Understanding Software Architectures: Tracing architectural knowledge in software architecture documentation

JENS RASMUSSEN

Supervisors
ANTON JANSEN, JAN SALVADOR VAN DER VEN, AND PARIS AVERGILOU
## Contents

1 Introduction 1
   1.1 Software Architecture Description 2
   1.2 Software Architecture Rationale 3
   1.3 A lack of traceability in software architecture description 3
   1.4 Contribution 5
   1.5 Context 5
   1.6 Organization of this thesis 6

2 Problem Definition 7
   2.1 Research Questions 7

3 Background 9
   3.1 Architectural Knowledge 9
      3.1.1 Traceability 10
      3.1.2 GRIFFIN domain model for Architectural Knowledge 12
      3.1.3 Related work 16
   3.2 Architectural Decisions 18
      3.2.1 Introduction 18
      3.2.2 Uses of architectural design decisions 19
      3.2.3 Problems in traditional software architecture 19
      3.2.4 Solving the problems by representing decisions as first-class citizens 21
   3.3 Current research in Architectural Decisions 22
      3.3.1 An ontology of architectural design decisions 22
      3.3.2 Templates for capturing architectural decisions 24

4 Analysis 27
   4.1 Software architecture documentation 27
      4.1.1 Barriers to documenting architectural knowledge 28
      4.1.2 Identifying Architectural Knowledge 30
      4.1.3 Conclusion 31
   4.2 First-class representation of Architectural Decisions 34
      4.2.1 Decisions are Essential Architectural Knowledge 34
      4.2.2 Templates for documenting decisions 35
      4.2.3 Templates solve problems of traditional documentation 35
      4.2.4 Handling Relationships 36
      4.2.5 Improving relations 39
      4.2.6 Conclusion 40
4.3 A Model of Architectural Knowledge ............................................. 42
   4.3.1 What is a model? ................................................................. 42
   4.3.2 Benefits of a domain model ..................................................... 42
   4.3.3 A domain model for architectural knowledge .............................. 43
   4.3.4 LOFAR domain model ........................................................... 45
   4.3.5 Traceability in the LOFAR Model ........................................... 46
   4.3.6 Conclusion .............................................................................. 47

5 Solution ........................................................................................................... 49
   5.1 Solution Concept ............................................................................... 50
   5.2 Environment ....................................................................................... 51
      5.2.1 The Knowledge Architect system .............................................. 51
      5.2.2 Word ....................................................................................... 53
   5.3 Requirements for the Word Client ...................................................... 53
   5.4 Implementation of requirements in the Word Client tool ..................... 54
   5.5 Implementation ................................................................................... 65
      5.5.1 Integration with existing tools .................................................... 65
      5.5.2 Offline use of documents with AK ............................................. 65
      5.5.3 Loading of documents is dependent on access/situation ................ 67
      5.5.4 Storage of metadata in documents ............................................. 67
      5.5.5 Use of different domain models ................................................. 68
      5.5.6 Network performance ............................................................... 69
   5.6 Future Work ......................................................................................... 72

6 Conclusion ....................................................................................................... 73

References .......................................................................................................... 77

Appendix ............................................................................................................ 79
   List of Tables .............................................................................................. 79
   List of Figures ............................................................................................... 79
Chapter 1

Introduction

"The reality is more excellent than the report." – Ralph Waldo Emerson

This thesis is about understanding software architecture documentation and how the integration of software architecture description and rationale can improve this understanding. Understanding software architecture (documentation) is important in the complete life cycle of a software architecture; during development, maintenance and evolution.

Understanding the documentation of a software architecture is difficult. A description of the design of a software architecture is far from enough to make it understandable. A description does not answer why a design is done as it is, or how the software architecture arrived at the present design. To understand a software architecture, one must know and understand the reasoning behind the design, i.e. the rationale, and how architectural artifacts are related.

Research in software architecture can be seen in two perspectives: a traditional and modern one. The traditional perspective describes the design of a software architecture in terms of components and connectors and how these are arranged. The second, more recent one describes software architecture in terms of the decisions and rationale that led up to the software architecture. Research and practice so far have been confined within these perspectives, but an integration of the traditional and modern perspectives to describe software architecture promises to solve some of the problems of software architecture.

Figure 1 illustrates these two perspectives. Both perspectives are about representing architectural knowledge, which is done in either a non-formalized or a formalized approach. The research areas that result from this subdivision are traditional software architecture documentation, modeling of software architectures, templates for capturing architectural decisions, and ontologies for representing architectural knowledge. This thesis will analyze the problems of understandability in these areas except modeling of software architectures, because this area is not relevant for documentation of software architecture. The focus lies on how understandability of software architecture documentation can be improved by integrating it with rationale.

The remainder of this chapter gives a short introduction to the traditional and modern perspectives on software architecture description and rationale. This is followed by a section on how an integration of these perspectives promises to solve some
1.1 Software Architecture Description

A software architecture can be described using either or both a documentation or modeling approach. Documentation is the use of natural language and diagrams to describe the software architecture. Models are visualized in diagrams which can describe the software architecture in different dimensions or views, each of which focus on a specific aspect. For example how components and connectors are organized, thereby providing a runtime view of the software architecture. See [1] for an overview of views for describing a software architecture. These approaches are not exhaustive. Usually natural language and diagrams are mixed in documentation, since they are complementary in describing the software architecture.

Traditional software architecture documentation is difficult to understand for many reasons. It is written in natural language, it is difficult to get an overview, because it is very heavy on details, and it has low readability. Furthermore, software architecture documentation requires that the reader has significant contextual and architectural knowledge about the described system. Contextual knowledge is needed because documentation does not describe the motivation behind the design or the decisions that led to it. All these properties hinder understandability of documentation.

Modeling describes software architecture in a formal manner using meta-models or architecture description languages. Modeling lets the user describe the concepts and their relationships of the software architecture, and offers specialized analysis features. Modeling is outside the scope of this thesis. See [2] and [3] for an overview and comparison of different modeling approaches for describing software architectures.
1.2 Software Architecture Rationale

Software Architecture Rationale is the documentation of the reasons behind the design of a software architecture. The rationale explains that the components, connectors, and constraints define a system that would satisfy the stakeholders’ concerns and requirements to the software architecture. Decisions are an important part of this, as their rationale describe why the system fulfills certain requirements. The rationale of these decisions describes why they fulfill requirements in a specific way. The type of rationale used in software architecture is split in a formal and a non-formal category, as illustrated in the bottom half of figure 1. One formal approach has been the creation of an ontology of design decisions, which classifies the concepts used in describing decisions and their rationale. An example of an informal approach is the use of templates to capture architectural decisions, which concentrates on providing agile documentation of decisions and rationale to enable knowledge sharing [4].

Templates are forms with predefined fields for specific information. Templates are informal, because they use natural language, but lend structure to information by predefining fields for what must be captured. This results in well-structured and consistent documentation, which can easily be shared. Templates for capturing rationale have been created by [4]. They offer a structured way to document architectural decisions and related information. However, they suffer from other problems which can be summarized as insufficiency in describing relations between architectural artifacts.

1.3 A lack of traceability in software architecture description

A major problem of software architecture today is understanding the documentation of software architectures. Documentation is often difficult to interpret and understand by people who were not involved in the initial design of a software architecture. Even software architects who created the design have problems understanding the design. This is because documentation describes a software architecture design in its current state and fails to capture the rationale and the decisions that lead to the software architecture. The effect of decisions is present in the software architecture design, but they are almost impossible to identify. This means that decisions and their rationale are quickly forgotten, and as a consequence, the software architecture documentation cannot be fully understood. This results in the following problems of software architecture ([5–7])

- Lack of first-class representation of decisions. When architectural decisions are not documented, the knowledge of why a decision is made and how it affects the design is quickly lost. This knowledge is almost impossible to recover, since decisions cannot be identified in the documentation.

- Design decisions are cross-cutting and intertwined. Decisions often affect multiple parts of the design, and are usually dependent on other decisions. To overcome this problem, documentation must not only capture decisions, but also their complex relations with other decisions and architectural artifacts.
• High cost of change. Due to the previous two points, it can be prohibitively expensive to change or remove existing decisions.

• Design rules and constraints are violated. When earlier design decisions are not available, these are easily violated during evolution of the software architecture.

• Obsolete artifacts are not removed from architecture. The consequences of removing obsolete artifacts are unknown, because it is not documented why they exist and how they are related to other artifacts. Obsolete artifacts are not removed out of a fear of breaking other parts of the design.

• Lack of stakeholder communication. When decisions and rationale are not documented and shared with stakeholders, it is difficult to perform tradeoffs and resolve conflicts of stakeholder concerns that the software architecture must address. This makes it difficult to set common goals for the software architecture since the reasons of it are not clear to everyone.

• Limited reusability. To reuse architectural artifacts efficiently, the rationale and tradeoffs of architectural artifacts must be known. Any alternatives must be known as well to avoid mistakes when reusing artifacts in different environments.

Architectural knowledge about decisions and rationale is vital to avoid these problems since these problems are caused by architectural knowledge vaporization. Knowledge vaporization occurs when knowledge remains in the heads of software architects, and is not documented or shared sufficiently. This tacit knowledge of decisions and design rationale is quickly forgotten when it is not captured in a more persistent form.

Knowledge vaporization occurs because there is a lack of knowledge and standards for documenting architectural knowledge [7]. For organizations and software architects, it is not clear what architectural knowledge is essential to document, and tool support for capturing this knowledge is insufficient. Knowledge vaporization impairs the understanding of the software architecture.

The understanding of software architectures is improved when it is possible to trace between the architectural knowledge necessary to understand the architecture design. The necessary architectural knowledge constitutes requirements, design, and the rationale of architectural artifacts [8–10].

Traceability is the ability to follow or traverse relationships between requirements and software architecture design in the software architecture life-cycle [8]. One aspect, traceability from requirements to design, is the ability of following the design elements that emerge from a given requirement. In other words, tracing the impact of a given requirement on the software architecture design. Traceability of requirements or design to rationale enables one to relate to the reason and justifications for these elements. In general, traceability means that the implicit relations between architectural artifacts can be traversed to determine the impact of artifacts on other artifacts in the software architecture, and to follow the reasoning behind this.

Traceability supports the software architect in the software architecture life cycle in many ways. This includes: explaining and verifying architecture design, identify impacts of changes and root causes of problems, and analyzing cross-cutting concerns [8]. Traditional software architecture lacks effective traceability methods to perform these tasks.
Architectural knowledge vaporization is the cause of incomplete architectural knowledge. Lost knowledge can be of all types of architectural artifacts and relationships between these. Knowledge vaporization results in a lack of traceability, since this is knowledge about the implicit relationships between requirements, design and rationale.

A lack of traceability inhibits the software architect in performing tasks in the software architecture life cycle mentioned above, which leads to an inflexible software architecture that is unnecessary complex, expensive to maintain and evolve, and suffers from a high degree of design erosion [5]. This decreases the understandability of the software architecture, and results in lower quality of the software architecture.

The GRIFFIN project [11] has identified some of the essential concepts of architectural knowledge, which consist of four interrelated groups: knowledge about people, processes, design, and decisions. The first three groups are well known, but the last, knowledge about decisions, has only recently received attention in research of software architecture rationale.

Architectural Decisions (AD) have been proposed as the missing link that bridges the gap between requirements, rationale, and software architecture design by capturing their implicit relations [4, 10]. Architectural decisions consist of major design decisions and the rationale behind them, and are related to stakeholders’ concerns and the design of the software architecture. When a software architecture is seen as the result of a set of decisions [9], then a decision describes a number of changes to the design and its rationale describes why these changes accommodate the requirements or stakeholder concerns. Thus, architectural decisions establish the missing link and enable traceability between requirements and design by being the vehicle of changes that led to the software architecture design.

1.4 Contribution

This thesis solves the problem of traceability in the domain of software architecture documentation by creating tool support for capturing architectural knowledge related to architectural decisions and enabling traceability of this to documentation. Traceability is achieved by modeling essential architectural knowledge and coupling the concepts of this model with the context in which they are used. This results in documentation that is easier to understand, because architectural knowledge relationships can be traversed.

The solution improves capturing and understanding essential architectural knowledge in software architecture documentation.

1.5 Context

This thesis was done as part of the GRIFFIN project in the SEARCH (Software Engineering & Architecture) group, and was done at the university of Groningen [12]. The GRIFFIN project researches architectural knowledge and uses thereof, and aims to create a grid for information on architectural knowledge. This work is done with several companies as case study partners, of which Astron is the case study partner.
of the SEARCH group. Astron is creating a very large array of radio telescopes, which generates large amounts of data. Their challenge is to produce a software-intensive system for the processing of this data. By participating in the GRIFFIN project, Astron is hoping to gather insight in how to improve the documentation of their software architecture.

1.6 Organization of this thesis

This chapter has given an introduction to the perspectives of research in software architecture, and the problems that a lack of traceability in software architecture documentation leads to.

The rest of this thesis is organized as follows. Chapter 2 outlines the requirements for enabling tool support for traceability in documentation. A number of research questions must be answered to reach this goal, and they are subsequently presented. Chapter 3 presents a background on architectural knowledge and decisions to enable the reader to understand this thesis. In chapter 4, these problems are analyzed, and the research questions are answered. Based on these results, a proof of concept tool for enabling traceability in software architecture documentation is developed, and this is presented in chapter 5. Chapter 6 presents a conclusion to this thesis.
Chapter 2

Problem Definition

A major problem of software architecture is the understandability of the software architecture documentation. To understand the documentation, one needs to understand the architectural knowledge embedded in the software architecture. A way to understand the architectural knowledge is by having traceability, such that one can follow the relationships between pieces of architectural knowledge.

The traceability of architectural knowledge should be extended to documentation. Undocumented architectural knowledge is quickly lost due to knowledge vaporization. Therefore, it is vital that the essential architectural knowledge needed to perform traceability is captured in documentation. Tool support is needed for both capturing and tracing architectural knowledge. To support effective traceability, a tool should be able to:

1. Represent architectural knowledge as first-class citizens
2. Support complex relationships between architectural knowledge
3. Capture architectural knowledge including relationships
4. Support traceability across relationships of architectural knowledge

2.1 Research Questions

To reach the goal of effective traceability, a number of questions must be answered.

1. Why is architectural knowledge in documentation difficult to understand?
2. What architectural knowledge is essential to understand documentation?
3. How can architectural knowledge be represented as first-class citizens in documentation?
4. How can architectural knowledge relationships be represented in a scalable way?
5. How can capturing of architectural knowledge be supported?

Chapter 4 will give an analysis that will answer these research questions. The first research question is answered in section 4.1, the following two are answered in sections 4.2 and 4.3. Section 4.3 also answers the fourth research question. The last research question is answered in chapter 5, where a proof of concept tool for capturing architectural knowledge and introducing traceability in documentation is presented.
The next chapter presents a background of architectural knowledge, architectural decisions, and current research in the latter area.
Chapter 3

Background

This thesis takes place in the domain of architectural knowledge and decisions. These are relatively new subjects within software architecture research, and this chapter introduces necessary knowledge and concepts to understand the rest of this thesis. Before the research questions can be answered, this background is presented such that readers not familiar with this area should be able to follow the discussions in later chapters.

The background consists of three sections. The first section is about architectural knowledge, where a general introduction is given to what it is, and how traceability of architectural knowledge is beneficiary to software architecture. This is followed by a presentation of a model of architectural knowledge developed in the GRIFFIN project.

The second section is about architectural decisions. The concept of architectural decisions presents a new perspective on software architecture, where instead of viewing a software architecture as a number of views, it is seen as a set of decisions. Using this perspective, it is hoped that some of the current problems of software architecture, such as architectural knowledge vaporization, can be reduced.

The third section presents some current research in the area of architectural decisions. An ontology describes architectural design decisions in a formal way, and categorizes what types of decisions exist and what information is relevant. This is followed by a presentation of an informal way of describing decisions using templates.

3.1 Architectural Knowledge

This section gives an introduction to how types of knowledge are kept in organizations and the problems associated with the different ways of keeping knowledge. The following section introduces a domain model for architectural knowledge, followed by a focus on the part of this domain model which is relevant for this thesis.

Architectural knowledge is knowledge produced in the software architecting process. The notion of architectural knowledge is broad, and there is no general agreement on what it should contain. In the knowledge management field, a distinction is made between three levels of knowledge:

- Tacit knowledge
- Documented knowledge
• Formalized knowledge

Tacit knowledge exists only in the heads of people or as unwritten rules in an organization, which are not made explicit, but still are known. Examples are experience and cultural norms, things which the holder of this knowledge often takes for granted or implicitly assumes. The latter two forms of knowledge are explicit and codifies knowledge in some form to improve sharing, learning, and assessment. The first of the explicit forms is documented knowledge, which is typically captured in natural language or other informal ways. Documented knowledge is all kinds of knowledge that is recognized to be valuable enough to be used by others. The second form is formal knowledge, which is knowledge codified in a formal language or model with exact defined semantics. The quintessential example of this being math. The advantage of formalized knowledge is that it can be exploited by computers.

The main problem of tacit knowledge is that it is easily lost. Since tacit knowledge is kept in the heads of employers, it is not clear who has the knowledge that is needed. This effect is called knowledge vaporization. The loss of architectural knowledge results in software architecture that is inadequate, wrong, and highly or prohibitively expensive to maintain and evolve. Architectural knowledge must be recovered, which besides being error-prone and time-consuming, is often not possible. The result can be that earlier design constraints are violated and that the design becomes overly complex when obsoleted artifacts cannot be removed, because of a fear of breaking the design.

To overcome knowledge vaporization, architectural knowledge needs to be moved from the tacit domain to the documented and formalized domain. To document knowledge, it must be recognized what knowledge is essential to be captured. Without resolving what knowledge must be documented, it will not be captured. When the knowledge is documented, it must be shareable and reusable to exploit it. This requires some standards for how to store the knowledge in an accessible and manageable way.

To capture and formalize knowledge, the essential concepts and their relations must be determined to codify it formally. A meta-model or formal language specifies the rules and relationships the captured knowledge must adhere to, and these concepts and rules must be extracted from the given domain. Formalization enables the use of computers to manipulate and exploit the knowledge in ways not possible or cumbersome with documented knowledge.

3.1.1 Traceability

Software architecture design is done top-down from a broad perspective to smaller detailed units of design. The resulting documentation starts with business requirements on an abstract level and is narrowed down during the development life-cycle to functional requirements, architecture design specifications and detailed design specifications. Traceability is the means to relate this information across levels of abstraction.

Requirements and design traceability allows tracing from requirements to design artifacts and vice-versa, which helps ensure that requirements and design match each other. Other aspects are tracing requirements and design to design rationale, and tracing the evolution of requirements and design. Traceability to design rationale
allows architects to see the rationale of specific points in the design and discover the impact of changes. Traceability can be supported in the whole development life-cycle, from stakeholder concerns to requirements to architectural decisions. Decisions eventually result in the creation of design artifacts and specifications for these.

Traceability is necessary for architecture development and evolution and to improve understandability of the software architecture. In summary, traceability helps with the following:

- To evaluate if design artifacts comply with requirements
- To assess maturity of software architecture trace
- To assess completeness of decisions and specifications so they cover all requirements
- To understand why decisions are made, and check they are made for the right reasons
- To understand impact of decisions on design, for example when requirements and decisions are changing

The following cases describe how traceable architecture design, design rationale, and decisions support the software development life-cycle:

- Explain architecture design - reasoning behind a design is linked through a causal relationship. A decision modifies/changes the design. An artifact’s existence can thus be explained by its decision (rationale).
- Identify change impacts - when a requirement or decision is proposed, the ripple effect of the resulting change should be traceable to analyse the change impacts to various parts of the system
- Trace root causes - when software defects, it can be due to many causes that all must be analysed and identified. Some can be because of conflicting requirements, constraints or assumptions, and require design rationale to help explain and identify them.
- Verify architecture design - capturing of traceable architectural knowledge (AK) supports (independent) verification of architecture design, without presence of original designers. This implies that capturing traceable AK enables knowledge sharing of a software architecture.
- Trace design evolution - decision/rationale results in changes and thus explains evolution of the design
- Relate architecture design artifacts - design artifacts may be related by concerns or requirements or constraints. This can be used to improve tradeoff decisions, such that seemingly unrelated artifacts can be traced when one is changed.
- Analyze cross-cutting concerns - cross-cutting concerns are concerns that cannot be confined in one part of the system, but impacts multiple and different parts it. These concerns are often non-functional requirements, some examples being security and performance, which are cross-cutting the architecture and often require tradeoffs for multiple decisions. Rationale of such decisions explains much about why and how the decisions have been made. A traceable architecture can relate otherwise disparate requirements and design artifacts that are part of the cross-cutting concern.
Thus traceability results in better quality assurance, change management and software maintenance, in effect a software architecture that satisfies stakeholder concerns and requirements. In conclusion, being able to trace design and requirements to design rationale helps software architects to understand, verify and maintain architecture design.

3.1.2 GRIFFIN domain model for Architectural Knowledge

The GRIFFIN project aims to make architectural knowledge explicit and has created a core model of architectural knowledge [13, 14]. It integrates four major parts to a model that defines concepts and interactions of architectural knowledge.

![Diagram of GRIFFIN Core Model of Architectural Knowledge]

Figure 3.1: The four parts of the GRIFFIN Core Model of Architectural Knowledge.

**Design**

The software architecture design has traditionally been seen as the heart of software architecting, as it is the result of the architecting process. Software architecture design can exist in all knowledge types, ranging from undocumented, tacit knowledge to natural language in the documented level and in architectural description languages at the formal level. Expressing the software architecture in views can cover both the documented and formal level, depending on the formal nature of the given view. Design is usually documented, or in the cases where it is not, it can be retrieved by reverse engineering. The context and some of the rationale may be partially retrieved from management documents, vision documents, requirement specifications, etc.

**Process**

Software architecture shape the processes in the organization and vice-versa. Organizational rules set up constraints on the software architecture, and software architecture impacts and transforms processes in the organization. Introducing an architecture in an organization cannot be done without considering the processes,
and the processes are thus a part of architectural knowledge. Processes are usually documented in management and vision documents.

People

The people involved in the software architecture are called stakeholders, because they each have some concerns about the direction of the software architecture. A fitting architecture must balance the concerns of its stakeholders and conflicts must be resolved. Knowledge about the stakeholders, their concerns and relationships forms a part of the architectural knowledge.

Decisions

Traditionally, knowledge about decisions has been tacit knowledge. Current research in software architecture puts decision-making at the heart of software architecting by arguing that a software architecture is the result of a set of decisions, and that decisions should be moved to the documented level. Interactions between people, processes and design happen through decisions, and are thus a central part of the architectural knowledge.

There are three levels of decisions, ranging in the scope and impact on the software architecture. The first are architectural decisions, which sets the stage for the software architecture indirectly. They are typically about people and processes involved in the organization. The second, architectural design decisions, are major decisions that influence the architectural style, high-level design and non-functional attributes directly and with a system-wide impact by setting the guidelines that further design should follow. Design decisions are usually decisions about more detailed design. A decision is significant and thus architectural when it has an impact on the whole or large parts of the system or a cross-cutting concern. The impact often only becomes apparent when the system must be changed, since it only then becomes clear if a decision is hard to change, therefore architectural design decisions and design decisions cannot always be distinguished.

In figure 3.1 above, the outermost arrows represent the interactions between the outermost boxes, but these interactions really happen through the decisions part. The levels of decisions can also be seen in the figure. Architectural decisions show that there is an indirect impact of decisions on people and processes on the design part. Architectural design decisions often involve the people part, as major decisions always are rooted in the concerns of the stakeholders. Design decisions do not have to involve processes or people since they are narrower in scope and can be rooted in subgoals. The next chapter elaborates on decisions.

Detailed view of the GRIFFIN model for Architectural Knowledge

The GRIFFIN domain model for Architectural Knowledge is a meta-model for representing architectural knowledge in the GRIFFIN project. The domain model has been developed to describe concepts for storing, sharing, and using architectural knowledge. The four areas in the figure above are represented, but with people replaced by stakeholder: stakeholder, process, design and decision. The full domain model is shown in figure 3.2 on the following page, and contains all four areas.
Squares represent domain objects (d.o.) and rounded squares represent domain functions that changes the state of one or more domain objects. Domain objects identify architectural knowledge concepts, and functions represent interaction between these concepts.

This thesis focuses on architectural design decisions which roughly limits the above domain model to the decision, design and part of the people parts. This subset of the GRIFFIN domain model is further explained below, and is illustrated in figure 3.3 on page 15.

A concern is a requirement in the perspective of a stakeholder and is usually expressed in terms of functional or quality attributes. Decision topics arise from concerns, since concerns are not very specific. A concern can often be distilled into more decision topics since a concern raises new and more specific questions. A decision topic is a question that a decision is the answer to, and is specific to one
3.1 Architectural Knowledge

Figure 3.3: Subset of domain model describing Architectural Design Decisions
subject, or attribute, such as performance, implementation language etc. An example is a concern about costs, which creates questions about different subjects such as how many man-hours are available, choice of database, programming language or framework to use. These questions are all different decision topics that stems from the same concern, and for which a decision must be made.

It is important to realize that decision topics and concerns can be n-to-n related. A concern has possibly many decision topics. Decision topics can be part of multiple concerns, e.g. when a decision topic for choice of database originates from a concern about costs, it can also be a decision topic for a concern about performance. This shows that highly intertwined architectural knowledge is supported.

A ranking of alternatives is weighted against different concerns in which the decision topic is relevant. For example, a choice of database can originate from a concern about cost, but the decision is also related to a concern about performance. So a ranking should not be based on only the concern from the decision topic, but also other concerns. However, since ranking is based on multiple concerns that are related by the decision topic, a way of finding these related concerns is useful. An alternative is a possible solution to a decision topic. All alternatives are evaluated and ranked to see how well alternatives can solve a decision topic, also when only one alternative has been suggested. The ranking is based on the concern from where the decision topic originated and other related concerns, and represents the rationale for choosing one alternative over another. An alternative becomes an architectural decision when chosen as the solution for a decision topic. The architectural decision is a subtype of an event since both a decision is a specialized event. An event can for example be related to a business or market change, and thus lead to new high-level concerns for the software architecture. An architectural decision is made internal to the organization, and can also result in new concerns being stated. This is for example the case when a high-level decision is made that creates concerns on a lower level. When an architectural decision is executed, it changes the architecture which will be reflected in a change to the models. A model is a part of the architectural design, which results from executing architectural decisions. Models can exist in many forms such as diagrams, text documents, and source code.

3.1.3 Related work

Recently, several other tools for capturing and using architectural knowledge have surfaced. These recent approaches all realize that architectural knowledge consists of more than design decisions and rationale, and they also capture design decisions and their relationships with requirements and architecture design.

Architecture Design Decision Support System (ADDSS), which is a web-based tool that provides traceability between requirements and design decisions [15]. It stores architectural design decisions following an iterative approach in the same way as architectures are developed and visualizes the evolution of the architecture over time. It captures architectural knowledge by combining mandatory and optional attributes to characterize design decisions. It sports a basic dependency model to enable dependencies between decisions.

Another tool is Archium, which models design decisions and their relationships with a range of concepts from requirement, decisions, architecture descriptions, to
3.1 Architectural Knowledge

It is implemented as an extension to the Java language and can turn source files into executable models.

[16] proposed an agile approach to capturing information on requirements, specifications and subsequent design decisions, and their relations in an agile environment. The information is captured by recording informal conversations, which are later annotated. Design rationale is extracted from the conversations, and a tool was created to aid capturing and annotating conversations. The traceability supported is from requirements to decisions in a narrowing scope. Being in an agile environment, little textual documentation is available and thus the tool captures what would remain undocumented knowledge.

[17] also used annotations for capturing design rationale from large, complex systems. In an attempt to solve the capture bottleneck problem, which is the lack of enough incentives to the users to invest time for sharing their knowledge or resources, a prototype of a unified repository of design information and rationale is created at low cost. Users are asked to annotate the design documents or e-mails to indicate design issues or design decisions. These annotations are then stored in the repository. The annotation scheme is reported to be promising in capturing rationale and improving communication.

The Architecture Rationale and Elements Linkage (AREL) is a UML-based tool to create and document architectural design with a focus on architectural decisions and design rationale. It models architecture design as causal relationships between design concerns, decisions and outcomes as UML entities [8].

[18] created a rationale-based architecture model for design traceability and reasoning. The model provides reasoning support to explain why design objects exist and what assumptions and constraints they depend on. Traceability techniques can be applied for change impact analysis and root-cause analysis to allow software architects to better understand and reason about an architecture design. The rationale-based model is implemented in UML, and a tool-set is implemented that support the capture and the automated tracing of the model.

The last tool is the Process-centric Architecture Knowledge Management Environment (PAKME), which is also a web-based tool that uses a data model for capturing and managing architectural constructs (such as design decisions, alternatives, rationale, and quality attributes), their attributes and relationships [19]. Each design decision is captured as a case along with rationale and contextual information using a template. PAKME is an extension to a collaborative platform and supports geographically distributed users.
3.2 Architectural Decisions

First a short introduction to traditional approaches of software architecture is given including the problems inherent. Then the concept and use of architectural design decisions is explained and how capturing and formalizing architectural design decisions promise to improve the areas where traditional approaches fail.

3.2.1 Introduction

Traditional approaches to creating software architectures have been defined with concern to the views used to convey different aspects of the architecture and to how the architecture is organized in components and connectors, and the relationships between these and the environment ([20–22]). By standardizing these definitions, the principles guiding the architectures design and evolution have matured to a point where architectural design and views on the architecture no longer represents great challenges for research ([5,23]).

The software architecting process is basically a series of decisions of what components a system is composed of and how the components relate, which should result in a software architecture that fulfills a set of requirements. [24] supports this notion by arguing that architecture results from effective decision-making, not from architectural view construction, and that understanding the architectural artifacts and their relationships is more critical to decision-making than understanding how selected components collaborate via architectural views. With current views of an architecture, it is difficult to see what decisions were made during the design process to get to the architecture shown in the views. Knowledge concerning design decisions, considered alternatives, rationale and assumptions for making a decision is usually not documented and is therefore often lost. Having a software architecture without knowing the decisions that led to this design or the rationale behind these decisions, makes it hard to understand why it was designed as it is. Views do not make or explain the design decisions made, views show the software architecture resulting from design decisions, which is useful to understand the result of the architecting process. Architectural decisions are a most significant forms of architectural knowledge as the software architecture cannot be fully understood without them. [10]

The software architecture is a result of a set of architectural decisions, which has changed and added constraints to the software architecture [9,10]. Architectural decisions are useful for explaining changes during the architecting process and why a decision was made. The figure below shows a change to a software architecture as a result of a design decision.

A definition of Architectural Design Decisions is given in [10]:

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.

This definition implies that design decisions are focused on requirements, since the (partial) realization of requirements in compliance with existing design constraints
3.2 Architectural Decisions

Figure 3.4: A decision changing the software architecture

is the goal of a design decision, and that the result of applying a design decision is a change to either the architecture or the design constraints. Alternatives are potential modifications to the software architecture with implications to structural elements, design rules and constraints.

3.2.2 Uses of architectural design decisions

Three important uses for first-class representation of design decisions are traceability, assessment, and knowledge management [25]. Design decisions add traceability upstream to requirements and downstream to design artifacts, and is useful during development and evolution of the software architecture. The second is assessment of maturity and evolutionary capabilities of the software architecture. The maturity can be assessed by reviewing decisions and tracing them to requirements, and the implications of decisions and alternatives can be used to assess the modifiability of the software architecture. The third use is knowledge management. By explicitly documenting design decisions, the architectural knowledge in an organization can be externalized to enable knowledge sharing and reuse.

To achieve these uses of architectural design decisions, documentation of a design decision should contain a description of at least the following items: the solution to a problem should be captured to document how it is solved, and the considered alternatives as they describe other solutions to this problem. Alternatives implicitly say something about the requirements and context when a decision is made. Thus the reasons why they are rejected contribute to the rationale. Alternatives also provide documentation of earlier considered decisions. This is useful to prevent decisions that violate earlier constraints, known as design erosion [5], if requirements or decisions are changed. The rationale of a decision is the reason why a specific solution was chosen, and the argumentation of how that solution better fits any requirements, constraints and other concerns.

By capturing design decisions and their rationale, the impact of architectural knowledge vaporization is reduced [5,9]. Therefore, design decisions are at least as important as views, and should be at the heart of software architecting. The next sections discusses problems in traditional software architecture.

3.2.3 Problems in traditional software architecture

Research in current software architectural traditional approaches have revealed the following problems ( [4,5]).
• Violation of design principles
• Obsolete artifacts are not removed from architecture
• Rationale and considered alternatives are not documented
• Documentation does not convey change history and implications
• Documentation does not trace to architectural elements

These problems lead to design erosion ([5]) and will be further explained in the following.

Existing design rules, principles or constraints in the software architecture may be violated later when the knowledge of these decisions is not explicit. When evolving the architecture, this knowledge must be retrieved either from explicit knowledge representations, such as views and design documents, or tacit knowledge in the organization. To some extent, design decisions can be recovered, but this carries a high cost, and the rationale of any decisions cannot be fully recovered [26,27]. Tacit knowledge is by definition not being shared, and it is often lost. Without the necessary architectural knowledge to avoid violating earlier decisions, software architecture evolution leads to design erosion.

Another problem is that obsolete artifacts are kept in the architecture, because their implications on other artifacts are unclear, or it is not known if they are needed, and to avoid risks and extra costs the obsolete artifacts are not removed ([5]), resulting in an unmanageable and rigid software architecture.

Architectural views break down in several ways. Design decisions have a cross-cutting impact on components and connectors and are highly intertwined with other design decisions, making views at the components and connectors level insufficient for documenting implications of design decisions ([5]). Views show only the resulting software architecture, and not the alternative solutions considered, decisions and their rationale cannot be retrieved. Without the alternatives and rationale captured, already considered alternatives might be discussed during evolution and earlier decisions will be violated ([4]). Views are insufficient for showing changes and the implications of these. The effect of changes are only implicitly visible in the architecture, since a series of design decisions has led to changes, but any single change cannot be extracted from the views, and much less the rationale behind it.

The organizational impact of design decisions is unclear since views cannot convey implications clearly. Components and connectors in the architecture cannot be traced directly to soft-goals or requirements resulting in a lack of focus on requirements.

The lack of documenting design decisions and their rationale makes it difficult to see how the architecture design process evolved and the reasons to why the components and relationships are arranged as is, cannot be understood without serious effort. Without this architectural knowledge, maintaining and evolving the software architecture can become prohibitively expensive, since earlier decisions must be rediscovered to not violate the design principles of the architecture. Implicit architectural knowledge can often not be rediscovered, or it is too expensive trying to. The loss of architectural knowledge leads to design erosion, and communication between stakeholders is hindered.
3.2 Architectural Decisions

3.2.4 Solving the problems by representing decisions as first-class citizens

Current research [9, 28] argues that representing design decisions as first-class citizens in software architecture can solve these problems. When design decisions are represented in the software architecture, also the rationale of these decisions and the considered alternatives are part of the software architecture. The problems caused by lack of or lost architectural knowledge are avoided as described in the next section.

A design decision either changes artifacts or constraints on the software architecture. When constraints are documented in the form of design decisions, it is easier to avoid violating design principles. Obsolete artifacts can now be removed since their rationale and implications can be traced from the design decisions which introduced the artifacts and design decisions that obsoleted them. The risk when removing the artifacts is significantly lowered, and the software architecture can be kept pristine with the result of improved maintainability and evolvability.

First-class design decisions also convey a change history and the implications of decisions on the software architecture. Each design decision has some implications on architectural artifacts or constraints, and a view of the artifacts before and after the decision can convey the changes. By chaining design decisions, a change history can be created.

All changes to a software architecture are implicitly done to realize requirements, but there is a gap between requirements and architectural artifacts that prevents traceability between them, resulting in a lack of focus on requirements. Design decisions bridge the gap between requirements and architectural artifacts. For example, traceability from requirements to architectural artifacts and vice-versa is necessary when assessing alternatives and thus choosing the best solution to realize a requirement. The gap is bridged since a design decision is focused on realizing requirements, and the changes and implications of a decision is conveyed by its impacts on the software architecture design.

Another major benefit of first-class design decisions is that they enable sharing and reuse of architectural knowledge, which would usually remain tacit, and improve communication between stakeholders. The first is of high importance to especially people who will later be involved in the software architecture, such as new software architects and maintainers, but also the original software architects benefit since their detailed architectural knowledge is often forgotten. Improved stakeholder communication ensures more transparency over organizational and business concerns and enables easier resolution of issues.

These problems are closely related to the important uses of design decisions mentioned above, and representing architectural design decisions explicitly is a way to solve these problems, because they enable traceability from requirements to design and capture the rationale behind decisions and thus the software architecture.

Some progress has recently been made in the research of capturing architectural decisions. The next section elaborates on the two major approaches.


3.3 Current research in Architectural Decisions

There is a general agreement that software architecture construction and evolution benefits from an explicit representation of architectural decisions. The problems presented in the previous section indicate a need of documenting architectural decisions. However, there is much uncertainty about what associated architectural knowledge is relevant to capture and document, and how to do it. This section discusses existing research in this area. [29] takes a formalized approach and created a classification of design decisions and their properties, and [4] takes an informal approach and created templates for documenting design decisions.

3.3.1 An ontology of architectural design decisions

[29] sets up an ontology of architectural design decisions. The ontology describes and categorizes the information that needs to be captured in architectural design decisions. [4] presents a template for documenting decisions and thereby states what knowledge is necessary to capture. The goal of both approaches is to capture enough information to make the architectural knowledge explicit to be exploitable for building and evolving software systems.

[29] identifies four types of decisions:

- Existence decision: These decisions concern what should exist in the software system, and can be of a structural or behavioral nature. The first is about existence of subsystems, components, partitions etc. The second is related to interaction of elements, or connectors. Existence decisions are usually visible in the design, but must be captured to relate them to other less-visible decisions, or alternatives.

- Ban or non-existence decision: It is important to capture why non-existing elements do not appear in design, for example in the case of rejected alternatives. During evolution, rejected alternatives must be known, to avoid reconsidering them, or to be reconsidered in the case of new or changing requirements.

- Property decision: This decision states a quality of the system, e.g. design rules, guidelines, or constraints. It can be part of fulfillment of quality/non-functional requirement of the system.

- Executive decision: These are business-driven decisions that affect the development process, the people, the organization, and choice of technology and tools, which can set all kind of constraints. They usually constrain existence and property decisions.

An architectural design decision has a number of attributes of which the following are essential:

- Epitome/Decision: A textual statement to summarize, list, or label decisions in diagrams and models.

- Rationale: The argumentation of why the decision was made. This can be a a tradeoff analysis, for example.

- Scope: Some decisions are limited in scope of system, time, design and development, or organization.
State: Design decisions evolve and exist in different states over the progress of a project. The following scheme has to match a specific decision and approval process, thus the scheme may be too simple or complicated. States can be used for queries and filtering when visualizing design graphs.

- 0 idea or obsolesced: capture ideas to not be lost; cannot constrain other decisions than ideas. Obsolesced meaning the decision became irrelevant
- 1 rejected: rejected decisions are kept for system rationale
- 2 tentative and challenged: what if scenarios, or previous decided/approved that is now being challenged
- 3 decided: current position. Must be consistent with related decisions
- 4 approved: as decided, but approved by review or board ceremony

Author, Time-stamp, History: The person who made the decision, and when. The changes to a decision should be recorded. This can typically be changes in formulation, scope and state.

Categories: This allows keyword tagging of design decisions, which is useful for expressing associations, which the taxonomy cannot easily capture, and exploring these associations. Keywords complement the taxonomy.

Cost: Financial impact of taking this decision.

Risk: Risks associated with design decision.

Not all attributes are applicable in all projects or organizations, e.g. the state attribute might be too simple or complicated for certain environments. Other attributes like Cost and Risk might be too difficult or expensive to assess, but they can be used for general ideas about the cost and risk, and they are important for the business drivers of decisions, and to stakeholders. This way, they become part of the rationale. Most attributes have a universal value for the scope attribute, but it should be possible to capture when the scope of a decision is limited, e.g. the decision is restricted to certain subsystems.

The ontology also classifies the relationships between architectural design decisions. [29] enumerates a list of relationships: Constrains, Forbids (excludes), Enables, Subsumes, Conflicts with, Overrides, Comprises (made of or decomposes into), Is Bound To (Strong), Alternative To, Related to (weak).

Some relationships are from higher level design decisions and encompasses multiple smaller design decisions, and the other way around, forming a design decision tree. This also means that the state of design decisions can be dependent of each other, with a dependency meaning either of the following: constrains, decomposes into, and overrides. The relationships are further explained in [29].

Design decisions also relate to external artifacts in multiple ways. The relations trace from/to design decisions trace technical artifacts upstream such as requirements and defects, and downstream to design and implementation elements, and management artifacts. The does–not–comply–with relationship is used to track portions of the system that are not compliant with design decisions or relationships, and can be used to track which things have to be changed where.

Other research focuses on a more pragmatic and less formal approach to documenting architectural decisions by using templates. The next section looks into this method.
3.3.2 Templates for capturing architectural decisions

[4] presents a template for capturing the essentials of design decisions in a less extensive taxonomy. The goal is to create agile and light-weight architectural knowledge that can easily be used by different stakeholders to get a clear understanding of the architecture according to their perspective.

[4] states that the template is business-driven, oriented towards fulfilling requirements or soft-goals of which a decision must contribute to if it is to be made. The decision-making is also kept to a minimum by deferring decisions that need not be made at a given point of time. This means that while assessing a decision there might be more focus and attention on it to capture the necessary information. Decisions that absolutely should be made are the ones that identify the key structural components, their externally visible connectors, and their relationships. Decisions are tested for architectural significance by an architect asking himself if the decision impacts one or more system qualities (performance, security, modifiability etc.), and in the positive case, the decision should be documented [4].

Two groups of stakeholders are accommodated by two decision views. The first is the template presented below and is for technical stakeholders. Grouping allows for filtering different aspects based on stakeholder interests. The second view is provided to management/business stakeholders where key decisions and implications are summarized in a PowerPoint presentation.

Agreements on design decisions must be socialized within the rest of the organization to reach final decisions. The design decision template is useful for providing a common language for discussing decisions since reviewers can easily see the status of the decision, the rationale and its impacts, and has proven more powerful than reviewing component models ([4]).

The following essential decision information is proposed to be captured.

- Issue: The design issue being addressed. To follow a minimalist approach, only issues that must be addressed now. Why an issue must be addressed now must also be documented
- Decision: The selection of one of the positions
- Status: Can be pending, decided, or approved. To keep documentation light, only approved decisions are documented.
- Group: Used to organize the set of decisions, e.g. presentation, data, integration
- Assumptions: From environment, e.g. cost, schedule, and technology used
- Constraints: Chosen alternative may pose additional constraints
- Positions: Viable list of alternatives/options with explanations/models/diagrams
- Argument: Reason why this position was selected. This can include implementation cost, availability of resources, total cost of ownership, time to market, etc.
- Implications: A decision may introduce new decision topics, pose new requirements or modify existing ones, add constraints, require negotiation of scope, schedule etc. A clear understanding and documentation of implications is useful for gaining stakeholder buy-in and create the roadmap for architecture execution.
• Related Decisions: Listing of other decisions. Meta-models, a traceability matrix, or decision trees have shown to be useful.

• Related Requirements: Decisions are mapped to requirements (as soft-goals). Decisions should be business-driven and only made when contributing to a soft-goal.

• Related Artifacts: Impacted architecture, design, or scope documents

• Related Principles: Decision should be consistent with enterprise principles to ensure alignment with domains/systems.

• Notes: For capturing discussion notes and issues during socializing process

The key points of [4] are the following:

Views are driven by decisions, rather than decisions by views, thus decisions are the primary view of the architecture. Views are used by stakeholders to discuss and make decisions.

Traceability between requirements, decisions and technical implementation is of utmost importance to ensure fulfillment of requirements, and can also be used for assessment of decisions.

The challenge of architecture construction is to gain consensus of architecture direction from stakeholders, and socializing architecture decisions makes this possible. The design decision documentation created is lightweight and agile to ensure that each decision can be communicated easily and separately.

In summary, both approaches help define and set standards for what knowledge need to be documented when capturing architectural design decisions. This ensures an architect is focused on fulfilling the requirements or soft-goals, it provides a common vocabulary enabling precision and clarity, it supports impact analysis of decisions on other decisions, concerns, and architecture assets, and also it creates the possibility of synchronizing views with decisions, so the views stay consistent with the decisions they represent.
Chapter 4

Analysis

4.1 Software architecture documentation

Architectural documentation is described in natural language with diagrams as a complementary way to visualize concepts and relationships. Natural language is useful for very detailed description, and diagrams or views are suited to give an overview of particular aspects and relations between architectural artifacts.

Architectural decisions and their rationale are essential parts of architectural knowledge (see section 3.2.4 on page 21). Documentation of architectural decisions and rationale support software architects in a number of ways that improve the quality of the software architecture. Rationale improves understandability of the software architecture by helping architects understand the reasoning behind the design and the trade-offs made. Decisions bridges the gap between requirements and design and allow software architects to analyze the impact of requirements on the design. This is useful for tracking down causes of issues and for verifying the design.

A problem to documenting architectural knowledge is that architectural design decisions and rationale are not being documented, but is kept as tacit knowledge in the minds of software architects, resulting in architectural knowledge vaporization. A survey on the use and documentation of design rationale [Tang] revealed that most organizations recognize the benefit of capturing design rationale to justify design, but that there are many barriers to actually do it. This survey by [7] showed several such barriers, and [5] and [30] stated additional barriers. The most significant of these are listed below.

- No perceived benefit
- Lack of budget and time
- Lack of standards
- Inadequate tool support

When architectural knowledge is documented, the documentation is still difficult to understand for individuals who were not involved in the initial design of the system [5]. The documented architectural knowledge is difficult to understand. The following factors explain why.
• Essential architectural knowledge, such as architectural decisions, is not documented, because of several barriers.

• Architectural knowledge is not easily identifiable in documentation

• Relationships between architectural knowledge entities cannot be conveyed

Decisions and rationale are an essential part of architectural knowledge. These are often not documented, which results in a need of contextual knowledge of the software architecture to understand its documentation. Furthermore, documented knowledge is not easily identifiable as it is not apparent what type of knowledge is denoted. Lastly, while relationships between architectural knowledge can be captured in the text medium, it is very difficult to structure this information in a readable way, and understanding these relationships is crucial to understanding a software architecture.

In the following, the barriers to documenting architectural knowledge will first be analyzed, followed by an analysis of the problem factors for identifying and understanding architectural knowledge in documentation.

4.1.1 Barriers to documenting architectural knowledge

The following discuss the barriers to documenting architectural knowledge mentioned above.

No perceived benefit

One barrier is the software architects themselves. Documenting decisions have no direct benefit, but shows it is valuable on a longer term, such as when maintaining and evolving the software architecture. Software architects know what decisions they have made, and documenting them have no perceived benefit. Then they skip the documentation to do other work. Documentation is a time-consuming task, and software architects are usually highly constrained in amount of time available. Since they already know the decisions made, no benefit is seen in documenting them, only extra work that is outside their core activities. The activity of documenting is therefore given a low priority, as it is perceived as hindering software architects in performing more important tasks. However, decisions are easily forgotten and cannot be identified in a resulting design.

This barrier is contrasted with the fact that most software architects describe access to decisions and their rationale as highly beneficial in later phases, such as when verifying, maintaining, and evolving a design.

Lack of budget and time

Survey respondents cite a lack of budget and time, which indicates that documenting design rationale is an activity that, while it has a perceived usefulness, it does not have a perceived urgency. Also, it is not clear how important it is later in the software architecture life cycle.

Budget and time reasons shows a lack of awareness of the long term costs. These are major reasons for documenting design rationale, because long term costs incurred
by lost rationale can become prohibitively expensive when maintaining and evolving a software architecture.

**Lack of standards**

Another barrier is a lack of knowledge of what to document. Many organizations do not know what architectural knowledge is necessary to document to make documentation understandable. One reason cited is a lack of standards for how to document the rationale of software architecture design. While the area of software architecture description has matured, resulting in standards for how software architecture design should be documented in different perspectives, software architecture rationale is a rather new area in research, and even more so in industry. Industry recognizes that rationale is important, but it is a vague realization that lacks how-to knowledge. When companies document rationale, this lack of standards results in unmanageable documentation.

The IEEE 1471 establishes recommended practices for documenting software architectures [20]. This standard identifies a need to document rationale for architectural decisions and descriptions. However, in the standard there is no mention of how to document this rationale or what architectural knowledge is essential [31].

**Inadequate tool support**

In the mentioned survey, respondents cited a lack of tool support to document design rationale. The most used tools for documenting are Microsoft Word, Powerpoint, and Visio. The first two do not have any functionality specifically related to documentation of capturing architectural knowledge, and the last is a tool for creating diagrams, suitable for documenting software architecture descriptions.

Software architects experience the activity of creating documentation as an interruption in their working routine. Design rationale capturing tools have been created, but they have been separate tools. When software architects must write documentation along side their work, these tools require a context switch in architects’ focus. This interrupts them in their work by having to switch to different tools to capture a specific aspect of software architecture documentation. This breaks the architect’s focus and reduces his productivity, thereby being perceived as annoying and hindering them in doing ‘real work’.

Another reason is a lack of knowledge of what architectural knowledge is important to document. This goes hand in hand with a lack of standards and processes to guide why, how and what to document, and a lack of tool support for capturing the architectural knowledge. Other reasons stated in the survey are a lack of a formal review process, and that it is not required for non-complex solutions, a fear of getting into long cycles of design review, and that the dynamic nature of technology and solutions make it useless to document design rationale.

The reasons given are interdependent; a lack of standards means that it is not clear what and how design decisions should be documented, and a lack of suitable tools is partly a result of a lack of standards, and the time and budget constraints make it difficult to establish standards and investigate tool support. [29] argues
that the extra burden of capturing decisions and rationale outweighs any immediate benefits, and that the benefits are only becoming obvious on a longer term.

4.1.2 Identifying Architectural Knowledge

Section 4.1.1 analyzed the barriers for why essential architectural knowledge is not documented. When architectural documentation is present, it is difficult to read and understand. This section analyzes the problems of identifying architectural knowledge in documentation. The following problems will be analyzed:

- Text suffers from a lack of overview
- The type of knowledge cannot be identified without contextual knowledge
- Relationships between architectural knowledge cannot be conveyed

Text has lack of overview

Text has certain properties that in general make it difficult to identify the information stored. First, text is a linear format. This means that to understand a piece of text, the reader must start reading at a chapter or section and must continue until the desired information has been identified. The linearity means that text has few entry points for the reader to jump into the text. Second, text has bad visual properties. It is dependent on the layout, but in general it is difficult to get a quick overview of the contents of documents that are large and densely packed with information, such as architectural documents. The consequences of these properties are that it is difficult to get an overview and identify pieces of architectural knowledge in the text.

For architectural documentation, the overview is improved by standardizing the document structure. Formatting and structuring give clues to what a section of text contains, but is not sufficient to present architectural knowledge in an easy understandable way. Structuring improves the high-level overview, but fails to improve the identification of architectural knowledge.

Types of architectural knowledge

Architectural documentation is difficult to understand without contextual knowledge of the software architecture. Architectural documents do not contain sufficient architectural knowledge required to understand a software architecture from its documentation [REF]. [5] noted that documentation is difficult to interpret and use by individuals that were not involved in the initial system design, even when the major design decisions are documented.

The problem is that the types of architectural knowledge in documentation are implicit. Which type of knowledge is represented must be interpreted by the reader, and this requires knowledge of the decision-making process and rationale, which only the initial software architects possess.

[5] stated that architectural decisions must be promoted to first-class citizens, because while their effect on a software is implicitly present, decisions are almost impossible to identify in a design. This proposition applies to documentation of
software architectures as well, and can be generalized to other types of architectural knowledge that must be identifiable.

Text has limitations in conveying relationships

This results in severe limitations for describing architectural knowledge, because architectural knowledge is highly intertwined. For example, architectural decisions have many relationships with other decisions and can impact large parts of a system. They also have implicit relationships with requirements and other types of architectural knowledge, such as design and discarded alternatives. By documenting decisions, the relationships between requirements and design are implicitly documented. The implicit relations can be very complex, especially in the case of cross-cutting concerns, and these complex relations cannot be captured sufficiently in textual documentation. In addition, these implicit relationships must be interpreted by the reader since they are not described explicitly in documentation.

These shortcomings mean that relationships between architectural knowledge are very difficult to recognize, even when they are documented implicitly as in the case of architectural decisions. The involved knowledge fragments must first be discovered and identified, before any relations can be discovered. Both knowledge fragments and their relations are implicit as these too first must be identified by their type. This is made more difficult since related knowledge can be in distinct places in documentation. To identify relationships, the reader must potentially hold all discovered architectural knowledge in memory before any relationships can be discovered. The reader must create his own overview that the text cannot supply to try to make the documentation understandable. This is the reason why documentation is difficult to understand for people not involved in the initial design.

This was illustrated by an experiment done in the GRIFFIN project [32]. The experiment showed that, except for software architects with significant experience with the architectural document, relationships between architectural knowledge types were very difficult to discover. It was easier when related knowledge was locally close, and when some architectural knowledge was explicitly mapped to other parts, which was done on a high-level only using tables mapping requirements to specifications for example. More specific knowledge contained in single or few sentences, constituting the bulk of architectural knowledge, was next to impossible to relate to requirements for people with less contextual knowledge about the software architecture.

The difficulty of discovering architectural knowledge and how it is related is a consequence of the text medium having low readability, conceptual difficulty of the problem area to write text in a highly structured way, and that architectural knowledge stays implicit in architectural documentation. This results in that in-depth architectural knowledge is needed to read architectural documentation.

4.1.3 Conclusion

Traditional documentation of software architectures has several problems that impairs its understanding. First, there are many barriers to documenting essential architectural knowledge which is needed to understand the documentation. These
barriers include budget and time reasons, a lack of standards for what to document and effective tool support to capture architectural knowledge.

Second, these barriers result in a lack of documentation of architectural decisions and rationale. These are essential parts of architectural knowledge, which are necessary to capture the implicit relationships between requirements and design. This architectural knowledge is essential to enable traceability between architectural knowledge, which in turn is necessary to understand the implications of decisions and thus the rationale of the software architecture.

Third, reading architectural documents requires contextual knowledge of the software architecture. The types of architectural knowledge in a document are implicit and must be identified to be understood in the context of the software architecture. An in-depth knowledge of the software architecture is needed, and less experienced people must rediscover the implicit architectural knowledge contained in documents. Thus, the documentation cannot be understood without having experience or other in-depth knowledge of the documented software architecture.

The understanding is further impaired, because documentation cannot convey the relationships that exist between the represented architectural knowledge. The reader must build up an understanding of the documented software architecture by having access to fragmented knowledge, and implicit relationships must be rediscovered and interpreted. This is a very difficult process which requires significant contextual knowledge.

How to improve the situation

To overcome the barriers of documenting architectural knowledge, tool support must be created for capturing it. One issue is the break of focus when documenting. To overcome this issue, a tool should be integrated with existing tools in the software architect’s toolbox. It must also be easy to use and should offer the software architect some additional benefit by using the tool to overcome the problem of lack of perceived benefit. Traceability is such a benefit, because it aids the software architect in understanding and analyzing the software architecture documentation.

To solve the problems of understandability, several things must happen. The essential architectural knowledge and its relationships must be made explicit in order to create understandable documentation for people with less or no experience with a software architecture.

This essential architectural knowledge, such as decisions and rationale must be documented and be represented as first-class citizens. This forms the basis for enabling traceability between requirements, design, and decisions/rationale. Representing architectural knowledge as first-class citizens also improves its identification.

A way to support relationships between architectural knowledge entities in documentation must be devised to capture the implicit relationships of architectural knowledge, which are otherwise lost. Making relationships explicit is also needed for creating tool support for traceability.

Challenges

- The essential architectural knowledge must be defined.
• Architectural decisions need to be explicitly represented in documentation.
• Documentation should support an overview of architectural knowledge, so differ-
  ent types of architectural knowledge can be identified and distinguished from
  each other.
• Tool support should be integrated with existing tools.
4.2 First-class representation of Architectural Decisions

The previous section concluded that architectural knowledge needs to be explicitly represented in documentation. Many problems are caused by the lack of documentation of a particular part of architectural knowledge, namely architectural decisions and rationale. This section looks into architectural decisions. First, a short look at why decisions are essential architectural knowledge. The rest of the section analyses templates, which are used for documenting decisions and promoting them to first-class citizens of documentation. First, how they improve software architecture documentation, then an analysis of their shortcomings.

4.2.1 Decisions are Essential Architectural Knowledge

The previous section analyzed several barriers to documenting essential architectural knowledge. These barriers result in that architectural knowledge essential for understanding architectural documentation, is not being documented. This section elaborates on why documentation of architectural decisions and rationale is essential to understand a software architecture. [9, 33] introduced a new perspective on software architecture, where a software architecture is thought of as a set of decisions. This perspective recognizes the importance of decisions in understanding and explaining the design of a software architecture.

In software architecture documentation, architectural decisions and rationale are usually not documented. This results in a gap between requirements and design, that prevents tracing from any of these artifacts to the other. Architectural decisions bridge this gap and makes tracing possible [10], because they capture the implicit relationships between requirements and design, or more generally, the relationships between architectural knowledge entities.

Decisions also enable traceability from architectural knowledge artifacts to rationale. For any given artifact, the rationale of its existence and how it is related to requirements can be found. Figure 4.1 shows how decisions bridge the gap between requirements and design. Because a decision leads to a change of the design, it connects the ‘what must be done’ of the requirement with the ‘what has been implemented’ of the design. The rationale explains why this decision best satisfies the related requirements. Decisions thus enable tracing from requirements to design and vice-versa by describing how these are related, and the rationale improves understanding of a software architecture design by explaining why decisions have been made and how they satisfy requirements.

When decisions and rationale are not documented, the implicit relationships between requirement and design artifacts are easily lost, and since rationale remains tacit knowledge, understandability of the software architecture becomes a problem. Hence, the following problems: rationale must be reconstructed to avoid that new decisions violate existing constraints and decisions. This is a costly and time-consuming activity, often ridden with mistakes. The criteria and environmental factors that influence the architecture might have become unclear because of knowledge vaporization. Business goals, constraints, and design integrity might be violated, bad or misunderstood trade-offs are made, and the impact of changing requirements and
environment on the design cannot be assessed.

4.2.2 Templates for documenting decisions

Representing architectural decisions in a template form has been proposed as a solution for documenting architectural decisions and their rationale, thereby raising them from a tacit to a documented knowledge level. This section elaborates on how templates in general improve software architecture documentation.

Templates are forms used to document architectural decisions and rationale and relationships to other architectural knowledge. They provide an overview of a software architecture by explaining the background and reasoning of significant decisions. The forms contain predefined fields for what must be captured to describe decisions and their relationships with architectural artifacts such as requirements and design. By defining a structure for documenting decisions and rationale in documentation, decisions and rationale are raised from a tacit knowledge level to the documented level.

[4] has proposed a template for architectural decisions, which defines the necessary knowledge to capture. It uses a lightweight and agile approach to documenting architectural decisions, hence only the most significant architectural decisions are supposed to be documented. However, in general templates show how architectural decisions can be promoted to first-class citizens in documentation.

Templates are ideal for knowledge-sharing of decisions to stakeholders, because it is simple to identify and extract architectural knowledge. See section 3.3.2 in the background chapter on page 24 for a list of information captured by templates. This information is not further discussed here as there is a general agreement to which information must be captured [4, 29]. How templates handle relationships is more interesting, and is discussed in section 4.2.4.

4.2.3 Templates solve problems of traditional documentation

Templates are a way of promoting decisions to first-class citizens in documentation. Templates solve many of the problems plaguing software architecture documentation:

- Remove some barriers to documenting essential architectural knowledge by defining what to document
- Promote decisions to first-class citizens

Figure 4.1: Decisions bridge the gap between requirements and design, enabling traceability between architectural knowledge.
• Architectural decisions and rationale are documented
• Improve overview and identification of architectural knowledge

The barriers to documenting architectural decisions and rationale were primarily caused by a lack of knowledge of what architectural knowledge is important to document. A part of this problem is the lack of standards for documenting architectural knowledge. Another barriers is a lack of tool support.

Templates focus on documenting decisions and creates awareness of their importance while supplying a method of doing so. By predefining what information to capture, templates set a de-facto standard for documenting decisions and rationale. Software architects can continue using existing tools such as word processors, because a decision template for such tools can easily be made.

Templates solve the lack of overview that textual documentation suffers from by structuring information in fields. Because these fields are predefined, it is clear to the software architect what architectural knowledge is essential to capture. This also imposes structure, which improves the overview of decisions, and specific architectural knowledge can easily be found and identified by its field.

By documenting architectural decisions explicitly in a template separate from other documentation and in a structured manner, they are promoted to first-class citizens of documentation on the level with other standardized ways of software architecture documentation.

Templates support capturing of relations to related requirements, decisions and assumptions. For each decision, all important relationships must be listed.

4.2.4 Handling Relationships

Decisions are always related in some way to other decisions. These relationships must be represented in documentation of the decisions to enable traceability. This section will analyze support of relationships with decisions and other architectural artifacts in templates as proposed by [4].

A template has five fields for filling in relationships to other artifacts. The different fields make it possible to describe different types of relationships:

• Implications
• Related Decisions
• Related Requirements
• Related Artifacts
• Related Principles

Implications describe any new decisions, requirements, or constraints, or modifications of these, resulting from the decision. It can also contain implications for other concerns, e.g. that scope or schedule must be renegotiated. Related decisions describe a network of decisions which are dependent of each other. A high-level decision typically narrows the problem space and results in new, more constrained, decision topics. In figure 4.2.4 below, an example of decision dependencies can be seen. Related requirement states the requirements that the decision is a, possible partial, solution to. Related artifacts state which components in the software
architecture is affected by the decision. Related principles state any architectural principles or guidelines upon which the decision has been made.

**Decision tree**

When decisions result in new decisions, the new decisions have a smaller scope than the originating decision, such that the decision space is narrowed down and the design is converging. This creates a decision tree that moves from architectural decision to more detailed design decisions. Figure 4.2.4 below shows such a decision tree. A decision to create an online store results in new decision topics within the scope of the decision. In the figure is shown the next decisions, that the online store should be accessible through both a browser interface and a Web Service API. This last decision results in a decision to use SOAP as protocol. Each decision limits the scope of the problem space and often creates a need for making more decisions.

![Decision Tree Diagram](image)

Figure 4.2: A tree of decisions narrowing in scope

Templates can describe the tree model of decisions using the field 'Related decisions' to describe relationships. When a new decision is documented, the originating decision is added as a relationship. This allows bottom-up traceability, but is not sufficient for tracing in the other direction to find originating decisions. To allow bidirectional tracing between two decisions, their relationship must be represented in both. That means that any modifications to decision relationships involve all the decisions in the relationship. This quickly results in unmanageable decision relationships.

- Relationships are one-way only
- Bidirectional traceability is unmanageable

**Complex relationships**

Decision relationships can, however, be more complex than a tree structure. Decisions are highly intertwined, and this is clear in different ways. First, some decisions
are based on cross-cutting concerns, that affect multiple parts of the software architecture. For example, a security requirement often has system-wide effects. All decisions which affect such a concern are related by their impact on this quality attribute. In figure 4.2.4, a decision graph is shown. The decision of using access control is cross-cutting and its relationships with other decisions are complex.

![Decision Graph](image)

Figure 4.3: Decisions can be influenced by cross-cutting, but seemingly unrelated decisions

Second, when a decision is changed, it can have a ripple effect that reaches other decisions in many ways\(^1\). If a decision is changed, then the constraints on which dependent decisions were made might not be valid anymore. Therefore, these decisions must be reconsidered.

Figure 4.2.4 shows a change to a decision of choice of web service protocol. In this case, changing the decisions from using SOAP to using REST changes some constraints on the system, which in this case has the effect of changing the decision of what framework to use. This change could result in other changes too, such as decisions related to deployment, planning etc.

Decisions often have ripple effects. To document these effects, the precise relationships must be captured in all involved decisions. This allows traceability from a changing decision to other affected decisions. Because decision relationships are of a complex nature, and the number of relationships can be expected to be significant, managing these manually quickly becomes unfeasible. When more relationships are added, the work to capture and maintain the documentation increases explosively.

**Tracing from other architectural knowledge to decisions**

The implications of a decision on other architectural artifacts are noted in a template. All templates which influence a specific artifact must state this implication, which enables traceability to the artifact. However, when the reverse relationships must be traced, all templates must be looked through to find the templates where the quality

---

\(^1\)This also illustrates why a software architecture is difficult to change.
attributes is stated. This is the case when the rationale for a design artifact must be found. This problem is general to all relationships described by templates, that relationships are supported in a one way direction only. This enables traceability from the decision to requirements, design and other decisions stated in the template, but traceability starting from requirements and design to decisions quickly become unmanageable, because templates scale badly in this aspect.

This problem raises the question of how to start tracing from requirements or design in a scalable manner. The documentation of requirements and design should also relate to dependent decisions to support this aspect of traceability. Templates are meant for documenting decisions alone and do not consider this aspect of traceability.

- Two-way traceability between decisions is supported, but is not scalable
- Only one-way traceability to other artifacts is supported
- Managing relationships does not scale

### 4.2.5 Improving relations

Templates are still to a high degree defined by the textual nature of capturing architectural knowledge, and text is not sufficient to describe two-way n-to-n relations.

A solution to the problem of one-way traceability from decisions to other architectural knowledge is to represent other architectural knowledge entities as first-class citizens too. In general, normalization of properties of architectural decisions allows finer grained relationships to be formed. The right level of normalization must be determined, such that only the essential properties that must be traceable are extracted into separate entities. This enables decisions with implications for the same quality attribute to reference the same first-class representation of this quality attribute.

![Diagram](image)

**Figure 4.4:** (a) describes how templates document relations, (b) describes two-way n-to-n relations.

Figure 4.2.5 describes how templates support relations versus a model supporting two-way relations. The left part, a, shows an abstract view of three decision templates. The decisions all have implications for performance of the system. The right part, b, describes two-way relations where the quality performance attribute is promoted to a first-class entity. This enables tracing from decisions to the quality
attribute, and vice-versa. In addition, the dependencies of decisions are described through the quality attribute. This solution to describing complex relations can be generalized to other architectural knowledge artifacts. A disadvantage of normalization is that more relationships must be created to model the relationships, but this also improves traceability since this is supported from all entities.

4.2.6 Conclusion

Decisions and rationale are essential parts of architectural knowledge. So essential that it has been proposed that the software architecture can be seen as a set of decisions [9]. Therefore, decisions must not only be documented, they must be promoted to first-class citizens of the documentation. By enabling traceability from requirements to design and back, and capturing the rationality of the decisions and thus also the design and how it relates to requirements, decisions as first-class citizens greatly improve the understandability of software architectures.

Templates are a way of capturing architectural knowledge on decisions. They are natural language forms that offer a consistent and structured way of capturing architectural knowledge. By making it possible to capture decision information, decisions are moved from the tacit to the documented knowledge level, and decisions become first-class citizens in documentation. Templates define what architectural knowledge is essential to capture, and includes both rationale of a decision and its relationships with other decisions and architectural knowledge. Because decisions are now first-class citizens and in a structured form, architectural knowledge is easier to identify, which decreases problems of knowledge vaporization. Finally, templates strengthen the consensus that decision and rationale capturing is necessary for understanding of software architectures.

Templates or the use of predefined forms has been shown to be a suitable way of representing decisions as first-class citizens in documentation. Furthermore, templates set standards for what knowledge is essential to capture, and improve the identification of decisions and rationale in documentation.

However, templates do not succeed in solving all problems. Templates form an attempt at solving the problem of traceability, but falls short in several ways. Traceability between decisions requires a lot of effort to document and maintain. The relationships included only support one-way tracing to related architectural knowledge, which is insufficient to enable the needed aspects of traceability. Only decision traceability is supported, tracing from requirements or design is not. The tracing that is supported does not scale beyond simple decision relationships.

The problems of traceability are rooted in the text media used and, ironically, an insufficient definition of architectural knowledge. Promoting the related architectural knowledge to first-class citizens would enable a better scalable model to capture relations.

The areas where templates are lacking to support traceability have been identified. They result in the following challenges to implement better capturing and traceability of architectural knowledge.
Challenges

- Alternatives should be better documented, or generally, the capturing of architectural knowledge should be more complete.
- The complex nature of relations between decisions must be supported.
- Traceability from requirements and design artifacts to decisions and rationale must be supported.
- Implications of alternatives is important architectural knowledge that is necessary when requirements are changing and decisions must be modified.
- The description of relations must be scalable.

Possibilities

- Other architectural knowledge artifacts than decisions can be represented as first-class citizens to make relations scalable and support all aspects of traceability.
- Capturing alternatives and their implications allows for creating a more complete documentation of architectural knowledge, which is useful when decisions must be changed.
4.3 A Model of Architectural Knowledge

This section gives a short introduction to models and how the definition of a model for architectural knowledge is beneficial to capturing and tracing architectural knowledge. The previous section discussed templates as a way of promoting architectural decisions to first-class citizens in documentation. Models present another way of promoting concepts to first-class citizens. The next section introduces a domain model for architectural knowledge used in the case study organization, following by an analysis of how it supports capturing and tracing of architectural knowledge.

4.3.1 What is a model?

A model is an abstraction of the real world, which enables reasoning about the properties captured in the model and leaving out details that would exist in the real world. For example, a blueprint for a building is an abstraction of the real building. The blueprint abstracts away details such as building materials to focus on other aspects such as design and shape. Having a model allows the architects to better reason about the building and to focus on specific aspects.

A domain model is a map of concepts and their relationships of a particular domain. Conceptual entities are first-class citizens of the model and together with their relations, the model is a structural description of the domain. The model focuses on and is constrained to the modeled concepts, which describes the semantics of the modeled domain.

To define a domain model, the significant concepts of the domain and how they interrelate must be recognized. This is done in a process called meta-modeling which is the construction or recognition of concepts within a domain. This process requires the knowledge of domain experts to determine the significant properties of the domain and highlight these in the properties of the model being created. The resulting domain model describes the constructs and rules, composing a formalized specification of entities and their interactions in the domain.

4.3.2 Benefits of a domain model

A domain model is an abstract and formalized model of a domain, and have several benefits:

- **First-class definition of concepts.** A model defines what concepts can be expressed. This effectively sets a standard for what to use the model for, since it is limited to its modeled concepts. Concepts modeled as entities are first-class citizens. Capturing data in the model must adhere to these concepts, and a clear structure is guaranteed. This also improves communication by introducing a common vocabulary. People can think and communicate using the concepts of the domain. Since the concepts are clearly defined in the model, misunderstandings caused by ambiguities in natural language are avoided. By focusing only on important concepts, irrelevant details are left out and reasoning is improved, since an abstract model allows to focus on its salient properties.

- **Complex relationships.** The structure of the domain model defines the possible relationships between concepts. These can be very complex as the model
is not limited in the number of relationships supported.

- **Tool support.** A domain model is a formalization of concepts, and can be represented on a computer. In general, a domain model is a graph where concepts and relationships make up the nodes and edges of the graph. This allows tool support for capturing and relating domain-specific data. Capturing data must be consistent with the modeled concepts, and any relations to other concepts must follow the model’s structure. This guarantees consistency and well-formedness of captured data. Modeled data is also easy to identify as it is captured as concept of the model.

- **Traceability.** The relationships between concepts can easily be traversed as these concepts are linked together. This enables traversing the relationships of one concept to go to another.

The following shortly describes how the domain of architectural knowledge benefits by applying these benefits. A domain model of architectural knowledge enables the manipulation of architectural knowledge concepts in a computer tool. By representing concepts as first-class citizens, tool support can be created for capturing and identifying architectural knowledge concepts in documentation. Traceability is supported by providing a way to traverse the relations between concepts. General properties of a model can be calculated using well-known graph algorithms. Such calculations can support the software architect in evaluating the software architecture in many ways. Some examples are checking completeness of decisions, identifying important decisions, and change impact analysis [34]. Tool support can also provide visualizations of architectural knowledge. This improves readability, overview, and identification of architectural knowledge in software architecture documentation.

### 4.3.3 A domain model for architectural knowledge

[7] mentions three steps to create a domain model for architectural knowledge that supports traceability.

1. Identify required information.
2. Create model to retain architectural knowledge, its rationale, and relationships
3. Use the model to support effective traceability.

First, it must be identified what architectural knowledge that must be captured. This includes concepts that make up different parts of architectural knowledge and the relationships between these concepts. Identifying concepts requires knowledge of domain experts.

Second, a model must be created which represents the identified architectural knowledge. The model retains architectural knowledge concepts, including their rationale, and how concepts relate to each other. For the best results, a domain model is defined in an iterative way, where discovered concepts are modeled, related, and evaluated to be in accordance to the real world domain. It requires significant time and effort to model domain concepts correctly.

Third, the created model enables traceability between concepts, and thus traceability between architectural knowledge.
Constraints on a domain model for architectural knowledge

To support capturing and traceability of architectural knowledge, the previous sections concluded on a number of challenges:

- Architectural decisions must be modeled as first-class citizens. A model must define the essential concepts of architectural knowledge to capture. Explicit architectural decisions have generally been recognized as enabling traceability, because they capture the implicit relationships between requirements and design.

- Other essential concepts of architectural knowledge must be modeled to support traceability. Templates were insufficient for supporting effective traceability, partly because they only allow to describe one-way relations from decisions to other artifacts. A domain model supports relationships and thus traceability between all concepts in the model. Exactly which concepts of architectural knowledge are essential must be determined, although requirements, design, decisions and rationale have been recognized to be essential. Traceability thus concerns four aspects ([4, 8]), which are needed to support full traceability:
  - Traceability between requirements and decisions, which is essential to understand why decisions were made
  - Traceability between decisions and design, which is essential to understand the impact of decisions on the system
  - Traceability between requirements and design to rationale, which is essential to assess how well the design and requirements match and to understand the system
  - Traceability between decisions, which is essential to convey a history of changes and how the decisions have evolved the software architecture

- Architectural knowledge has highly intertwined relations. To support highly intertwined relationships, the concepts must be modeled at a sufficient granularity, and their relationships determined.

- It can be domain specific. The modeled concepts are dependent on the vocabulary used in a given organization. Concepts can vary somewhat in name, scope and semantics, but should be mappable to a general model of architectural knowledge [13].

Capturing knowledge

Using a domain model implies that a way of capturing data in this model must be determined. The model itself only supplies the definition of data to capture, and leaves capturing as an exercise for the user. Capturing becomes a manual effort where it must be determined what type of entity some data is, before it can be inserted in the model. How to support capturing architectural knowledge is one of the research questions of this thesis, and will be answered in the next chapter where a solution is presented. In the following, a domain model for architectural knowledge, which has been defined in the GRIFFIN project, is presented.
4.3.4 LOFAR domain model

Previous research on architectural knowledge discovered concepts of architectural knowledge used in multiple organizations participating in the GRIFFIN project. The concepts used to describe architectural decisions and the life-cycle of these were used as a basis to create a generic model of architectural knowledge [11,13]. One of these case studies resulted in the LOFAR domain model for modeling architectural knowledge about decisions.

The rest of this section will introduce the LOFAR domain model and explain how it supports capturing and tracing of architectural knowledge.

Concepts

This section explains the domain model used to model the case study Software Architecture Documentation. The following list gives an overview and explanation of each knowledge entity in this model:

- Knowledge Entity. The main element in the knowledge model. All entities in the model are Knowledge Entities (KE). Knowledge entities have an ID, a name, and a description of its rationale.
- Concern. A concern is an interest to the systems development, its operation or any other aspect that is critical or otherwise important to one or more stakeholders.
- Requirement. Something that is demanded from the system. A specific type of concern.

Figure 4.5: The case study domain model of architectural knowledge.

- Decision Topic
- Concern
- Alternative
- Quick Decision
- Decision
- Requirement
- Specification
- Risk

- originates from
- raises
- addresses
- creates
- chooses

• Knowledge Entity. The main element in the knowledge model. All entities in the model are Knowledge Entities (KE). Knowledge entities have an ID, a name, and a description of its rationale.
• Concern. A concern is an interest to the systems development, its operation or any other aspect that is critical or otherwise important to one or more stakeholders.
• Requirement. Something that is demanded from the system. A specific type of concern.
• Risk. A risk is a special type of concern, which expresses a potential hazard, and needs to be emphasized. Decision Topics can arise from a risk, so a risk is a specific kind of concern.

• Decision Topic. A design decision is made to solve a certain problem. The problem can be how specific requirements can be met or how the design can improve on some quality aspects. Often stated as a question, e.g. where does the data transport layer consist of?

• Alternative. To solve the described problem, one or more potential alternatives can be thought up and proposed.

• Decision. For a design topic there are often multiple alternatives proposed, but only one of them can be chosen to address the described decision topic. The decision is which alternative to use.

• Quick Decision. If only one Alternative is stated and no (sufficient) motivation is given why this alternative is chosen, we have a quick decision.

• Specification. A specification is the lowest level architectural design decision that is being made. The refinement process for the architect is finished when it comes to specifications.

The architectural knowledge entities in the LOFAR domain model reflect the decisions-making process. Entities are modeled at a granularity where decisions have been decomposed into different entities. This enables capturing of architectural knowledge in an upfront manner where the software architect can document the decision-making process in steps leading up to a decision. Therefore, less information must be captured at each step. It is like that this will result in a more complete capture of knowledge and that software architects perceive the task as less work than a posterior documentation effort will.

The domain model has shown that it is mappable to the core model of architectural knowledge [13]. This indicates that it is fairly complete regarding decision modeling.

4.3.5 Traceability in the LOFAR Model

This section look into how the LOFAR model supports traceability.

Traceability of decisions to decisions is supported both in a narrowing scope and between decisions at the same level. As shown in figure 4.3.4, a decision can raise new concerns, which in turn raises decisions topics which lead to new decisions. When decisions create concerns and decision topics at a lower level, the scope is narrowed until decisions are specifications, which is a subtype of a decision. This is the most detailed level of decision-making for the software architect. Decisions leading in other decisions at the same level are typically related via cross-cutting concerns. The decision topics raised can be at the same level, but scoped at other subsystems. This is illustrated by decisions with ripple effects to several subsystems.

Rationale is captured within each domain model concept, such that it is documented for each step in the decision-making process. This enables tracing rationale from any architectural knowledge entity.

Traceability of requirements to decisions and decisions to requirements is similar to the above, since a requirement is a subtype of concern.
To enable full traceability, traceability to design artifacts or descriptions must be supported. Design artifacts are not modeled as first-class citizens in the LOFAR model. This is a challenge that must be overcome to support traceability in documentation of software architectures.

Complex relationships

The LOFAR model supports complex relationships between architectural knowledge entities. The main reason is that it defines the architectural knowledge concepts in a complete and fine-grained way. Compared to templates, the LOFAR model covers a wider range of concepts, and thus allows for a much higher level of granularity in relationships. Some template properties have been promoted to first-class citizens, such as alternatives and requirements. The explicit representation of these concepts allows complex relations to be captured while keeping the model scalable. In templates only limited traceability is supported by filling in fields for related requirements, implications, decisions, etc. Since relations can be created across all entities of the model, it supports complex and highly intertwined relationships in architectural knowledge.

In summary, the traceability properties of the LOFAR domain model:

- Traceability between requirements and decisions
- Traceability between decisions and design
- Traceability between decisions in a narrowing scope and between decisions on cross-cutting concerns
- Traceability from and to rationale is supported by tracing to the relevant knowledge entity
- Traceability from and to design is not supported since design is not part of the model

4.3.6 Conclusion

This section introduced a domain model as a solution to representing architectural knowledge as first-class citizens and their relationships in a scalable way.

To summarize, a domain model for architectural knowledge has the following benefits:

- Formalizes the representation of data
- Concepts are promoted to first-class citizens
- Traceability between concepts is supported by traversing the model
- Enables tool support
- Properties can be calculated to support computer-aided analysis

The concepts of a given domain can be formalized in a model of the domain. A formalized model gives a number of advantages; A domain model of architectural knowledge enables the manipulation of architectural knowledge concepts in a computer tool. By representing concepts as first-class citizens, tool support can be
created for capturing and identifying architectural knowledge concepts in documenta-
tion. Traceability is supported by providing a way to traverse the relations between
concepts.

Tool support can incorporate functionality to support the software architect in
several ways. General properties of a model can be calculated using graph algorithms.
Some examples are evaluating the maturity of a software architecture, identifying
important decisions, and change impact analysis [29]. Tooling can also support the
reader of software architecture documentation to a better understanding by providing
visualizations, and helping to identify architectural knowledge.

The LOFAR domain model responds to several of the challenges concluded on
in previous sections; it defines the essential architectural knowledge in the concepts
modeled, architectural decisions are explicitly represented, alternatives, which should
be better documented, are modeled, and it supports complex, scalable relationships
and traceability between requirements, decisions, and rationale.

There are three remaining challenges. First, how can traceability be extended
to design, which in the context of this thesis means documentation. Stated the
other way around, how can the domain model, and thus architectural knowledge,
be integrated with documentation? Second, how can architectural knowledge in
a domain model be integrated with documentation. Third, how can capturing of
architectural knowledge be supported in a usable way. These challenges will be
answered in the next chapter, where a solution is presented together with a proof
of concept tool for capturing architectural knowledge and enabling traceability in
software architecture documentation.

Challenges

• Extending traceability from the domain model to design descriptions in docu-
mentation.

• Integrating architectural knowledge in domain model with documentation.

• Capturing knowledge in a usable way.

Possibilities

• Use the LOFAR domain model to represent architectural knowledge and rela-
tionships to enable traceability.
Chapter 5

Solution

This chapter describes a solution to the problem of traceability in software architecture documentation. First, goals stemming from the problem definition and from the results from the analysis chapter are summarized to make it clear what a tool must support. Second, the solution concept is explained, after which requirements derived from the goals are presented. Third, it is shown how the proof-of-concept tool supports the above mentioned goals. This is followed by an overview of important decisions and concerns considered during the development of the tool. Finally, the section on future work discusses improvements to be done to the tool.

The problem definition stated the following challenges, which are restated here as goals, that a tool must support:

1. Represent architectural knowledge as first-class citizens
2. Support complex relationships between architectural knowledge
3. Capture architectural knowledge and their relationships
4. Support traceability across relationships of architectural knowledge

The following list summarizes challenges which were remained after the analysis chapter. These, too, are goals:

5. Integrating architectural knowledge in domain model with documentation.
6. Extending traceability from the domain model to design descriptions in documentation.
7. Tool support should be integrated with existing tools.
8. Capturing knowledge in a usable way.
9. Visualize types of architectural knowledge to improve identification and overview

Goals 1, 2, and 4 can be supported by using a domain model for architectural knowledge. Regarding goal 3, the use of templates have shown to be a viable manner of capturing architectural knowledge.

To solve goal 5, information in the domain model must be combined with documentation such that the user can access and manipulate it directly. 6 is related to 5, but concerns the integration of traceability, such that the user can trace from architectural knowledge to design descriptions. Together, 5 and 6 state that traceability
in the domain model for architectural knowledge must be extended to documentation, such that first-class knowledge entities also have first-class representations in the documentation.

Goal 7 requires that a tool must be built as an extension to or closely integrated with a documentation tool. Software architects do not want to use other documentation tools than they normally do, so this is required for a tool to be usable. Goal 8 is another usability requirement. Capturing architectural knowledge in the domain model should be easy. Goal 9 concerns visualization of architectural knowledge in documentation, such that the user easier can create an overview of the knowledge contained.

5.1 Solution Concept

The concept of the tool is based on a combination of different approaches to documenting architectural decisions and knowledge. Templates provide forms for capturing knowledge in a structured way, but were not sufficient for capturing relationships and thus traceability in architectural knowledge. The LOFAR domain model provides solutions to the problems of templates by defining the essential concepts of architectural knowledge and allowing traceability between these.

Furthermore, it must be integrated with existing tooling, which typically is Microsoft Word.

Architectural knowledge is documentation in natural language, and this knowledge must be made explicit. The idea is that when writing documentation, the software architect adds annotations to the text containing metadata about architectural knowledge\(^1\). An annotation is defined as follows:

\[
\text{An annotation is extra information associated with a particular point in a document or other piece of information. – Wikipedia}
\]

Annotations can be added to fragments of documentation, and as such does not interfere with the documentation itself. The metadata includes information on what type of knowledge is represented in an annotation, corresponding to entities in the LOFAR domain model, and how it is related to other fragments of architectural knowledge. By capturing domain model data in metadata, the metadata forms a graph of related knowledge across documents. The metadata can also contain other useful information, if needed.

Fragments of documentation can usually not be taken out of their context without breaking understanding. This approach has the advantage of making the types and relationships of implicit architectural knowledge explicit while retaining the architectural knowledge in its context.

When annotating a knowledge fragment of knowledge, two objects are created. The first object is an architectural knowledge entity (KE), which represents a domain model entity, including any relations. The second object is an annotation, which consists of a KE and a context, in which the KE is used. An annotation is thus

---

\(^1\)During the creation of the LOFAR domain model, the annotation of architectural documents provided a valuable way of discovering types of architectural knowledge used in the case study organization.
a local object in a document, while a KE is global, meaning that it exists in the Knowledge Architect repository, and can be referenced in multiple contexts.

Capturing architectural knowledge will be done with the use of forms, that must be filled in with the above mentioned information when annotating a fragment of documentation. Forms have the same benefits as templates, as they predefine what information must be captured.

The tool is integrated with Microsoft Word, which is the most used tool for documentation. Word has a rich application programming interface and a plugin model, which allows building extra functionality for manipulating documents. Furthermore, it is possible to store metadata in documents.

The (documented) architectural knowledge in a software architecture is located in several documents across the organization. This calls for a central knowledge repository. This allows relating architectural knowledge located across multiple documents, for example relating requirements in one document with design decisions in another.

This implies that KEs are stored in the repository, while annotations are stored in documents. This also ensures that a KE can be used in multiple contexts.

5.2 Environment

This section discusses the platform on which the tool is implemented. The tool is built as part of the Knowledge Architect tool suite to support capturing of and using architectural knowledge in an architectural knowledge repository called the Knowledge Architect.

The Knowledge Architect (hereafter Knowledge Architect) backend and the Word Client projects were developed in parallel, allowing the needs of the Word Client to be reflected in the Knowledge Architect API. The Knowledge Architect API is largely based on functionality required in the Word Client. Functional responsibilities were placed such that the Knowledge Architect API can be used by multiple clients and the Knowledge Architect Word Client is only loosely coupled to the Knowledge Architect. The Knowledge Architect was implemented in a different thesis project.

5.2.1 The Knowledge Architect system

The Knowledge Architect is a repository for architectural knowledge and access to architectural knowledge is given through the Knowledge Architect API. It is ontology based and uses a domain model to specify the types of architectural knowledge and their relations supported by the system. The API is implemented as a web service, so other systems easily can be created. The rationale for creating a central knowledge repository was requirements of interoperability with partners in the GRIFFIN project, and concurrency, since many users should be able to manipulate data in the Knowledge Architect at the same time.

Figure 5.1 gives an overview of the Knowledge Architect system. The upper part describes the user interaction part, consisting of various clients. The processing of the data to the database is done in the API. The API also makes sure the various clients can access the data in the database. The database contains all architectural
The Knowledge Architect

knowledge fragments and their relations. The Knowledge Architect can be configured to work with different domain models.

Being ontology based means that semantic relationships between architectural knowledge data can be enforced and exploited. Some functionality to analyze these relationships have been implemented, for example a completeness check, which check if a decision topic have been completely satisfied by the related decisions.

The knowledge stored in the Knowledge Architect consists of architectural knowledge entities, which are instances of the entities in the domain model used. A key driver in the creation of the Knowledge Architect is flexibility, because the domain model for architectural knowledge have changed in the past, and to accommodate organizational differences regarding architectural concepts. Therefore, the Knowledge Architect must be able to interpret various kinds of model definitions for knowledge representation. This means that the Knowledge Architect is independent of the domain model used, and can be configured to use other domain models. Information on the domain model in use, such as knowledge entities and relations supported can be obtained through the API.

The interface supports create-read-update-delete (CRUD) functionality for architectural knowledge entities, and information about the domain model in used, and uses of AK- functionality (completeness check).

Notes: This design for a knowledge grid will eventually enable full traceability from requirement to decisions to design, whether the AK is stored in documents or formal model of the software architecture.

Figure 5.1: Knowledge Architect system architecture.
5.2.2 Word

The tool is built as a plugin to Microsoft Word 2003 in Visual Studio 2005. Word has a .NET API which gives access to all functionality in the program and to document data and methods.

5.3 Requirements for the Word Client

This section introduces the requirements for the Knowledge Architect Word Client. Table 5.3 shows the requirements for the Word Client. Each requirement has been derived from the high-level requirements stated in the begin of this chapter, which is indicated in the 'from' column.

<table>
<thead>
<tr>
<th>ID</th>
<th>Goals</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR1</td>
<td>1,3,5,8</td>
<td><strong>Annotation of architectural knowledge in documents.</strong> Capturing architectural knowledge must be supported by annotating fragments of architectural knowledge in documentation and formalizing these in the domain model.</td>
</tr>
<tr>
<td>WR2</td>
<td>3, 7</td>
<td><strong>Word Client supports editing and removal of existing annotated knowledge entities in documents using the context menu.</strong></td>
</tr>
<tr>
<td>WR3</td>
<td>5</td>
<td>A user can <strong>add additional information to a knowledge entity.</strong></td>
</tr>
<tr>
<td>WR4</td>
<td>2, 3</td>
<td>The Word Client supports connecting knowledge entities to other entities existing in the Knowledge Architect backend, when adding or editing an annotation.</td>
</tr>
<tr>
<td>WR5</td>
<td>2,3,4,7</td>
<td>Knowledge entity in document can <strong>reference knowledge entities in the Knowledge Architect backend.</strong></td>
</tr>
<tr>
<td>WR6</td>
<td>2, 4</td>
<td>Completeness of entities can be checked and visualized. The Word Client can check if an entity has all its required relations, e.g. does a decision topic have alternatives and has a decision been made. The seriousness of problems can be visualized using colors.</td>
</tr>
<tr>
<td>WR7</td>
<td></td>
<td><strong>Traceability from a knowledge entity to another.</strong> The user can jump to related knowledge entities in a document.</td>
</tr>
<tr>
<td>WR8</td>
<td>5, 9</td>
<td><strong>Visualization of annotated knowledge entities based on type.</strong> Annotations can be colored in the document according to their type. This is a feasible way of creating an overview of the knowledge entities in a document.</td>
</tr>
<tr>
<td>WR9</td>
<td>5,7,9</td>
<td>A <strong>reference list of annotated knowledge entities</strong> can be created in the Word Client. The list must show related entities of each entity.</td>
</tr>
<tr>
<td>WR10</td>
<td>5, 7</td>
<td>Documents containing knowledge entities must be <strong>distributable</strong>, such that annotations in documents are readable on systems without the Word Client installed.</td>
</tr>
<tr>
<td>WR11</td>
<td>1, 2</td>
<td><strong>Domain model configuration.</strong> A model definition which contains the usable knowledge entity types and their relations can be input. This enables to tool to be used with different domain models.</td>
</tr>
<tr>
<td>WR12</td>
<td>7</td>
<td>Word Client is an <strong>add-in for Word 2003,</strong> and it should be <strong>easy to install</strong> on systems with Windows XP and Office 2003.</td>
</tr>
<tr>
<td>WR13</td>
<td>7, 8</td>
<td>The Word Client can <strong>export annotated knowledge entities</strong> from documents to an XML file.</td>
</tr>
<tr>
<td>WR14</td>
<td>3, 8</td>
<td><strong>Use Knowledge Architect API to store and retrieve knowledge entities.</strong> Architectural knowledge must be usable across tools and available organization-wide.</td>
</tr>
</tbody>
</table>

Table 5.1: Requirements for the Word Client
5.4 Implementation of requirements in the Word Client tool

This section shows how the requirements to the Word Client tool are fulfilled. First, a short introduction to the Word Client tool is given, before it is shown how each requirement is fulfilled.

The Word Client functionality is reached through a toolbar and a context menu. In figure 5.2, the toolbar is shown. It consists of eight buttons. From left to right, these are:

Add a knowledge entity Creates annotation with new knowledge entity from marked text in document
Add a context to existing knowledge entity Creates annotation with existing knowledge entity stored in Knowledge Architect repository
Create knowledge entity table Creates a reference table of all knowledge entities references in document. The table is inserted in an appendix to the document
Color knowledge entities Visualizes knowledge entities based on their type.
Show all knowledge entities Shows a list of all knowledge entities in a document
Export annotations to XML Annotations can be saved to an XML file.
Show completeness Visualizes the completeness of the relations of knowledge entities
Import knowledge entities from document to repository A document with annotations can be used to fill the Knowledge Architect repository with architectural knowledge. This is useful when sharing architectural knowledge or starting a new project.

WR1. Annotation of architectural knowledge in documents.

Figure 5.3 shows the form for adding an annotation to an architectural document. This can also be done via the context menu as shown in figure 5.4.

When a fragment of architectural knowledge is annotated, a knowledge entity (KE) is created in the Knowledge Architect repository. The KE references the context in which it is created. In the document, the KE and its context forms an annotation. A KE can be referenced by multiple annotations, which means that architectural knowledge concerning the same KE can be spread across multiple locations in documents. When this is done, a new annotation is created referencing the same KE. This is one way the Word Client supports highly intertwined architectural knowledge. Another is by the use of relating knowledge entities, which is described later.
5.4 Implementation of requirements in the Word Client tool

Figure 5.3: Creating annotation of a fragment of architectural knowledge.

Figure 5.4: Using the context menu to annotate a fragment of architectural knowledge.
WR2. Editing and removal of existing annotated knowledge

The architecture shown here is based on small (cheap) Ethernet switches and limited all-to-all performance. This provides enough bandwidth and flexibility to fulfill all requirements on the network. In the detailed design we may choose to provide more all-to-all bandwidth by using for instance stackable switches.

Figure 5.5: Editing existing annotation.

Figure 5.5 shows how an annotation in a document can be edited. In the context menu, some options for handling the knowledge entity is presented. The Edit option will bring up the knowledge entity form shown in figure 5.3 on page 55. Any changes to a knowledge entity is saved in the Knowledge Architect repository. The Remove option will bring up a confirmation dialog box, and if answered positive, the annotation will be removed.

WR3. Add additional information to a knowledge entity

See figure 5.3. In the notes field, extra information can be filled in. This is useful for saving discussion points and status of the KE, and would preferably be done using a link to an external document.

WR4. Connecting knowledge entities to other entities

Figure 5.6 shows adding of relations. When a type is selected, relations to other knowledge entities can be added. When pressing the 'Add' button, the type of relation type can be selected. Then knowledge entities to relate to can be selected. The Word Client limits possible relations to those that are valid according to the domain model used. This is done by retrieving information on relevant relations from the Knowledge Architect API.

Relations to other knowledge entities can be added.
5.4 Implementation of requirements in the Word Client tool

Figure 5.6: Selecting relation type.
WR5. Reference knowledge entities in the Knowledge Architect backend

This requirement is closely related to WR4. The knowledge entities that can be related are always retrieved from the backend, so refer to the above description for more information. See also figure 5.7.

WR6. Completeness of entities can be checked and visualized.

Knowledge entities can be checked for completeness. This is useful for determining the maturity of a software architecture. The completeness check is based on model checks on the domain model to identify incomplete parts. For example, a decision topic which have not been decided by a decision is incomplete. This indicates that there are loose ends, i.e. decisions that must be taken in the software architecture.

The document can be colored according to completeness levels. The tool distinguishes between four levels ranging from high to low severity:

- **Red** One or more primary rules are violated.
- **Orange** The primary rules are adhered to, but one or more secondary rules are violated.
- **Yellow** Both primary and secondary rules are passed. However, the KE has not achieved the status of validated yet.
- **Green** Both primary and secondary rules are passed. In addition, the KE has been validated by a reviewer.
Primary rules are those rules that check whether a knowledge entity is complete enough to provide a minimum level of traceability. This minimum level should ensure the existence of at least one reasoning path a reader could follow. Secondary rules focus more on the completeness of the architecture design. The rules are dependent on the domain model, as they use the concepts and relationships of the domain model to attach semantics to missing information. The completeness check is implemented in the Knowledge Architect repository, and completeness information can be retrieved through its API.

In the LOFAR domain model, the following rules are defined. The primary rules are:

- All Alternatives address one or more Decision Topics.
- All Decision Topics are addressed by at least one Alternative.
- All Decisions choose exactly one Alternative, unless it is a Quick Decision.
- All Decision Topics have an originating Concern or Alternative.

The secondary rules are:

- Concerns, that are not Requirements or Risks, are created by Alternatives.
- Chosen Alternatives and Quick Decisions either create at least one Concern, or raise at least one Decision Topic, unless the choice is made by a Specification.
- Quick Decisions should not have chooses or chosen by relations to other KEs.
- A Quick decision should be the only Alternative addressing a Decision Topic.
- Exactly one Alternative should be chosen for a Decision Topic.

Completeness levels can be visualized in a document. See figure 5.8 for an example where annotations are colored according to their completeness levels. Information on which completeness checks has failed for a specific annotation can be obtained through the context menu, see also figure 5.9.

WR7. Traceability from a knowledge entity to another

The user can right-click on an annotation, and the context menu will show existing relations to other KEs. By choosing a relation, the Word Client will jump to the place in the document where the KE is located. This provides useful traceability among KEs. See figure 5.10 for an example.

WR8. Visualization of annotated knowledge entities based on type.

The Word Client must be able to visualize annotations in a document such that the types of architectural knowledge can easily be recognized. This enables a software architect to gain a quick overview over the embedded architectural knowledge just by browsing through a document. Visualization is done similar to visualization of completeness levels. Each type of knowledge entity has its own color, and the context in a document will be colored according to this. In figure 5.11, an example is shown.

Coloring can be enabled and disabled by pressing the color icon in the Word Client toolbar. When completeness levels are shown, type coloring is disabled.
1.1 Purpose of this document

This document gives a description of the CEP facility hardware and software architectures. The CEP facility is described from multiple viewpoints in order to focus on specific aspects of the system. This document should provide enough information for the detailed design and construction of the subsystems.

1.2 Applicable documents

The role of the current document in the CEP facility engineering workflow is shown in Figure 1.

The final product is the architecture definition. The architectural process and reasoning leading to this definition is reported in [3]. The document contains the results from the analysis and architectural design process. The database architecture is described in [1] and assumed to be known in the remainder of this document.

![Figure 1](dependency_relations.png)

**Figure 1** Dependency relations between the CEP facility engineering documents. The document numbers refer to LOFAR-ASTRON documents.

This architecture description will be the basis for detailed design documents of the various CEP subsystems. Some preliminary detailed design information is given in the draft detailed design document [3].

Details of the applications that run on the CEP facility are described separately from this document. Each application scenario or special application will be described in a dedicated application definition document.

So far the following application definition documents exist, see Table 1:

![Figure 5.8](visualization.png)

**Figure 5.8:** Visualization of completeness check.

![Figure 5.9](completeness_coloring.png)

**Figure 5.9:** Completeness coloring and error message.
5.4 Implementation of requirements in the Word Client tool

WR9. A reference list of annotated knowledge

To create an overview of all annotated architectural knowledge embedded in a document, a table can be generated as shown in figure 5.12. By clicking name of the KE, the user jumps to its context in the document. A table of all architectural knowledge entities in a document can be generated to give an overview.

WR10. Distributable documents

Documents with annotated knowledge must be usable for people who do not have the Word Client installed. Annotations are saved in a document as comments stored in XML format. This ensures that the annotated knowledge can be read (manually) when the Word Client is not installed. The comments are created with a specific user name, so a reader can also turn off these comments. See figure 5.13.

WR11. Domain model configuration.

The Word Client retrieves the domain model from the Knowledge Architect. The domain model defines what types of architectural knowledge entities are available, and what relations they can form. The domain model is configured in the Knowledge Architect, and can be created using Protege/OWL. Figure 5.14 shows the definition of a domain model in the Protege/OWL tool. Since the Word Client is dependent on the domain model used in the Knowledge Architect, it can retrieve information on this from the Knowledge Architect API.

WR12. Easy to install add-in for Word

The Word Client is implemented as an add-in for Microsoft Word 2003. It is packaged in a standard installer, which installs all dependencies, such as the .NET framework,
2.1 Top-level specifications

Table 2: Top-level specification of the CEP facility

<table>
<thead>
<tr>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correlator capacity</td>
<td>Nominal: 17 stations, 1 FOV, 32 MHz, 16 bit samples</td>
</tr>
<tr>
<td></td>
<td>Full: 34 stations, 23 FOVs, 32 MHz, 16 bit samples</td>
</tr>
<tr>
<td>Beamformer capacity</td>
<td>One array: 22 stations, 2 FOVs, 32 MHz, 16 bit samples</td>
</tr>
<tr>
<td>Hardware specs</td>
<td></td>
</tr>
<tr>
<td>Input data rate</td>
<td>Sustained: 800 Gbps</td>
</tr>
<tr>
<td>Processing power</td>
<td>Input section: 0.4 TFlops</td>
</tr>
<tr>
<td></td>
<td>Blue Gene/C: 27 TFlops</td>
</tr>
<tr>
<td></td>
<td>Aux. processing: 0.2 TFlops + 3 TCMACs</td>
</tr>
<tr>
<td></td>
<td>Online processing: 1 TFlops</td>
</tr>
<tr>
<td>Storage</td>
<td>1.6 TB</td>
</tr>
<tr>
<td>Internal network</td>
<td>in clusters</td>
</tr>
</tbody>
</table>

Figure 5.11: Visualization of knowledge entities based on type.
5.4 Implementation of requirements in the Word Client tool

APPENDIX: ARCHITECTURAL KNOWLEDGE

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Provide portability</td>
<td>Decision</td>
<td>provides portability between CEP and local resources at universities.</td>
</tr>
<tr>
<td>2</td>
<td>Offline cluster standard</td>
<td>Decision</td>
<td>Offline cluster complies to common accepted and installed standards at universities</td>
</tr>
<tr>
<td></td>
<td>compliance</td>
<td>Topic</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Inspection of data products</td>
<td>Decision</td>
<td>Inspection of data products in running applications (pipelines)</td>
</tr>
<tr>
<td>4</td>
<td>No support for inspection</td>
<td>Decision</td>
<td>Data implemented in a limited way. The CEP frame architecture does allow for extensions of these features</td>
</tr>
<tr>
<td></td>
<td>of data products</td>
<td>Topic</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Observation modes not tested</td>
<td>Risk</td>
<td>Not all observation modes has been tested/prototyped yet.</td>
</tr>
</tbody>
</table>

Figure 5.12: A table of architectural knowledge entities contained in a document.

Figure 5.13: Annotations stored in comments.
if they are not already installed on the system.

WR13. Export annotated knowledge

The annotations stored in a document can be exported to an XML file. This is useful for sharing architectural knowledge with other applications.

WR14. Use Knowledge Architect API to store and retrieve knowledge entities.

Since the Word Client is a part of the Knowledge Architect system for capturing and using architectural knowledge, it must use the Knowledge Architect to store and retrieve KEs. The Knowledge Architect exposes its API through a web service API. The Word Client consumes this web service in several ways:

- Adding and editing knowledge entity when capturing architectural knowledge
- Retrieving domain model information
- Filling the repository with architectural knowledge from documents

To ensure that the Word Client is minimally impacted by changes in the Knowledge Architect API, it implements a facade through which API calls are done, see figure 5.15. When the API changes, only the facade needs to be updated. This facade also implements a collection of active knowledge entities in the document. This is described in more detail later.
5.5 Implementation

This section will go into implementation details for relevant decision topics and issues that were raised during development. For each decision topic, one or more alternatives will be presented, a rationale for the selected alternative, implications, assumptions, and any related decisions. The first part discusses decision topics related to the solution domain, while the second part discusses more general issues related to the quality of the solution.

The concerns are discussed with the most important, or basic constraints, discussed first. For each concern, the discussion first covers what the concern is, and then what decision was chosen.

5.5.1 Integration with existing tools

Any tool support should be implemented as an extension to existing tools in the software architect’s toolbox. Instead of introducing new programs for the software architect to use, a better approach would be to extend existing tools used by the software architect. See table 5.2 on page 66.

5.5.2 Offline use of documents with AK

To some extent, it should be possible to use the client program when being offline. Knowledge entities saved in the repository cannot be accessed when offline, and to enable distributability and offline use of documents, annotations should also be saved as metadata in the documents. When loading the document, the client program reconstructs the contained knowledge entities such that the user have access to a local subset of knowledge entities. See table 5.3 on page 66.
**Integrate with existing tools**

**Quick Decision**

The tool should be an extension to Microsoft Word.

**Rationale**

Word is the most used tool by software architects, and most documentation is already done in Word. Word can be extended with plugins and has an extensive API that gives programmatic access to all functionality. This allows taking advantage of built-in features in Word (for data storage etc).

Software architects should not be forced to use a new tool as this will make them resistant to using it. A new tool requires both time spent in learning to use it, and a context switch that will interrupt the software architect when using it. A solution should therefore be an extension to an existing tool. This also raises the possibility that functionality of the existing tool can be reused and exploited in the solution.

**Implications**

A plugin can use existing functionality in the tool. This can leverage plugin functionality, make development easier, and ease adoption of use. This decision leads to another decision of using built-in functionality when present.

Table 5.2: Decision concerning integration with existing tools

<table>
<thead>
<tr>
<th>Alternative 1</th>
<th>The user only has read-only access to knowledge entities/annotations when offline. No synchronization issues arise, but the user cannot add or change knowledge entities.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative 2</td>
<td>The user can add and edit knowledge entities/annotations, and must synchronize these with the repository when online. There is a risk of conflicting data, since the data in the repository could have changed when the user was offline.</td>
</tr>
<tr>
<td>Decision</td>
<td>Alternative 2 is chosen. The client program should be able to use knowledge entities/annotations in offline settings. When the user is online, he must import/synchronize changed data.</td>
</tr>
<tr>
<td>Rationale</td>
<td>AK should be usable when offline. This is useful during review of architectural documents, which could be done during traveling. It requires more effort to implement this solution compared to the first alternative, but the usability of the tool will be increased.</td>
</tr>
<tr>
<td>Implications</td>
<td>This decision raises the following concerns: synchronization of architectural knowledge, storage of metadata in documents, loading documents. To be able to create and edit architectural knowledge when offline, a mechanism for synchronizing or importing knowledge entities from documents to the repository is useful. An import function is especially useful during development of the repository and client program, since the repository will likely be emptied often.</td>
</tr>
<tr>
<td>Related</td>
<td>distributability, synchronization, metadata storage.</td>
</tr>
</tbody>
</table>

Table 5.3: Decision concerning how to support offline use of documents
5.5 Implementation

5.5.3 Loading of documents is dependent on access/situation

A document contains metadata for reconstructing annotations. Annotations internally refer to a knowledge entity with an id number. The annotations in a document are loaded differently dependent on repository access. When the user is offline, the Knowledge Architect repository cannot be accessed to retrieve necessary architectural knowledge entities. See table 5.4.

<table>
<thead>
<tr>
<th>Loading of documents is dependent on access in situation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick Decision</td>
</tr>
<tr>
<td>If the user is online, the referenced knowledge entities in annotations are loaded from the repository to make sure the latest version is used. If the user is offline, annotations are reconstructed from metadata saved in the document. When loading a document, the AK entities in the document are used as follows. The ID numbers saved in the document are used to look up the entities in the Knowledge Architect repository to ensure that the newest version is used. When the user is offline, document-local annotations can be reconstructed from metadata saved in the document. The document contains information on all aspects of an AK entity in the case of offline use, but the authoritative AK is always considered to be on the server.</td>
</tr>
<tr>
<td>Rationale</td>
</tr>
<tr>
<td>A user should always have access to the annotated architectural knowledge in documents.</td>
</tr>
<tr>
<td>Implications</td>
</tr>
<tr>
<td>The user must synchronize any documents that were updated when being offline with the Knowledge Architect backend.</td>
</tr>
</tbody>
</table>

Table 5.4: Decision concerning how to load KEs from documents

5.5.4 Storage of metadata in documents

Documents with annotations must be distributable, such that they can be used on machines where the Word Client is not installed. This is different to offline use, since the KEs cannot be seen. However, the comments can be read since they are saved in XML format.

<table>
<thead>
<tr>
<th>Function</th>
<th>Comment</th>
<th>Bookmark</th>
<th>Endnote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save marked text in XML format</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Location reference in document</td>
<td>+</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Short name for AK entity</td>
<td>−</td>
<td>+</td>
<td>−</td>
</tr>
</tbody>
</table>

Table 5.5: Built-in possibilities for storing annotations in Word

Information on knowledge entities must be saved within the document such that the annotations created with the Word Client can be used on setups without the Word Client installed. It should be possible to save annotation in XML format, create references to annotations, and give annotations a short name. Built-in functionality
in Word should be used when possible. Table 5.5 shows which requirements are supported by the built-in Word concepts: notes, bookmarks, and comments. The decision is shown in table 5.6.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Storage of metadata in documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store data containing the following: metadata encoded as XML data, a location of the annotation in the document to enable navigating and jumping to annotations in documents. Metadata about annotations must be stored in documents. Word has several methods to store data.</td>
<td></td>
</tr>
<tr>
<td>Alternative 1</td>
<td>Variables can be saved in documents.</td>
</tr>
<tr>
<td>Alternative 2</td>
<td>Use internal structures. The three main ways to add notes and links to a document are foot notes, bookmarks and comments.</td>
</tr>
<tr>
<td>Decision</td>
<td>Metadata about annotations must be stored in documents.</td>
</tr>
<tr>
<td>Rationale</td>
<td>Word has different ways to embed metadata in documents. Data can be saved in foot notes or in comments. By using built-in functionality, like using comments to store AK information, some extra benefits are realized. Comments can be filtered, such that comments created by the Knowledge Architect Word Client are not shown, or only annotations created by specific authors are shown. Another possibility would be to see which authors have created the annotations using Word’s tracking ability. Bookmarks are not sufficient for saving enough metadata, and footnotes cannot be referenced to a location. Metadata can be saved as internal variables, but have special side effects. Comments can be referenced by location, and this is required such that traceability to fragments of architectural knowledge in documentation can be supported. The simplicity and versatility of comments makes them superior for this purpose.</td>
</tr>
<tr>
<td>Related</td>
<td>offline use, distributability.</td>
</tr>
</tbody>
</table>

Table 5.6: Decision concerning storage of metadata in documents

5.5.5 Use of different domain models

Different organizations use different concepts for architectural knowledge, since there is no standardized domain model and some are dependent on legacy documentation. Furthermore, the concepts and relations in the domain model used in the case study are not stable and are subject to change. The client program should be able to load and use different domain models. The architectural knowledge entities in the repository are organized according to the domain model in use, such that the client program is dependent on the repository. See table 5.7 on page 69.
Use of different domain models

<table>
<thead>
<tr>
<th>Quick Decision</th>
<th>The domain model in use can be requested from the Knowledge Architect. The client program should be able to use other domain models, and can request information on the domain model used from the Knowledge Architect.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rationale</td>
<td>The client program is more flexible when it is independent of the domain model used, and can be used in organizations that use other concepts of architectural knowledge. More generally, the client tool can be used to create a network of annotated knowledge in any domain. In the case study, one domain model is used. This domain model is not yet fixed and chances are great that it will be changed again.</td>
</tr>
<tr>
<td>Assumptions</td>
<td>The Knowledge Architect repository must be available when requesting information on the domain model.</td>
</tr>
<tr>
<td>Implications</td>
<td>When the client program is used in off-line mode, it should still be possible to reconstruct knowledge entities from the document. A domain model should be used and traceability should still be supported. Completeness check, however, cannot be supported in offline mode.</td>
</tr>
<tr>
<td>Related</td>
<td>Offline use of tool.</td>
</tr>
</tbody>
</table>

Table 5.7: Decision about supporting use of different domain models

5.5.6 Network performance

Architectural knowledge resides in a repository which is accessed through a web service API. This creates a concern about the influence of network performance on usability of the client program. When the client program needs to access architectural knowledge, the user should not experience excessive delays or waiting time. The performance of the repository is the repository’s responsibility, but it cannot take slow or digested network connections into account. Therefore, the client program should implement a mechanism for avoiding this problem.

One solution is to limit the amount and frequency of accesses to the repository. See table 5.8 and 5.9 below.
A cache to hold the currently used knowledge entities would avoid re-requesting the repository for the same knowledge entities. Policies for how the cache operates must be decided upon, and applies to read and write operations. The cache caches knowledge entities and any request to load these must go through the cache. It looks up the requested knowledge entities and in the case of a hit, it returns them. In the case of a cache miss, the knowledge entities are loaded from the repository, and the cache is updated. Any additions or changes to knowledge entities should be propagated to the repository immediately to prevent conflicting data. A write-thru policy ensures that the repository and cache are kept up-to-date. In the case of changes to a KE, a validity check of the cached KE should be done prior to writing changes. If the validity check fails, which can happen when two users are changing the same KE simultaneously, the cache is updated. This is a trade-off between usability and performance on one side, and concurrency on the other side.

Implementation: The cache is implemented as a layer which all access to the repository must go through. This has several benefits: The cache proxy is a facade for the repository, and can present a consistent interface to the client program. It introduces loose coupling between the client program and the repository, such that the client program is not affected by API changes in the repository web service, which improves maintainability. This is important since the repository API is subject to change. Usability is improved.

<table>
<thead>
<tr>
<th>Decision</th>
<th>Network performance</th>
</tr>
</thead>
</table>

Table 5.8: Decision concerning network performance. Table is continued below.
<table>
<thead>
<tr>
<th>Network performance cont’d</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rationale</strong></td>
</tr>
<tr>
<td><strong>Assumptions</strong></td>
</tr>
<tr>
<td><strong>Implications</strong></td>
</tr>
<tr>
<td><strong>Related</strong></td>
</tr>
</tbody>
</table>

Table 5.9: Decision about solving network performance problems
5.6 Future Work

This section discusses possible future enhancements to Knowledge Architect in general and the Word Client in particular. Most major changes require changes to the Knowledge Architect as well as to the Word Client. Indeed, some of this functionality was not implemented in the Word Client, because it was not yet supported in the Knowledge Architect.

The most obvious changes to do would be to give the tool a more finished touch. This involves functionality such as configuration of visualization colors, and which knowledge repository to use. In addition, a state for knowledge entities could be added, such that a discussion or maturity state can be indicated.

The Knowledge Architect should be extended to support multiple projects, such that architectural knowledge can be separated between projects and shared, where needed.

Traceability should be extended, such that it is possible to trace to knowledge entities located in other documents. This requires that the destination document and the location in the document can be retrieved. An implication of this is that the Knowledge Architect repository should be expanded to support whole documents besides architectural knowledge fragments.

A possible useful feature is version control of the Knowledge Architect repository. This can be used to show how architectural knowledge change over time.

Another feature to implement is grouping. Other approaches uses simplicity as argument for encapsulating AK concepts in a bigger concept, such that a 'complete decision' contains alternatives, concerns etc. This could be supported by adding grouping of concepts. A group could abstract a set of knowledge entities into e.g. subsystems. A group could contain all relevant information for a decision, enabling a simpler view on the documentation by exposing only groups. This is useful when the software architect’s work is focused within the constraints of a group, and for knowledge sharing. Viewing architectural decisions as groups will enable a high-level view of a software architecture, where users can focus on specific decision topics or quality attributes.

Specifically for the Knowledge Architect, more client tools could be built that support the capture and use of architectural knowledge in different ways. For example, clients for visualizing relationships in architectural knowledge would be beneficial to understandability of a software architecture.

Further research could delve into the area of configuration and variability modeling where decisions determine the configuration of a software architecture. [36] shows how this area is related to architectural decisions. Other research could combine the formal parts of software architecture description and rationale to incorporate architectural decision knowledge in formal descriptions of software architectures. Some research has already been done in this area, see [37].

This section has explored some possibilities for improving and extending the Knowledge Architect and Word Client system, and suggested directions for further research.
Chapter 6

Conclusion

This thesis described a way of introducing traceability in software architecture documentation. A proof of concept tool for capturing and tracing architectural knowledge was implemented. To reach the goal of effective traceability, a number of research questions were posed. The answers to these will be summarized below.

Why is architectural knowledge in documentation difficult to understand?
Some architectural knowledge is essential to understand documentation. Much of this architectural knowledge needed to understand documentation is not being documented due to a number of barriers. This was explained in section Barriers to documenting architectural knowledge on page 28.

Another reason is that architectural knowledge stays implicit in documentation. Reading the documentation requires significant contextual knowledge, since the types of architectural knowledge cannot be identified without. Furthermore is it difficult to document relationships between architectural knowledge in a structured and usable way, resulting in this knowledge often being implicit or lost. Furthermore, documentation is dense and difficult to read. See Identifying Architectural Knowledge on page 30.

What architectural knowledge is essential to understand documentation?
Knowledge of architectural decisions essential to understand documentation. Architectural decisions must be explicitly represented in documentation, because they bridge the gap between requirements, design, and rationale. Relationships between architectural knowledge are essential to understand documentation, as they enable traceability among this knowledge. Thus architectural decisions must be promoted to first-class citizens. See section First-class representation of Architectural Decisions on page 34.

How can architectural knowledge be represented as first-class citizens in documentation?
This question was answered in two different ways. First, templates for documenting architectural decisions showed how decisions can be promoted to first-class citizens. Second, A Model of Architectural Knowledge on page 42 presented a way of promoting several concepts of architectural knowledge to first-class citizens without the drawbacks of the first answer.
How can architectural knowledge relationships be represented in a scalable way?
This question was answered by the model of architectural knowledge on page 42. The model defines relationships between its concepts, and these are shown to be scalable and support highly intertwined relationships. This enables traceability and tool support for architectural knowledge. The section also introduced a domain model of architectural knowledge.

How can capturing of architectural knowledge be supported?
Capturing of architectural knowledge is supported by a hybrid approach of templates for capturing the knowledge and a domain model for storing it and capturing the relationships. An answer to this question is devised in the Solution chapter, see specifically section 5.1 on page 50 for a conceptual answer.

The main goal of this thesis was to enable traceability between architectural knowledge in software architecture documentation. This has been reached by implementing a tool that supports the following to enable capturing and tracing of architectural knowledge:

- Represent architectural knowledge as first-class citizens
- Support complex relationships between architectural knowledge
- Capture architectural knowledge including relationships
- Support traceability across relationships of architectural knowledge

The tool breaks the barriers for capturing architectural knowledge. It defines what knowledge must be documented, and makes it easy to capture it. This is done by integrating a domain model with a traditional documentation tool. This allows software architects to use their preferred tool for documentation. Essential architectural knowledge can easily be captured by annotating knowledge fragments and defining their type. The captured knowledge entities can be related in complex ways, such as defined by the domain model. A domain model defines what architectural knowledge should be captured, and the tool makes it easy to do this.

The tool also supports visualization of knowledge entities embedded in documentation. This is done directly in documentation using coloring based on type of architectural knowledge. This greatly enhances the overview of documentation. The created knowledge entities can be used in multiple contexts, and annotations enable access to them from all contexts. A reference list of architectural knowledge fragments can be created to support a better overview of documentation.

Traceability is supported in a straight-forward manner. Using the context menu with the mouse, the user can quickly see what relations a knowledge entity has. The user can easily navigate the document by tracing to related knowledge entities. This enables the user to find related architectural knowledge without needing to have an overview of the complete documentation, and highly improves understandability of the documentation.

The software architect is also supported in another way. A completeness check can help assessing the design maturity of the software architecture and the completeness of documentation. A completeness check evaluates structural rules in the
relationships of knowledge entities. This can be visualized in a similar way mentioned above using colors. Error messages can also be shown for each knowledge entity.

In conclusion, the tool helps the software architect capture essential architectural knowledge, enables traceability, and improves visualization to improve the understanding of software architecture documentation.
Bibliography


List of Tables

5.1 Requirements for the Word Client .................................................. 53
5.2 Decision concerning integration with existing tools .......................... 66
5.3 Decision concerning how to support offline use of documents .......... 66
5.4 Decision concerning how to load KEs from documents ....................... 67
5.5 Built-in possibilities for storing annotations in Word ....................... 67
5.6 Decision concerning storage of metadata in documents ..................... 68
5.7 Decision about supporting use of different domain models ............... 69
5.8 Decision concerning network performance. Table is continued below ... 70
5.9 Decision about solving network performance problems .................... 71
## List of Figures

1.1 Areas of software architecture research ........................................ 2
3.1 The four parts of the GRIFFIN Core Model of Architectural Knowledge. 12
3.2 GRIFFIN domain model for architectural knowledge ........................ 14
3.3 Subset of domain model describing Architectural Design Decisions .... 15
3.4 A decision changing the software architecture ............................... 19
4.1 Decisions bridge the gap between requirements and design, enabling traceability between architectural knowledge. ......................... 35
4.2 A tree of decisions narrowing in scope ...................................... 37
4.3 Decisions can be influenced by cross-cutting, but seemingly unrelated decisions ............................................................... 38
4.4 (a) describes how templates document relations, (b) describes two-way n-to-n relations .......................................................... 39
4.5 The case study domain model of architectural knowledge. .............. 45
5.1 Knowledge Architect system architecture ..................................... 52
5.2 Word Client toolbar .................................................................... 54
5.3 Creating annotation of a fragment of architectural knowledge ........ 55
5.4 Using the context menu to annotate a fragment of architectural knowl-
edge ......................................................................................... 55
5.5 Editing existing annotation .......................................................... 56
5.6 Selecting relation type ............................................................... 57
5.7 Adding relations to other knowledge entities ................................. 58
5.8 Visualization of completeness check ............................................ 60
5.9 Completeness coloring and error message .................................... 60
5.10 Tracing from a knowledge entity to other ..................................... 61
5.11 Visualization of knowledge entities based on type ......................... 62
5.12 A table of architectural knowledge entities contained in a document .. 63
5.13 Annotations stored in comments .................................................. 63
5.14 Defining a domain model in Protege ........................................... 64
5.15 Word Client uses Knowledge Architect API through a facade layer .. 65