e.g. AREL [151], PAKME [6], ADDSS [26], after the evaluation. This is another direction for future work, which at this time of writing is being pursued actively by the authors of these tools.

For the recovery of architectural design decisions, the ADDRA approach has been validated using a case study. This proved that it is a possible to recover some architectural design decisions afterwards. However, to what extent this is possible is still an open research question (see also 7.8.2). Another open question is whether such a recovery makes economical sense, i.e. the benefits outweigh the cost of recovery. Thus these two questions provide another direction for further work.

APPENDIX

A.1 Archium meta-model

The following figure on the next page gives an overview of the Archium meta-model, as described in chapters 4 and 5. Note, that the description of the internal model elements of the design decision is missing in this figure. The conceptual model presented in figure 4.1 on page 85 presents these elements.

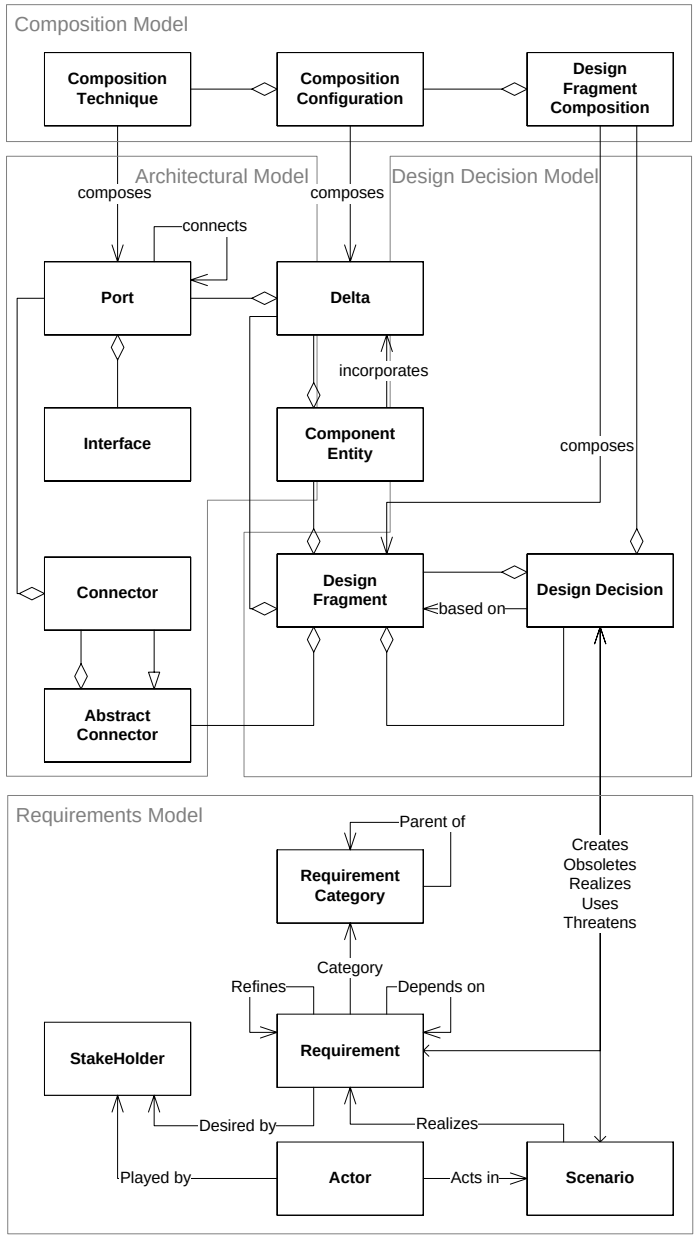

Figure 1: Overview of the complete Archium meta-model

REFERENCES

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[2] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the 24th international conference on Software engineering*, pages 187–197. ACM Press, 2002.

[3] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *ECOOP 2003 – Object-Oriented Programming: 17th European Conference*, volume 2743 of *Lecture Notes in Computer Science*, pages 74–102. Springer-Verlag, July 2003.

[4] G. Arango, L. Bruneau, J. F. Cloarec, and A. Feroldi. A tool shell for tracking design decisions. *IEEE Software*, 8(2):75–83, March 1991.

[5] The Archium website, http://www.archium.net.

[6] M. Babar, I. Gorton, and B. Kitchenham. A framework for supporting architecture knowledge and rationale management. In A. H. Dutoit, R. McCall, I. Mistrík, and B. Paech, editors, *Rationale Management in Software Engineering*, chapter 11, pages 237–254. Springer-Verlag, March 2006.

[7] M. A. Babar, R. C. de Boer, T. Dingsøyr, and R. Farenhorstir. Architectural knowledge management strategies:approaches in research and industry. In *Proceedings of the 2nd Workshop on SHAring and Reusing architectural Knowledge - Architecture, rationale, and Design Intent (SHARK/ADI 2007)*, May 2007.

[8] M. Bachler, S. Buckingham Shum, D. D. Roure, D. Michaelides, and K. Page. Ontological mediation of meeting structure: Argumentation, annotation, and navigation. In *1st International Workshop on Hypermedia and the Semantic Web*, 2003.

[9] E. L. A. Baniassad, G. C. Murphy, and C. Schwanninger. Design pattern rationale graphs: Linking design to source. In *Proceedings of the 25th ICSE*, pages 352–362, May 2003.

[10] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison Wesley, 1998.

[11] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice 2nd ed.* Addison Wesley, 2003.

[12] L. Bass, P. Clements, R. L. Nord, and J. Stafford. Capturing and using rationale for a software architecture. In A. H. Dutoit, R. McCall, I. Mistrik, and B. Paech,

editors, *Rationale Management in Software Engineering*, chapter 12, pages 255–272. Springer-Verlag, March 2006.

[13] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multi-dimensional separation of concerns. In *Proceedings of the 9th European software engineering conference*, pages 48–57. ACM Press, 2003.

[14] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 73–87. ACM Press, 2000.

[15] B. W. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, January 2000.

[16] C. Boekhoudt. The big bang theory of ides. *Queue*, 1(7):74–82, 2003.

[17] G. Booch, J. RumBaugh, and I. Jacobson. *The unified modeling language user guide*. Addison Wesley, 1998.

[18] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41(5):257–273, 25 March 1999.

[19] J. Bosch. *Design & Use of Software Architectures, Adopting and evolving a product-line approach*. ACM Press/Addison Wesley, 2000.

[20] J. Bosch. Maturity and evolution in software product lines: approaches, artefacts and organization. In *Proceedings of the 2nd Software Product Line Conference (SPLC 2002)*, August 2002.

[21] J. Bosch. Software architecture: The next step. In *Software Architecture, First European Workshop (EWSA)*, volume 3047 of *LNCS*, pages 194–199. Springer, May 2004.

[22] L. Bratthall, E. Johansson, and B. Regnell. Is a design rationale vital when predicting change impact? a controlled experiment on software architecture evolution. In *Second International Conference on Product Focused Software Process Improvement (PROFES)*, volume 1840 of *LNCS*, pages 126–139. Springer, 2000.

[23] M. Broy. Automotive software and systems engineering. *memocode*, 0:143–149, 2005.

[24] J. E. Burge and D. C. Brown. An integrated approach for software design checking using design rationale. In *1st International Conference on Design Computing and Cognition (DCC '04)*, pages 557–576, July 2004.

[25] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns*. John Wiley & Sons, Inc., 1996.

[26] R. Capilla, F. Nava, S. Pérez, and J. C. Dueñas. A web-based tool for managing architectural design decisions. *SIGSOFT Software Engineering Notes*, 31(5), 2006.

[27] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and Wui-Gee. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[28] A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino. Identifying objects in legacy systems using design metrics. *Journal of Systems and Software*, 44(3):199–211, 1999.

[29] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures, Views and Beyond*. Addison Wesley, 2002.

[30] E. J. Conklin and K. B. Yakemovic. A process-oriented approach to design rationale. *Human-Computer Interaction*, 6(3/4), 1991.

[31] J. Conklin and M. L. Begeman. gibis: a hypertext tool for exploratory policy discussion. *ACM Transactions on Information Systems (TOIS)*, 6(4):303–331, 1988.

[32] J. Conklin and M. L. Begeman. gibis: a tool for all reasons. *Journal of the American Society for Information Science*, 40(3):200–213, 1989.

[33] M. E. Conway. How do committees invent? *Datamation*, 14(4):28–31, April 1968.

[34] J. O. Coplien and N. B. Harrison. *Organizational Patterns of Agile Software Development*. Pearson Prentice Hall, 1995.

[35] K. Czarnecki. Overview of generative software development. In *Proceeding of the Unconventional Programming Paradigms, International Workshop (UPP 2004)*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341. Springer, September 2004.

[36] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[37] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *Proceedings of the 24th international conference on Software engineering*, pages 266–276. ACM Press, 2002.

[38] R. C. de Boer and R. Farenhorst. In search of 'architectural knowledge'. In *SHARK '08: Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, pages 71–78, New York, NY, USA, 2008. ACM.

[39] R. C. de Boer, R. Farenhorst, P. Lago, H. van Vliet, and A. G. J. Jansen. Architectural knowledge: Getting to the core. In *Proceedings of the Third International Conference on the Quality of Software Architectures (QoSA 2007)*, volume 4880 of *LNCS*, pages 197–214, July 2007.

[40] V. Dhar and M. Jarke. Dependency directed reasoning and learning in systems maintenance support. *IEEE Transactions on Software Engineering*, 14(2):211–227, 1988.

[41] F. P. Dusan Bàlek. Software connectors and their role in component deployment. In K. Zielinski, K. Geihs, and A. Laurentowski, editors, *Third International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, volume 198 of *IFIP Conference Proceedings*. Kluwer, 2001.

[42] A. H. Dutoit, R. McCall, I. Mistrik, and B. Paech, editors. *Rationale Management in Software Engineering*. Springer-Verlag, March 2006.

[43] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM'01)*, pages 602–611. IEEE Computer Society, November 2001.

[44] D. Falessi, G. Cantone, and M. Becker. Documenting design decision rationale to improve individual and team design decision making: an experimental evaluation. In

Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering (ISESE '06), pages 134–143, New York, NY, USA, 2006. ACM Press.

[45] R. Farenhorst, R. C. de Boer, R. Deckers, P. Lago, and H. van Vliet. What's in constructing a domain model for architectural knowledge? In *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE2006)*, July 2006.

[46] R. Farenhorst, P. Lago, and H. van Vliet. Effective tool support for architectural knowledge sharing. In *Proceedings of the First European Conference on Software Architecture (ECSA 2007)*, volume 4758 of *LNCS*, pages 123–138, September 2007.

[47] L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. *Software - Practice and Experience*, 28(4):371–400, April 1998.

[48] M. Fowler. Dealing with roles. In *Proceedings of the 4th Annual Conference on the Pattern Languages of Programs (PLoP)*, September 2-5 1997.

[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[50] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[51] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., 1991.

[52] P. Gibson. Feature requirements models: Understanding interactions. In L. L. P. Dini, R. Boutaba, editor, *Feature Interactions in Telecommunications Networks IV*, pages 46–60. IOS Press, June 1997.

[53] R. L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information & Software Technology*, 44(8):491–506, 2002.

[54] R. Godin, G. Mineau, R. Missaoui, M. St-Germain, and N. Faraj. Applying concept formation to software reuse. *International Journal of Software Engineering and Knowledge Engineering*, 5(1):119–142, 1995.

[55] D. G. Gregg, U. R. Kulkarni, and A. S. Vinzé. Understanding the philosophical underpinnings of software engineering research in information systems. *Information Systems Frontiers*, 3(2):169–183, 2001.

[56] Griffin project website, http://griffin.cs.vu.nl.

[57] M. L. Griss. Implementing product-line features by composing aspects. In *Proceedings of the first conference on Software product lines : experience and research directions*, pages 271–288, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[58] M. L. Griss. Implementing product-line features with component reuse. In W. B. Frakes, editor, *Software Reuse: Advances in Software Reusability, 6th International Conference, ICSR-6*, volume 1844 of *Lecture Notes in Computer Science*. Springer, June 2000.

[59] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the rseb. In *ICSR '98: Proceedings of the 5th International Conference on Software*

Reuse, page 76, Washington, DC, USA, 1998. IEEE Computer Society.

[60] Y.-G. Gueheneuc. A systematic study of uml class diagram constituents for their abstract and precise recovery. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 265–274. IEEE Computer Society, 2004.

[61] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[62] I. Habli and T. Kelly. Capturing and replaying architectural knowledge through derivational analogy. In *SHARK-ADI '07: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, page 4, Washington, DC, USA, 2007. IEEE Computer Society.

[63] M. T. Hansen, N. Nohria, and T. Tierney. What's your strategy for managing knowledge? *Havard Business Review*, 77(2):106–116, March-April 1999.

[64] N. B. Harrison, P. Avgeriou, and U. Zdun. Architecture patterns as mechanisms for capturing architectural decisions. *IEEE Software*, 24(4):38–45, 2007.

[65] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, 1993.

[66] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America. Generalizing a model of software architecture design from five industrial approaches. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005)*, pages 77–88. IEEE Computer Society, 2005.

[67] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture*. Addison Wesley, 2000.

[68] C. Hofmeister, R. L. Nord, and D. Soni. Global analysis: moving from software requirements specification to structural views of the software architecture. *IEE Proceedings Software*, (4):187–197, August 2005.

[69] H. J. Holz, A. Applin, B. Haberman, D. Joyce, H. Purchase, and C. Reed. Research methods in computing: what are they, and how should we teach them? In *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 96–114, New York, NY, USA, 2006. ACM Press.

[70] IEEE/ANSI. *Recommended Practice for Architectural Description of Software-Intensive Systems*, 2000. IEEE Standard No. 1471-2000, Product No. SH94869-TBR.

[71] V. Jakobac, A. Egyed, and N. Medvidovic. Improving system understanding via interactive, tailorable, source code analysis. In M. Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2005.

[72] C. B. Jaktman, J. Leaney, and M. Liu. Structural analysis of the software architecture - a maintenance assessment case study. In P. Donohoe, editor, *Software Architecture (WICSA1)*, volume 140 of *IFIP Conference Proceedings*, pages 455–470. Kluwer, Februari 1999.

[73] A. Jansen. Feature based composition. Master's thesis, University of Groningen, September 2002.

[74] A. Jansen, R. Smedinga, J. van Gurp, and J. Bosch. First class feature abstractions for product derivation. *IEE Proceedings Software*, 151(4):187–197, August 2004.

[75] A. G. J. Jansen. Athena, a large scale programming lab support tool. In *Proceedings of the Dutch National Computer Science Education Congress (NIOC)*, pages 83–89. Uitgeverij Passage, 2004.

[76] A. G. J. Jansen and J. Bosch. Evaluation of tool support for architectural evolution. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 375–378. IEEE, September 2004.

[77] A. G. J. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005)*, pages 109–119, November 2005.

[78] A. G. J. Jansen, J. Bosch, and P. Avergiou. Documenting after the fact: recovering architectural design decisions. *Journal of Systems and Software*, 81(4):536–557, April 2008.

[79] A. G. J. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *Proceedings of the 6th IEEE/IFIP Working Conference on Software Architecture (WICSA 2007)*, page 4, Januari 2007.

[80] A. G. J. Jansen, J. van Gurp, and J. Bosch. Reconstructing architectural design decisions: A case study. Technical Report IWI preprint 2003-7-02, Department of Mathematics and Computing Science, University of Groningen, PO Box 800, 9700 AV The Netherlands, December 2003.

[81] JavaCC website, http://javacc.dev.java.net/.

[82] JGraph website, http://www.jgraph.org.

[83] Chris johnson's website on research in computing science. http://www.dcs.gla.ac.uk/~johnson/teaching/research_skills/research.html.

[84] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, Carnegie Mellon University, 1990.

[85] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[86] R. K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 226–235. IEEE Computer Society, May 1999.

[87] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[88] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[89] M. Klein. Capturing design rationale in concurrent engineering teams. *IEEE Computer*, 26(1):39–47, 1993.

[90] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture : Practice and Promise*. Addison-Wesley, 2003.

[91] R. L. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A two-phase process for software architecture improvement. In *International Conference on Software Maintenance (ICSM99)*, pages 371–380, september 1999.

[92] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.

[93] P. Kruchten. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen Workshop on Software Variability*, pages 54–61, December 2004.

[94] P. Kruchten. Casting software design in the function-behavior-structure framework. *IEEE Softw.*, 22(2):52–58, 2005.

[95] P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In *Proceedings of the Second International Conference on the Quality of Software Architectures (QoSA 2006)*, June 2006.

[96] P. Kruchten, P. Lago, H. van Vliet, and T. Wolf. Building up and exploiting architectural knowledge. In *WICSA 5*, November 2005.

[97] W. Kunz and H. W. J. Rittel. Issues as elements of information systems. Technical Report Working paper 131, Institut fur Grundlagen der Planung, Universitat Stuttgart, July 1970.

[98] P. Lago and P. Avgeriou. First workshop on sharing and reusing architectural knowledge. *SIGSOFT Software Engineering Notes*, 31(5):32–36, 2006.

[99] P. Lago and H. van Vliet. Explicit assumptions enrich architectural models. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 206–214, New York, NY, USA, 2005. ACM Press.

[100] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, 36(3):211–231, 1997.

[101] J. Lee. Extending the potts and bruns model for recording design rationale. In *Proceedings of the 13th International Conference on Software Engineering (ICSE 1991)*, pages 114–125. IEEE, 1991.

[102] A. MacLean, R. M. Young, V. M. Bellotti, and T. P. Moran. Questions, options, and criteria: Elements of design space analysis. *Human-Computer Interaction*, 6(3&4):201–250, 1991.

[103] G. Malpohl. Jplag website. http://www.jplag.de/.

[104] E. Marcos. Software engineering research versus software development. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[105] C. McNamara. *Field Guide to Consulting and Organizational Development: A Collaborative and Systems Approach to Performance, Change and Learning*. Authenticity Consulting, LLC, 2006.

[106] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Enginerring (ICSE 1999)*, pages 44–53. IEEE Computer Society Press, 1999.

[107] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[108] N. C. Mendonça and J. Kramer. Developing an approach for the recovery of distributed software architectures. In *6th IEEE International Workshop on Program Comprehension*, pages 28–36, Ischia, Italy, June 1998. IEEE. The paper describes the initial work on the X-RAY architecture recovery approach and tools.

[109] T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE 2003)*, April 2003.

[110] I. G. Muhammad Ali Babar and R. Jeffery. Toward a framework for capturing and using architecture design knowledge. Technical Report UNSW-CSE-TR-0513, University of New South Wales, Australia ans National ICT Australia Ltd., june 2005.

[111] I. Nonaka. A dynamic theory of organizational knowledge creation. *Organization Science*, 5(1):14–37, February 1994.

[112] I. Nonaka and H. Takeuchi. *The Knowledge-creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press Inc, USA, 1995.

[113] D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236. ACM Press, 2003.

[114] OMG. Common object request broker architecture (corba/iiop). version 3.0.3. Technical Report formal/2004-03-12, Object Management Group, 2004.

[115] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 177–186. IEEE, 1998.

[116] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[117] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[118] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.

[119] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[120] C. Petrie. Constrained decision revision. In *Proceedings of the Tenth AAAI Conference*, pages 393–400, 1992.

[121] F. Plàsil, D. Bàlek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *Proceedings of the International Conference on Configurable Distributed Systems*, page 43. IEEE Computer Society, 1998.

[122] C. Potts and G. Bruns. Recording the reasons for design decisions. In *Proceedings of the 10th International Conference on Software Engineering (ICSE 1988)*, pages 418–427. IEEE, 1988.

[123] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

[124] C. Prehofer. An object-oriented approach to feature interaction. In *Feature Interactions in Telecommunications Networks IV*, pages 313–325. IOS Press, June 1997.

[125] T. Qin, L. Zhang, Z. Zhou, D. Hao, and J. Sun. Discovering use cases from source code using the branch-reserving call graph. In *Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference (APSEC)*, page 60, Washington, DC, USA, 2003. IEEE Computer Society.

[126] B. Ramesh and V. Dhar. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, June 1992.

[127] A. Rashid, P. Sawyer, A. M. D. Moreira, and J. Araújo. Early aspects: A model for aspect-oriented requirements engineering. In *Proceedings of the 10th IEEE Joint International Conference on Requirements Engineering (RE 2002)*, pages 199–202. IEEE Computer Society, September 2002.

[128] W. Regli, X. Hu, M. Atwood, and W. Sun. A survey of design rationale systems: Approaches, representation, capture and retrieval. *Engineering with Computers*, 16(3-4):209–235, December 2000.

[129] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 117–133, New York, NY, USA, 1998. ACM Press.

[130] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, New York, NY, USA, 2002. ACM Press.

[131] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3, 2007.

[132] R. Roshandel, A. V. D. Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.*, 13(2):240–276, 2004.

[133] S. Sarkar and S. Thonse. Eaml- architecture modeling language for enterprise applications. In *CEC-EAST '04: Proceedings of the E-Commerce Technology for Dynamic E-Business, IEEE International Conference on (CEC-East'04)*, pages 40–47, Washington, DC, USA, 2004. IEEE Computer Society.

[134] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, New York, NY, USA, 2003. ACM Press.

[135] A. Selvin. Leveraging existing hypertext functionality to create a customized environment for team analysis. In *Proceedings of the Second International Workshop on Incorporating Hypertext Functionality Into Software Systems*, March 1996.

[136] M. Shaw. What makes good research in software engineering? *International Journal*

on Software Tools for Technology Transfer (STTT), 4(1):1–7, October 2002.

[137] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.

[138] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.

[139] http://www.sigcse.org/. The ACM Special Interest Group on Computer Science Education (SIGCSE) website.

[140] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *Third International Conference on Software Product Lines (SPLC)*, volume 3154 of *LNCS*, pages 197–213, 2004.

[141] M. Sinnema, J. S. van der Ven, and S. Deelstra. Using variability modeling principles to capture architectural knowledge. In *Proceedings of the Workshop on SHAring and Reusing architectural Knowledge (SHARK 2006)*, June 2006.

[142] Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECCOP)*, pages 550–570, London, UK, 1998. Springer-Verlag.

[143] G. F. Smith and G. J. Browne. Conceptual foundations of design problem solving. *IEEE Transactions on Systems, Man and Cybernetics*, 23(5):1209–1219, September 1993.

[144] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–110. ACM Press, 1998.

[145] Software engineering institute software architecture definition page. http://www.sei.cmu.edu/architecture/definitions.html.

[146] I. Sommerville. *Software Engineering*. Addision-Wesley, 8 edition, 2007.

[147] Z. R. Stepenson. *Change Management in Families of Safety- Critical Embedded Systems*. PhD thesis, University of York, 2002.

[148] M. R. T. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, October 1998.

[149] A. Tang, M. A. Babar, I. Gorton, and J. Han. A survey of the use and documentation of architecture design rationale. In *Proceeding of the Fifth Working IEEE / IFIP Conference on Software Architecture (WICSA 2005)*, pages 89–99, November 2005.

[150] A. Tang, M. A. Babar, I. Gorton, and J. Han. A survey of architecture design rationale. *Journal of Systems & Software*, 79(12):1792–1804, 2006.

[151] A. Tang, Y. Jin, and J. Han. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software*, 80(6):918–934, June 2007.

[152] A. Tang, Y. Jin, J. Han, and A. E. Nicholson. Predicting change impact in architecture design with bayesian belief networks. In *Proceeding of the Fifth Working IEEE / IFIP Conference on Software Architecture (WICSA 2005)*, pages 67–76. IEEE Computer Society, November 2005.

[153] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation:

multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. IEEE, 1999.

[154] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Trans. Softw. Eng.*, 22(6):390–406, 1996.

[155] P. Tonella and A. Potrich. Static and dynamic c++ code analysis for the recovery of the object diagram. In *Proceedings of the International Conference on Software Maintenance*, pages 54–63, October 2002.

[156] P. W. Trygve Reenskaug and O. A. Lehne. *Working with Objects The OOram Software Engineering Method*. Manning Publications, 1995.

[157] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *Journal of Systems & Software*, 49(1):3–15, 1999.

[158] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecure. *IEEE Software*, 22(2):19–27, 2005.

[159] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Working Conference on Reverse Engineering (WCRE 2000)*, pages 258–267. IEEE Computer Society, 2000.

[160] http://www.uml.org/, The Unified Modeling Language (UML) website.

[161] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic. Taming architectural evolution. In *Proceedings of the 8th European software engineering conference*, pages 1–10. ACM Press, 2001.

[162] J. S. van der Ven, A. G. J. Jansen, P. Avgeriou, and D. K. Hammer. Using architectural decisions. In *Second International Conference on the Quality of Software Architecture (Qosa 2006)*, 2006.

[163] J. S. van der Ven, A. G. J. Jansen, J. A. G. Nijhuis, and J. Bosch. Design decisions: The bridge between rationale and architecture. In A. H. Dutoit, R. McCall, I. Mistrik, and B. Paech, editors, *Rationale Management in Software Engineering*, chapter 16, pages 329–348. Springer-Verlag, March 2006.

[164] A. van Deursen. Software architecture recovery and modelling: [wcre 2001 discussion forum report]. *ACM SIGAPP Applied Computing Review*, 10(1):4–7, 2002.

[165] A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In *Proceedings of the Second International Conference on Software Product Lines (SPLC)*, pages 217–234, London, UK, 2002. Springer-Verlag.

[166] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings of the 4th IEEE/IFIP Working Conference on Software Architecture (WICSA 2004)*, page 122. IEEE Computer Society, 2004.

[167] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.

[168] J. van Gurp and J. Bosch. Design erosion: Problems & causes. *Journal of Systems & Software*, 61(2):105–119, March 2002.

[169] R. van Ommering. *Building Product Populations with Software Components*. PhD thesis, University of Groningen, 2004.

[170] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[171] Velocity website, http://jakarta.apache.org/velocity.

[172] M. Visconti and C. R. Cook. Assessing the state of software documentation practices. In F. Bomarius and H. Iida, editors, *PROFES*, volume 3009 of *Lecture Notes in Computer Science*, pages 485–496. Springer, 2004.

[173] Z. Wang, K. Sherdil, and N. H. Madhavji. ACCA: An architecture-centric concern analysis method. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 99–108, November 2005.

[174] D. W. Weber. Change sets versus change packages: Comparing implementations of change-based scm. In *Proceedings of the SCM-7 Workshop on System Configuration Management*, pages 25–35. Springer-Verlag, 1997.

[175] http://www.webster.com, 2006.

[176] T. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In *Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 24–32. IEEE Computer Society, October 1997.

[177] H. Yan, D. Garlan, B. R. Schmerl, J. Aldrich, and R. Kazman. Discotect: A system for discovering architectures from running systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 470–479. IEEE Computer Society, 2004.

[178] P. Zave. Feature-oriented description, formal methods, and dfc. In *Proceedings of the FIRE-works Workshop on Language Constructs for Describing Features*, pages 11–26, May 2000.

[179] M. V. Zelkowitz and D. R. Wallace. Experimental validation in software engineering. *Information & Software Technology*, 39(11):735–743, 1997.

[180] H. Zhuge. *The Knowledge Grid*. World Scientific Publishing Company, 2004.

[181] B. Zimmermann and A. M. Selvin. A framework for assessing group memory approaches for software design projects. In *Proceedings of the conference on Designing interactive systems*, pages 417–426. ACM Press, 1997.

PUBLICATIONS

- **First class feature abstractions for product derivation** Anton Jansen, Rein Smedinga, Jilles van Gurp, and Jan Bosch. Special issue on Early Aspects: Aspect-oriented Requirements Engineering and Architecture Design. IEE Proceedings Software 151(4), pp. 187-197, August 2004.
- **Design Decisions: The Bridge between Rationale and Architecture** Jan S. van der Ven, Anton Jansen, Jos A. G. Nijhuis, Jan Bosch. Chapter 16, Rationale Management in Software Engineering, Allen H. Dutoit, Raymond McCall, Ivan Mistrik, Barbara Paech Editors, pp. 329-346, Springer-Verlag, April 2006.
- **Software Architecture as a Set of Architectural Design Decisions** Anton Jansen, Jan Bosch. Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005), pp. 109-119, November 2005.
- **Tool support for Architectural Decisions** Anton Jansen, Jan van der Ven, Paris Avgeriou, Dieter K. Hammer. Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), Januari 2007.
- **Evaluation of Tool Support for Architectural Evolution** Anton Jansen, Jan Bosch. Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), pp. 375-378, September 2004.
- **Documenting after the fact: recovering architectural design decisions** Anton Jansen, Jan Bosch, and Paris Avegriou, Documenting after the fact: recovering architectural design decisions. Journal of Systems and Software (JSS) 81(4), pp. 536-557, Elsevier Science, April, 2008.
- **Athena, a large scale programming lab support tool** Anton Jansen. Proceedings of the Dutch National Computer Science Education Congress, 2004.
- **Using Architectural Decisions** Jan S. van der Ven, Anton Jansen, Paris Avgeriou, Dieter K. Hammer. Second International Conference on the Quality of Software Architecture (Qosa 2006).

SUMMARY

A software architecture can be considered as the collection of key decisions concerning the design of the software of a system. Knowledge about this design, i.e. architectural knowledge, is key for understanding a software architecture and thus the software itself. Architectural knowledge is mostly tacit; it only exists in the heads of the creators. A problem is that this type of knowledge is easily lost. This phenomenon is called architectural knowledge vaporization and contributes to a number of problems that the industry is struggling with: expensive system evolution, difficult stakeholder communication, and limited reusability.

The central theme of this thesis is how to reduce this vaporization of architectural knowledge. The focus is on one important form of architectural knowledge: architectural design decisions. This form is important, as the architecting process is all about making these key decisions. To reduce vaporization, this thesis explores a codification solution, in which these decisions are documented and modeled.

For codification, the concepts one wants to codify must be understood well. To this end, a conceptual model of architectural design decisions is presented that explains the parts that this concept is made of and how it relates to other concepts. Based on this model, two approaches have been developed. The first, the Archium approach, is used for codifying, managing, and maintaining architectural design decisions in a forward engineering setting. The second approach is the Architectural Design Decision Recovery Approach (ADDRA) for recovering architectural design decisions in a reverse engineering setting.

The Archium approach is evaluated in two ways. First, a study of how Archium deals with common use-cases for managing architectural knowledge is presented. Second, to address the issue of expensive system evolution, the Archium tool is compared with other tools using an evaluation framework. To evaluate ADDRA, the approach is applied on a case study.

Both ADDRA and Archium help the architect with codifying architectural design decisions. This codification has two important consequences for software architecting. First, design decisions become explicit bridges between design and rationale.

Second, once codified, a software architecture can be seen as set of architectural (design) decisions. Both consequences put a software architecture into a new perspective and deepen our understanding about what software architecting is all about and they also help to reduce architectural knowledge vaporization. As future work, we plan to investigate other types of architectural knowledge and their relationship to architectural design decisions.

SAMENVATTING

Een software architectuur kan beschouwd worden als een verzameling van kern beslissingen m.b.t. het ontwerp van de software van een system. Kennis over dit ontwerp, oftewel architectuurkennis, is cruciaal voor het begrijpen van een software architectuur en daarmee ook de software zelf. Architectuurkennis is meestal impliciet; het bestaat alleen in de hoofden van de mensen die het hebben gemaakt. Een probleem is dat deze impliciete vorm van kennis gemakkelijk verloren gaat. Dit proces wordt ook wel kennis verdamping genoemd en draagt bij tot enkele problemen waarvan de industrie last heeft: dure evolutie van systemen, moeilijke communicatie tussen belanghebbenden en beperkte mogelijkheden tot hergebruik.

Het centrale thema van dit proefschrift is het reduceren van deze architectuurkennis verdamping. De focus ligt op één belangrijke vorm van architectuurkennis: architectuur ontwerp beslissingen. Deze vorm is belangrijk, daar het architectuur proces voor een groot deel draait om het maken van deze cruciale beslissingen. Om de verdamping te reduceren wordt een zgn. codificatie oplossing verkend waarin beslissingen gedocumenteerd of gemodelleerd worden.

Voor codificatie is het van belang om een goed begrip te hebben van de concepten, die men wil codificeren. Om deze reden, is een conceptueel model van architectuur ontwerpbeslissingen ontwikkeld. Dit model beschrijft de delen waaruit een architectuur ontwerpbeslissing bestaat en hoe zij relateren met andere (bestaande) concepten. Op basis van dit model zijn twee aanpakken ontwikkeld. De eerste is de Archium aanpak voor het codificeren, managen en onderhouden van architectuur ontwerpbeslissingen in een forward engineering setting. De tweede aanpak is de Architectural Design Decision Recovery Approach (ADDRA) voor het terugvinden van eerder gemaakte architectuur ontwerpbeslissingen in een reverse engineering setting.

Beide aanpakken zijn op verschillende wijze geëvalueerd. Voor Archium is onderzocht hoe de aanpak algemene use-cases voor het managen van architectuur kennis ondersteund. Een tweede studie evalueert en vergelijkt Archium met andere aanpakken in de context van systeem evolutie. De evaluatie van ADDRA vindt plaats

door de aanpak toe te passen in een case studie.

Zowel ADDRA, als Archium, helpen een architect met het codificeren van architectuur ontwerpbeslissingen. Deze codificatie heeft twee belangrijke consequenties voor software architectuur. Ten eerste worden hierdoor ontwerp beslissingen expliciete verbindingen tussen een ontwerp en de redenen achter het ontwerp. Ten tweede, kan een software architectuur nu gezien worden als een set van architectuur (ontwerp) beslissingen. Beide gevolgen plaatsen software architectuur in een nieuw perspectief, vergroten onze kennis over software architectuur en helpen met het reduceren van architectuur kennis verdamping. Een interessante richting voor verder onderzoek is om te kijken naar andere types van architectuur kennis en hoe deze zich verhouden met architectuur ontwerpbeslissingen.

INDEX