DISSERTATION

submitted

to the

Combined Faculty for the Natural Sciences and Mathematics

of

Heidelberg University, Germany

for the degree of

Doctor of Natural Science

Put forward by

M.Sc. Tom-Michael Hesse

Born in Leipzig

Oral examination: ....

Supporting Software Development by an Integrated Documentation Model for Decisions

Advisors: Prof. Dr. Barbara Paech
Prof. Bernd Bruegge, Ph.D.

Abstract

Decision-making is a vital activity during software development. Decisions made during requirements engineering, software design, and implementation guide the development process. In order to make decisions, developers may apply different strategies. For instance, they can search for alternatives and evaluate them according to given criteria, or they may rely on their personal experience and heuristics to make single solution claims. Thereby, knowledge emerges during the process of decision making, as the content, outcome, and context of decisions are explored by developers. For instance, different solution options may be considered to address a given decision problem. In particular, such knowledge is growing rapidly, when multiple developers are involved. Therefore, it should be documented to make decisions comprehensible in the future.

However, this documentation is often not performed by developers in practice. First, developers need to find and use a documentation approach, which provides support for the decision making strategies applied for the decision to be documented. Thus, documentation approaches are required to support multiple strategies. Second, due to the collaborative nature of the decision making process during one or more development activities, decision knowledge needs to be captured and structured according to one integrated model, which can be applied during all these development activities.

This thesis uncovers two important reasons, why the aforementioned requirements are currently not fulfilled sufficiently. First, it is investigated, which decision making strategies can be identified in the documentation of decisions within issue tickets from the Firefox project. Interestingly, most documented decision knowledge originates from naturalistic decision making, whereas most current documentation approaches structure the captured knowledge according to rational decision making strategies. Second, most decision documentation approaches focus on one development activity, so that for instance decision documentation during requirements engineering and implementation are not supported within the same documentation model.

The main contribution of this thesis is a documentation model for decision knowledge, which addresses these two findings. In detail, the documentation model supports the documentation of decision knowledge resulting from both naturalistic and rational decision making strategies, and integrates this knowledge within flexible documentation structures. Also, it is suitable for capturing decision knowledge during the three development activities of requirements engineering, design, and implementation. Furthermore, a tool support is presented for the model, which allows developers to integrate decision capturing and documentation in their activities using the Eclipse IDE.

Zusammenfassung

Eine wichtige Aktivität im Softwareentwicklungsprozess ist das Treffen von Entscheidungen. Entscheidungen etwa beim Erheben von Anforderungen, dem Softwareentwurf oder der Implementierung steuern den gesamten Entwicklungsprozess. Um Entscheidungen zu treffen, nutzen Softwareentwickler unterschiedliche Strategien. Beispielsweise können sie nach Lösungsalternativen suchen und diese anhand gegebener Kriterien analysieren, oder sie orientieren sich an ihrer Erfahrung und Heuristiken, um sich ohne weitere Abwägung für eine Lösung zu entscheiden. Während dieses Prozesses entstehen große Mengen an Wissen zu Inhalt, Ergebnissen und Kontext von Entscheidungen. Die Menge dieses Wissens kann rapide ansteigen, wenn mehrere Entwickler an einer Entscheidung beteiligt sind - etwa durch die Diskussion zu unterschiedlichen Lösungsmöglichkeiten eines Entscheidungsproblems. Damit die getroffenen Entscheidungen in Zukunft nachvollziehbar bleiben, sollte Entscheidungswissen deshalb dokumentiert werden.

Trotzdem wird die Dokumentation von Entscheidungswissen in der Praxis oft nicht durchgeführt, da Softwareentwickler zunächst einen Dokumentationsansatz für Entscheidungswissen finden und anwenden müssen, der die von ihnen verwendete Entscheidungsstrategie unterstützt. Das erfordert Dokumentationsverfahren, die möglichst verschiedene Strategien zum Treffen von Entscheidungen unterstützen. Außerdem entsteht durch die Zusammenarbeit von mehreren Entwicklern während des Entscheidungsprozesses über verschiedene Entwicklungsaktivitäten hinweg Entscheidungswissen, das in einem integrierten Modell dokumentiert werden sollte, welches in allen Entwicklungsaktivitäten angewendet werden kann.

Die vorliegende Arbeit untersucht zwei wesentliche Gründe, warum die beiden vorgenannten Anforderungen derzeit nicht hinreichend erfüllt werden. Erstens wird untersucht, welche Strategien zum Treffen von Entscheidungen in bereits dokumentierten Entscheidungen aus Issue-Tickets des Firefox-Projektes identifiziert werden können. Dabei zeigt sich, dass das meiste dokumentierte Wissen aus naturalistischen Entscheidungen stammt, obwohl viele Dokumentationsverfahren sich auf Wissen aus rationalem Entscheiden fokussieren. Zweitens wenden sich die meisten bestehenden Dokumentationsverfahren einer Entwicklungsaktivität zu, sodass beispielsweise Entscheidungen zu Anforderungsmanagement und Implementierung nicht im selben Dokumentationsmodell vorgehalten werden können.

Der wichtigste Beitrag der vorliegenden Arbeit ist ein Dokumentationsmodell für Entscheidungswissen, welches diese beiden Gründe für die bisher selten explizit stattfindende Dokumentation von Entscheidungswissen aufgreift. Dieses Modell unterstützt insbesondere die Dokumentation von Entscheidungswissen, welches sowohl aus rationalem als auch aus naturalistischem Entscheiden entstanden sein kann. Dieses Wissen wird in flexiblen Dokumentationstrukturen integriert. Darüber hinaus ist das Modell in der Lage, Entscheidungswissen in den drei verschiedenen Entwicklungsaktivitäten Anforderungsmanagement, Softwareentwurf sowie Implementierung zu erfassen. Dafür wird eine Werkzeugunterstützung für das Modell entwickelt und vorgestellt, welche es Entwicklern parallel zu ihren Tätigkeiten ermöglicht, Entscheidungswissen innerhalb der Entwicklungsumgebung Eclipse zu erfassen und zu dokumentieren.

Acknowledgements

First and foremost, I thank my supervisor Prof. Barbara Paech for her enduring support and advise. She taught me her systematic and comprehensive way of working on scientific problems. I learned from her to unfold the full complexity of a problem in multiple iterations over time, and to review the related solution options carefully. With this idea, she also gave fundamental direction to my solution approach for documenting decisions.

Furthermore, I would like to thank Prof. Bernd Bruegge, PhD., for a very successful and inspiring cooperation within the URES project. He provided invaluable feedback on our project work and my thesis, which helped me to improve my results significantly. Also, I enjoyed working with Tobias Roehm, with whom I had many insightful discussions about my research.

Great thanks deserve all members of the Software Engineering Group at Heidelberg University for a very constructive atmosphere and a lot of valuable feedback on my ideas. Moreover, many students supported my research by their theses and by working with me as student assistants. In particular, I want to thank Arthur Kuehlwein for supporting me with the creation of DecDoc.

Finally, I deeply thank my parents and Wiebke for supporting me and believing in me. They helped me to overcome the most urgent obstacle in finishing this thesis — me, when being doubtful or distracted.

Contents

List of Figures

List of Tables

List of Abbreviations

DKE	Decision Knowledge Element
DM	Decision Making
DRL	Decision Representation Language
GQM	Goal Question Metric
IDE	Integrated Development Environment
NDM	Naturalistic Decision Making
QOC	Questions, Options and Criteria
RDM	Rational Decision Making
RQ	Research Question
UML	Unified Modeling Language

Part I

Preliminaries

1 Introduction

This chapter introduces the domain of decision documentation, its current shortcomings and the contributions of this thesis to improve documentation by developers. First, the need for reflecting the actual decision making strategies and multiple development activities within decision documentation is motivated. Also, the research goals resulting from this need are defined and justified. Second, the major contributions of this thesis are outlined to illustrate how the research goals are investigated. This is the foundation for the structure of this thesis, which is presented in the third section. Finally, a list of previous publications is presented to describe the findings and results published beforehand.

1.1 Motivation and Research Goals

Decision making is a crucial activity within the process of software engineering [Ruhe 2003]. The decisions made by developers in various software engineering activities give direction to the development process and heavily impact its outcome [Davide Falessi, Cantone, and Becker 2006]. For instance, important decisions are made regarding the realization of architecturally significant requirements during *requirements engineering* [Chen, Ali Babar, and Nuseibeh 2013], on the structure of the system's architecture during software design [Jansen and Bosch 2005], and on the optimal coding for particular functionality during *implementation* [Lougher and Rodden 1993; Canfora, Casazza, and De Lucia 2000]. It should be noted, that the terms "software" and "system" are used synonymous in this thesis.

Two major kinds of strategies for decision making (abbreviated as *DM*) can be distinguished: *Rational decsion making* (abbreviated as *RDM*) and *naturalistic decision making* (abbreviated as *NDM*). These strategies consist of different possible actions, which can be executed by the developers in order to determine a decision according to the respective strategy. Such actions are called *strategy elements*.

Developers follow an RDM approach when they perform a detailed analysis of the decision situation and explore all available options according to defined criteria [Zannier, Chiasson, and Maurer 2007]. In contrast, developers may also rely on their personal knowledge and experience from former decisions to make a solution claim without a detailed investigation of different options [Lipshitz et al. 2001]. Then, they apply an NDM approach.

During decision making, developers acquire and evaluate knowledge related to their decisions. In the remainder of this thesis, all knowledge related to a decision is called *decision knowledge*. Within decision knowledge, different parts of knowledge focusing on a certain topic can be distinguished. For instance, developers explore the current decision problem, alternatives for solving the problem, rationales justifying their choice, and additional information on the decision context, such as constraints for the decision [Tyree and Akerman 2005]. These parts of decision knowledge are called *decision knowledge elements* (abbreviated as *knowledge elements*). For developers, access to and exploitation of decision knowledge with its elements has great importance. A major reason is that developers need to comprehend and review their decisions in order to understand and improve many software engineering artifacts over time. This is highlighted by Jansen and Bosch when they describe the software architecture of a system as “a set of design decisions” [Jansen and Bosch 2005]. In consequence, knowledge management for decision making during software engineering should be supported [Zannier, Chiasson, and Maurer 2007], which is one of multiple areas addressed by *decision support systems*. In general, decision support systems are concerned with supporting and documenting both decision making strategies and decision knowledge. For instance, they can be realized as *negotiation support systems* to support group decision making or as *knowledge management-based systems* to support knowledge transfer, storage and retrieval [Arnott and Pervan 2008]. Because a system for *decision documentation support* is a knowledge management-based system for decision support, it captures and structures decision knowledge based on a decision knowledge model. Then, the term *documentation model* is used synonymous for knowledge model. A summary of the relationships between decision support systems, decision documentation, decision knowledge and decision making strategies is given in Figure 1.1.

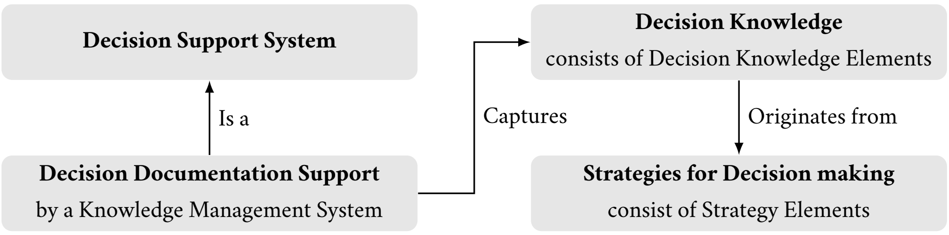

Decision Support System
Decision Knowledge
consists of Decision Knowledge Elements
Is a
Captures
Originates from
Decision Documentation Support
by a Knowledge Management System
Strategies for Decision making
consist of Strategy Elements

Figure 1.1: Relationships between Decision Support Systems, Decision Documentation, Decision Knowledge and DM Strategies

In particular, developers require knowledge management-based systems to document their decision knowledge with respect to their individual documentation preferences and needs [Davide Falessi, Cantone, and Becker 2006; Tang, Ali Babar, et al. 2006; Manteuffel, Tofan, Koziolek, et al. 2014]. Otherwise, decision knowledge might erode and may be lost completely over time [Jansen and Bosch 2005], as teams may change and decisions may be adapted. Actually, in practice, decision knowledge often remains implicit and undocumented. The survey of Tang, Ali Babar, et al. emphasizes this observation: Only 35.8% out of 81 professional software designers documented potential shortcomings of their design decisions, and only 39.5% captured the reasons why their solution was implementable [Tang, Ali Babar, et al. 2006]. Then, most information needs towards the decisions made remain unsatisfied, as a study of Ko, DeLine, and Venolia at Microsoft with 17 developer teams showed: In 44% of all cases the question "Why was the code implemented this way?" could not be answered [Ko, DeLine, and Venolia 2007].

The findings of the studies of Tang, Ali Babar, et al. and Ko, DeLine, and Venolia are surprising, as already many approaches exist to support developers with capturing and managing decision knowledge. For requirements engineering, Ngo and Ruhe and Aurum, Wohlin, and Porter describe different approaches for capturing decisions on prioritizing requirements [Ngo and Ruhe 2005; Aurum, Wohlin, and Porter 2006]. Various approaches exist for documenting design decisions using text templates and knowledge models, as investigated by the comparative studies of Ali Babar, Boer, et al. [Ali Babar, Boer, et al. 2007] and Tang, Avgeriou, et al. [Tang, Avgeriou, et al. 2010]. Also during implementation, different support mechanisms are proposed by researchers in order to make decisions explicit, such as markup languages for decisions by Lougher and Rodden [Lougher and Rodden 1993] and links between code and decision knowledge proposed by Canfora, Casazza, and De Lucia [Canfora, Casazza, and De Lucia 2000] or Burge and Brown [Burge and Brown 2008]. These approaches already cover a broad range of different methods for capturing, structuring and accessing decision knowledge. However, the study findings of Tang, Ali Babar, et al. and Ko, DeLine, and Venolia are not satisfactory, as many information needs related to decisions cannot be fulfilled in practice due to unavailable knowledge. Thus, further improvements in decision documentation support are required in order to supply developers with explicit and structured decision knowledge.

These improvements should address two practice problems, which decrease the acceptance and application of current decision documentation approaches [Manteuffel, Tofan, Koziolek, et al. 2014]. First, current documentation approaches mostly rely on codifying knowledge based on RDM, instead of also considering knowledge resulting from NDM decisions. Second, documentation is typically focused on particular development activities, instead of integrating decision knowledge across related activities. An overview of these practice problems is given in Figure 1.2. In the upper half, it depicts three developers. Each of them is concerned with a different development activity. In addition, all developers apply individual mixes of decision making strategies. For instance, the developer performing the implementation mostly relies on NDM, whereas the designer mixes both, RDM and NDM. This illustrates that documentation support needs to address decision knowledge resulting from mixed decision making strategies containing both RDM and NDM elements. Over time and during subsequent development activities, developers contribute various decision knowledge elements as a result of their decision making process. This is shown in the lower half of the figure for all considered development activities. As an example, consider an alternative proposed by the requirements engineer in development iteration 2. This alternative is further refined by an implication identified by the designer in development iteration 3. Thus, documentation support also needs to consider that decision knowledge results from an incremental decision making process, which intertwines related development activities.

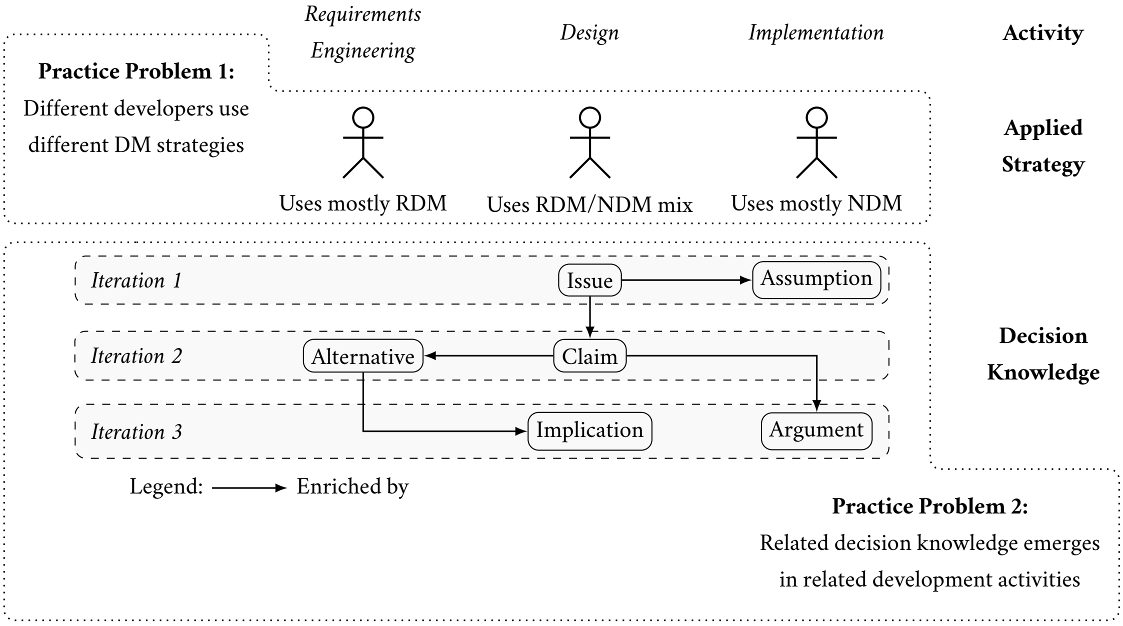


Figure 1.2: Practice Problems Addressed in this Thesis

The practice problems addressed by this thesis are introduced in more detail in the following paragraphs. For each practice problem, a corresponding research goal is described with several open questions. This thesis will contribute findings for these open questions to the respective research goal. Based on these findings, solution approaches will be proposed in order to address the practice problems.

Practice Problem 1: Different Developers Use Different DM Strategies

The origin of the first practice problem is that both RDM and NDM strategies may be applied by developers in practice [Zannier, Chiasson, and Maurer 2007; Zannier and Maurer 2006]. According to the study findings of Zannier, Chiasson, and Maurer, developers may even mix both strategies within the same decision. This impacts the methods and tools required for supporting decision documentation, as the applied decision making strategy should be reflected within the knowledge management enabled by decision documentation [Maule 2010]. However, Ali Babar, Boer, et al. discuss a gap between the intentional codification of decision knowledge in research approaches and the unintentional personalization of decision knowledge in practice [Ali Babar, Boer, et al. 2007]. On the one hand, most documentation approaches for decisions in research rely on intentional and structured documentation of knowledge by developers. For instance, Davide Falessi, Cantone, Kazman, et al. have claimed that developers may only use RDM for design decisions, as RDM is required to make reasonable decisions within the engineering discipline of software design [Davide Falessi, Cantone, Kazman, et al. 2011]. Thus, research approaches typically presume developers to apply an exhaustive rational decision making strategy. On the other hand, in practice, developers also apply NDM strategies, or mix both RDM and NDM [Zannier, Chiasson, and Maurer 2007]. Then, they can use their personal experiences and individual knowledge for making decisions. This allows for making efficient decisions considering the real-world time and resource restrictions of development projects [Klein 2008]. However, such decision making and its related personalized knowledge typically are not taken into account by current decision documentation approaches, as these approaches focus on RDM strategies.

Research Goal 1: Investigate Decision Documentation for Mixed DM Strategies

Recent qualitative studies (cf. [Tang and Vliet 2015; Tang, Aleti, et al. 2010; Zannier, Chiasson, and Maurer 2007]) indicate that NDM strategies significantly contribute to the decision making of developers in practice. But it is currently unknown, how NDM in general and a mix of decision making strategies in particular should be supported during decision documentation [Zannier, Chiasson, and Maurer 2007]. Thus, the existing qualitative studies need to be complemented with quantitative findings on the the mix of RDM and NDM in practice. First, it is not clear which quantity of NDM is actually used and documented by developers. Second, it is unknown which strategy elements of NDM are reflected by developers within their decision documentation, and how these elements are related to RDM strategy elements. Third, it is unknown whether specific distributions for the percentage of NDM exist according to specific kinds of decision situations. Findings on these open questions will provide further insights on the overall percentages of DM strategies documented, and on the distribution of particular strategy elements documented for decisions. Based on these quantitative insights, documentation support for decisions can be improved in order to reflect NDM within decision documentation appropriately. In detail, the quantity of NDM used and documented is needed to ground and specify a flexible decision documentation, which considers both RDM and NDM. In addition, suitable documentation structures and an appropriate support for NDM in relation to RDM can be identified according to the distribution of their strategy elements. Moreover, knowledge about characteristics of this distribution is required to determine any specific challenges and topics decision documentation should address to increase its value for developers. All these open questions contribute to the first research goal addressed by this thesis: *Investigate documentation support for decision knowledge resulting from the mixed use of NDM and RDM.*

Practice Problem 2: Decision Knowledge Emerges over Time
The second practice problem concerns the missing support for decision documentation across different development activities during the software engineering process. Current approaches for decision documentation typically focus on one development activity within this process. Most existing approaches primarily address knowledge on design decisions (cf. the studies of [Tang, Ali Babar, et al. 2006] and [Ali Babar, Boer, et al. 2007]). These approaches consider decisions and results of other development activities mainly by providing relations between them. For instance, links are provided for connecting design decisions with related requirements [Jansen and Bosch 2005; Capilla, Nava, Pérez, et al. 2006; Tang, Jin, and Han 2007] or code files [Canfora, Casazza, and De Lucia 2000; Burge and Brown 2008], or specific textual attributes are given to describe related knowledge [Tyree and Akerman 2005; Capilla, Nava, and Duenas 2007]. In addition, a few approaches also focus on decision documentation during requirements engineering [Ngo and Ruhe 2005; Aurum, Wohlin, and Porter 2006] or implementation [Lougher and Rodden 1993].

However, decision making during software engineering is a collaborative and incremental process, which spans across the individual development activities of requirements engineering, design and implementation. For instance, developers work together during all these activities to determine and realize design decisions on architecturally significant requirements [Nuseibeh 2001]. Then, a decision originating from requirements can be also subject to refinements and adaptions within design and implementation. An example for such a process is depicted in Figure 1.2. Here, an alternative is contributed to a given decision problem in the second development iteration during requirements engineering. This alternative is further explained by an argument identified during design in the third development iteration. In consequence, the involved developers need to access and edit all decision knowledge concerning a decision within different activities [Tang, Aleti, et al. 2010]. In particular, this is important to support developers working at different locations [Rekha and Muccini 2014]. As a consequence, not all decision knowledge is available at once, as developers are collaborating during decision making with multiple discussions and actions. Also, previous decisions may be revised by developers in follow-up development iterations [Ko and Chilana 2011]. Furthermore, different developers may stick to different decision making strategies, so that the available decision knowledge varies depending on the applied strategy. Then, there is no complete set of decision knowledge from the beginning, but the knowledge grows over time. In consequence, it is important to bundle all decision knowledge in a knowledge model combining different development activities and aggregating incrementally growing knowledge for each decision from all available sources.

Research Goal 2: Investigate Decision Documentation during Different Development Activities

As many current approaches already address decision documentation during development activities, it is known in principle how decisions can be documented for RDM appropriately. However, it is complicated to address the aforementioned documentation needs originating from a collaborative and incremental decision making process within only one activity, such as design [Tang, Aleti, et al. 2010]. Thus, it is even more challenging to provide documentation support across different development activities. First, it is not clear which knowledge elements should be contained in a core set for documentation during requirements engineering, design and implementation. This insight is required to create a collaborative knowledge model for decisions integrating the knowledge documentation of these activities. Second, it needs to be investigated which specific capturing mechanisms for decision knowledge should be provided during each development activity. Insights on the integration and adaption of these capturing mechanisms are necessary to capture decision knowledge incrementally within each activity. These two open questions contribute to the second research goal addressed by this thesis: *How to support the integrated decision knowledge documentation during closely related development activities?*

1.2 Research Methodology

The overall goal of this thesis is to enhance the documentation of decision knowledge during software engineering activities to increase the amount of captured and accessible decision knowledge within software development projects. To achieve this goal, it is necessary to investigate the described research goals with their current open questions. Based on the findings from these investigations, an integrated knowledge model for decision knowledge and its respective tool support are created. As research methodology, this thesis applies a design science approach. In general, design science is concerned with "design and investigation of artifacts in context" [Wieringa 2014, p. 3]. Therefore, *engineering cycles* are performed to investigate and treat a given design problem, so that the treatment can be implemented in practice [Wieringa 2014]. The actual investigation, creation, and validation of the treatment is performed in *design cycles*. The entire process is depicted in Figure 1.3. It should be noted that design science research only addresses design cycles, as the implementation of the treatment is a technology transfer to practice [Wieringa 2014].

Problem Investigation

The first action during the engineering and design cycle is to investigate the *design problem*. Such problems "call for a change in the real world" [Wieringa 2014, p. 4], and are addressed by design solutions. They may result in multiple *knowledge questions*, which ask for knowledge about specific

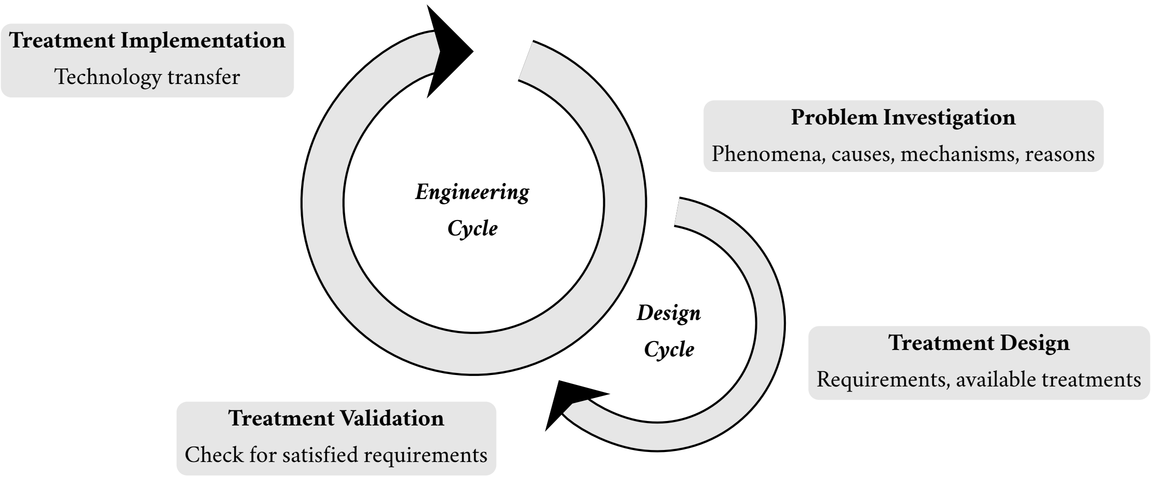


Figure 1.3: Overview of Engineering and Design Cycle according to [Wieringa 2014]

phenoma, causes, mechanisms, or reasons within the real world [Wieringa 2014]. For this thesis, the aforementioned overall goal describes the design problem to be addressed within the engineering cycle. Two open questions were described in Section 1.1, which represent important knowledge questions regarding this design problem. They are investigated by performing an observational case study on decision making behavior of developers in practice and a survey as a literature review on existing literature regarding decision documentation approaches.

Treatment Design

The treatment design represents the development and refinement of one or more solutions to the addressed design problem. In detail, currently available treatments have to be analyzed, and requirements for the newly design need to be specified [Wieringa 2014]. Currently available approaches for decision documentation are examined and compared within the literature review. Both the observational case study and the literature review are then used to identify and specify requirements for the decision documentation approach developed in this thesis. When creating the documentation approach and its tool support during the design cycle, the contribution of each requirement to the overall goals and its corresponding research goals is highlighted.

Treatment Validation

The developed solution is evaluated during treatment validation. In particular, the actual effect of the solution and its capability to satisfy the specified requirements need to be assessed [Wieringa 2014]. In this thesis, the treatment validation is performed by applying the documentation approach and its tool support in individual single-case experiments. Whereas this experiment is performed by using the documentation with practice data, the tool support is demonstrated within a complex example case.

Treatment Implementation
Finally, if the developed solution has been found to be a suitable treatment for the addressed requirements, it is implemented in practice [Wieringa 2014]. This implementation typically requires a technology transfer from research projects to the market. Thus, it is not part of this thesis as a design science research project [Wieringa 2014].

1.3 Solution Approach and Contributions

Towards a solution of the overall goal, the thesis provides three fundamental contributions according to the described research methodology:

Studies on Open Questions
The thesis comprises two different studies in order to investigate the open questions described for each research goal. Therefore, each study refines one research goal to a more concrete goal for the respective study outcome. Then, both studies investigate different research questions, which address and further refine the open questions given in this introduction. The first study is an empirical investigation of documented decision making strategy elements in open-source software. Therefore, comments on 260 issue reports of the Firefox project are evaluated for decision documentation. The results of this study provide insights on how developers document different decision making strategies in practice. In particular, the documentation of NDM within the issue comments for different issue types is investigated. Thereby, the study addresses the open questions for research goal 1. The second study investigates current approaches for decision documentation during requirements engineering, design and implementation given in the literature. Therefore, approaches are investigated, which are concerned with capture, linkage and usage support for documented decision knowledge. The findings of this study describe how decision knowledge is currently documented during different development activities. The knowledge models and functionality of the related tool support are compared for similarities and differences. In particular, the knowledge elements and capturing mechanisms for decision knowledge of each approach are investigated. Thereby, the study addresses the open questions for research goal 2. Based on the investigation results of both studies, requirements are derived for the decision documentation model and its related tool support.

Documentation Model for Decisions
In this thesis, a documentation model is presented, which integrates decision documentation for multiple related development activities. The model is build upon the insights and results of the two performed studies and the resulting requirements. It integrates findings and documentation structures of previous documentation approaches, such as QOC [MacLean et al. 1991] and DRL [Lee 1991]. The model combines selected knowledge elements from current approaches in a flexible

manner, so that an expressive model is created, which avoids static and prescriptive structures. In detail, it supports documentation of RDM as well as of NDM in any mixed way. In addition, the model allows for adding knowledge elements incrementally by different developers in a collaborative manner. Therefore, the model provides different levels of knowledge element abstraction, so that given knowledge can be refined or extended by developers. Moreover, knowledge elements in the model can be linked in various ways to other knowledge elements and to development artifacts related to the knowledge element. Thus, findings of the studies are reflected and used to improve the support for decision documentation.

Tool Support for Decision Documentation Model

The thesis also contributes to the improvement of decision documentation in practice by providing an individual tool support for the presented documentation model. The tool support not only makes model instances editable, but also allows for capturing decision knowledge semi-automatically during different development activities, because the tool support is integrated with Eclipse. It enables developers to import decision knowledge during requirements engineering. Therefore, knowledge from a security analysis of documented use cases can be imported and used to propose security-related decision knowledge. Furthermore, decisions can be captured during UML design with direct links to any related UML entities. Moreover, developers are enabled to document decisions during implementation by adding annotations containing the decision knowledge to their code. These features address the findings for research goal 2. Thereby, they help developers to document their decision knowledge as soon as it emerges during the respective development activity.

1.4 Structure of the Thesis

The structure of the thesis is depicted in Figure 1.4. The first part of this work, the *Preliminaries*, consists of the thesis introduction and a description of fundamentals on decision making, decision knowledge and the addressed development activities in Chapter 2. The next part is a detailed *Problem Analysis* of the open questions for each research goal. First, this analysis is performed by studying the documented decision making as the state of practice. The related case study is described in Chapter 3. Second, the scientific state of the art for decision documentation is investigated in a literature review on current documentation approaches for decision knowledge in Chapter 4. Part III presents the *Solution Approach* of this thesis. Based on the problem analysis findings, the documentation model for decision knowledge is explained in Chapter 5. Next, the design and implementation of the tool support for the documentation model is given in Chapter 6. Part IV contains the *Evaluation* of the documentation model and its tool support. In particular, the feasibility of the model to support decision documentation is assessed in an empirical study presented in Chapter 7. Furthermore, the tool support is evaluated in a case study described in Chapter 8. Finally, the thesis is concluded with

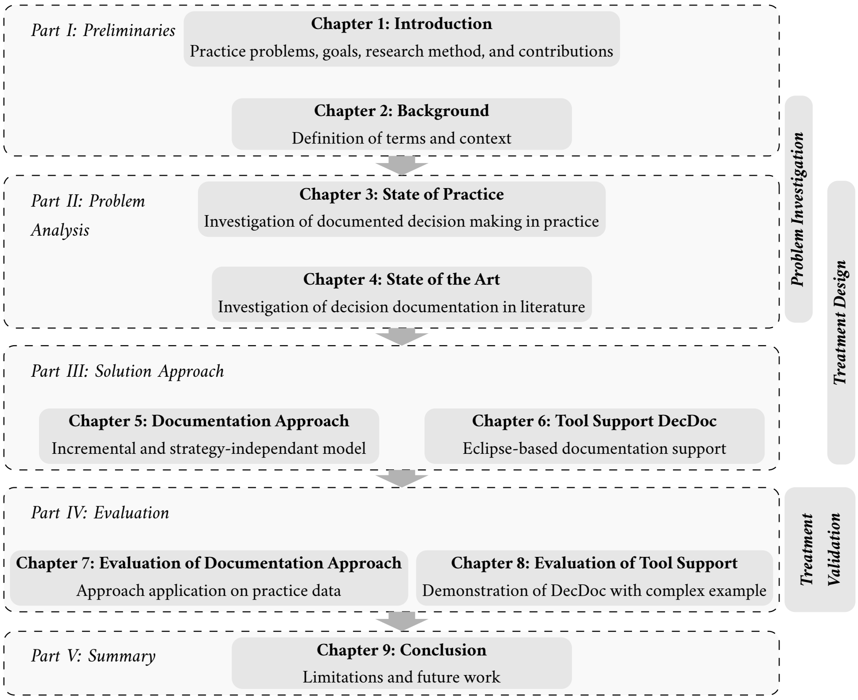

Part I: Preliminaries
Chapter 1: Introduction
Practice problems, goals, research method, and contributions
Chapter 2: Background
Definition of terms and context
Part II: Problem Analysis
Chapter 3: State of Practice
Investigation of documented decision making in practice
Chapter 4: State of the Art
Investigation of decision documentation in literature
Part III: Solution Approach
Chapter 5: Documentation Approach
Incremental and strategy-independant model
Chapter 6: Tool Support DecDoc
Eclipse-based documentation support
Part IV: Evaluation
Chapter 7: Evaluation of Documentation Approach
Approach application on practice data
Chapter 8: Evaluation of Tool Support
Demonstration of DecDoc with complex example
Part V: Summary
Chapter 9: Conclusion
Limitations and future work
Problem Investigation
Treatment Design
Treatment Validation

Figure 1.4: Overview of Chapters in this Thesis

a presentation of its limitations and ideas for future work in Chapter 9.

1.5 Previous Publications

Parts of the studies, formal concepts and implementation results for the tool support described in this thesis were published in scientific venues beforehand. This section gives an overview in which publications these parts can be found. The results from the state of practice (cf. Chapter 3) and further fundamentals on decision making and decision knowledge (cf. Chapter 2) were published in:

> Tom-Michael Hesse, Veronika Lerche, Marcus Seiler, Konstantin Knoess, and Barbara Paech (2016). "Documented decision-making strategies and decision knowledge in open source projects: An empirical study on Firefox issue reports". In: *Information and Software Technology* 79, pp. 36–51

A previous version of the literature review from the scientific state of the art and parts of the background related to decision making (cf. Chapter 4) are given in:

> Barbara Paech, Alexander Delater, and Tom-Michael Hesse (2014). "Supporting Project Man-

agement Through Integrated Management of System and Project Knowledge". In: *Software Project Management in a Changing World*. Ed. by Guenther Ruhe and Claes Wohlin. Berlin, Heidelberg: Springer, pp. 157–192

An early version of the documentation model for decisions (cf. Chapter 5) has been described in:

> Tom-Michael Hesse and Barbara Paech (2013). "Supporting the Collaborative Development of Requirements and Architecture Documentation". In: *Proceedings of the 3rd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks'13)*. IEEE, pp. 22–26

The conceptual model for importing knowledge on security-related decisions from use case descriptions (cf. Chapter 5) and ideas for supporting this import within the DecDoc tool support (cf. Chapter 6) were published in:

> Tom-Michael Hesse, Stefan Gaertner, Tobias Roehm, Barbara Paech, Kurt Schneider, and Bernd Bruegge (2014). "Semiautomatic security requirements engineering and evolution using decision documentation, heuristics, and user monitoring". In: *Proceedings of the 1st International Workshop on Evolving Security and Privacy Requirements Engineering (ESPRE)*. IEEE, pp. 1–6

The conceptual model for annotating decision knowledge within code (cf. Chapter 5) as well as its implementation within the DecDoc tool support (cf. Chapter 6) are given in:

> Tom-Michael Hesse, Arthur Kuehlwein, Barbara Paech, Tobias Roehm, and Bernd Bruegge (2015). "Documenting Implementation Decisions with Code Annotations". In: *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*. KSI Research, pp. 152–157

The evaluation results of DecDoc with decision data from different industry design sessions (cf. Chapter 7) were presented in:

> Tom-Michael Hesse and Barbara Paech (2016). "Documenting Relations Between Requirements and Design Decisions: A Case Study on Design Session Transcripts". In: *Requirements Engineering: Foundation for Software Quality: 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings*. Ed. by Maya Daneva and Oscar Pastor. Cham: Springer International Publishing, pp. 188–204

In addition, decision data of an architectural logbook was analyzed using the documentation model to evaluate it (cf. Chapter 7). The results were reported in:

> Tom-Michael Hesse, Christian Kuecherer, and Barbara Paech (2015). "Experiences with Sup-

porting the Distributed Responsibility for Requirements through Decision Documentation". In: *Softwaretechnik-Trends* 35.1, pp. 14–15

Finally, a brief overview of the architecture and functionality of DecDoc (cf. Chapter 6) and its evaluation (cf. Chapter 8) was presented in:

Tom-Michael Hesse, Arthur Kuehlwein, and Tobias Roehm (2016). "DecDoc: A Tool for Documenting Design Decisions Collaboratively and Incrementally". In: *2016 1st International Workshop on Decision Making in Software ARCHitecture (MARCH)*. IEEE, pp. 30–37

2 Background

In the following sections, terms and topics are introduced which are fundamental for this thesis. First, *decision problems* are defined, as they form the context for decision making strategies, decision knowledge and different kinds of decisions. Second, the *development activities* addressed by this thesis are described to introduce their typical activities, tools, and decisions. Third, different kinds of *decision making strategies* with corresponding example strategies are presented. Finally, the term *decision knowledge* is described in more detail with fundamentals on documentation and management of decision knowledge.

2.1 Decision Problems

Decision making is a problem solving activity for developers, as developers explore and structure the underlying decision problem in order to make a decision. Thereby, they acquire and evaluate knowledge on the decisions' context and its potential solutions [Zannier, Chiasson, and Maurer 2007]. Thus, decision problems influence the applied decision making strategies. Moreover, they are the origin for all decision knowledge, and drive the kind of decision which is taken to solve the decision problem. Decision problems consist of a set of different options and criteria to evaluate these options [Ngo and Ruhe 2005]. Two kinds of decision problems can be distinguished: *Well-structured* and *ill-structured* problems [Zannier, Chiasson, and Maurer 2007]. Problems are well-structured if developers are aware of criteria indicating appropriate solutions for the given problem. In this case, developers can search for viable solution options in a structured and managed way using these criteria. Moreover, different solutions become comparable, as their outcome typically can be evaluated as true or false [Paech, Delater, and Hesse 2014]. According to these attributes, a chess game is a typical example of well-structured decision problem. On the contrary, problems are ill-structured, if criteria for searching and evaluating solution options are not transparent to the developers and need to be revealed. Such problems are closely related to wicked problems, for which specifying the problem appropriately is a major part of solving the problem [Zannier, Chiasson, and Maurer 2007]. Specifying these problems is difficult, as there is no stopping rule for evaluating solutions that could be formalized [Hesse, Lerche, et al. 2016]. In addition, solution options cannot be evaluated as true or false, but only as good or bad [Paech, Delater, and Hesse 2014]. According to the described properties, the architecture and design of a building is an example for an ill-structured problem.

Due to their characteristics, each kind of decision problem is related to the decision making strategy applied by developers: whereas well-structured problems promote the usage of RDM [Zannier, Chiasson, and Maurer 2007], ill-structured problems are often addressed by using NDM [Klein and Klinger 1991]. However, in practice, developers also mix and merge both RDM and NDM for the same decision problem [Zannier, Chiasson, and Maurer 2007]. An overview of the described attributes of both kinds of decision problems is given in Table 2.1.

Characteristic	**Well-structured Problems**	**Ill-structured Problems**
Criteria for Solution Evaluation	Transparent	Unrevealed
Process of Problem-solving	Structured, managed	Unstructured
Evaluation of Solutions as	True or false	Good or bad
Related Decision making Strategy	RDM	NDM

Table 2.1: Characteristics of Decision Problems

2.2 Development Activities

Before decision making strategies and decision knowledge are introduced in detail, a brief overview of the addressed software development activities, their software development tools, and related potential decision problems is given. In general, *software development activities* (abbreviated as *development activities*) are "technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system" [Sommerville 2010, p. 36]. Whereas the organization of these activities and their actual course of actions may vary according to the process model, project, and the people involved, their goals and methods are similar [Sommerville 2010]. During each development activity, developers typically use different *software development tools* (abbreviated as *development tools*), which "are programs that are used to support software engineering process activities" [Sommerville 2010, p. 37]. Such tools may be compilers, debuggers, editors for textual and graphical artifacts, or interactive development environments (IDEs) [Sommerville 2010].

This thesis aims to support decision documentation during the development activities *requirements engineering*, *design*, and *implementation* in general. These activities are fundamental to both, classic development process models, such as the waterfall model, as well as incremental process models, like SCRUM [Sommerville 2010]. Decision documentation during the further activities of software validation and evolution is not addressed primarily. First, software validation typically evaluates results of previous development activities. Therefore, decisions made during validation typically do not shape the system under construction, but organize the type, content, and sequence of tests. Furthermore, in case validation decisions impact the system, they are likely to require further requirements engineering, design, or implementation to be carried out by the developers. Then, validation decisions will be captured there. Second, software evolution resembles a continuous process of software development in multiple iterations, rather than a single development activity. Then, again, evolution decisions will be captured within the addressed development activities. However, such evolution decisions may require to adapt or even withdraw decisions previously made.

Requirements Engineering

During *requirements engineering* (abbreviated as *RE*), developers are concerned with "understanding and defining what services are required from the system and identifying the constraints on the system's operation and development" [Sommerville 2010, p. 36]. Requirements of the stakeholders and system requirements need to be elicited, specified, validated, and managed. An important result of this activity is an agreed requirements document with high-level statements for users and customers, as well as a detailed system specification to guide the further development activities [Sommerville 2010]. Different methods and tools exist to specify the requirements within this document. For instance, *use cases* are often used to document the step-wise "interactions between the system and its users or other systems" [Sommerville 2010, p. 107], including the system functions to respond to user actions. Use cases may be specified in textual templates or as graphical models. For instance, such models can be created in use case diagrams using the *Unified Modeling Language* (abbreviated as UML) [Sommerville 2010]. Documentation of requirements is supported by various tools, such as IBM Rational DOORS [*IBM Rational DOORS* 2017] or Atlassian JIRA [*Atlassian JIRA* 2017]. Besides documenting requirements, these tools typically also support to version the requirements and link requirements to other development artifacts. Decision problems related to RE results typically result from identifying and defining architecturally significant requirements, such as core features and quality attributes of the system [Chen, Ali Babar, and Nuseibeh 2013]. An example addressed in this thesis are requirements related to system security [Hesse, Gaertner, et al. 2014]. However, decision problems may also be related to the RE process, such as the selection of the appropriate RE method [Jiang and Eberlein 2003].

Design

When developers *design* a software, they create and refine "a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used" [Sommerville 2010, p. 38]. The actual design actions vary according to type of system being developed. However, two general actions can be distinguished. First, developers create a coarse-grained *architectural design* to identify the overall system structures with principal components and their respective relationships [Sommerville 2010]. Second, a more fine-grained design details the interfaces, components and data structures for each principal component [Sommerville 2010]. Typically, such design actions are documented in UML diagrams, which can be created by specialized editors for the UML notation. Important decision problems during design primarily concern architectural design, such as decisions on the system's principal structure [Jansen and Bosch 2005], or the engineering of software product lines [Capilla and Babar 2008]. However, also more fine-grained decision problems may arise. Examples are the detailed design of components and classes with their attributes, methods, and relations [Davide Falessi, Cantone, and Becker 2006], and the selection of appropriate design patterns and styles for these tasks [Ali Babar and Gorton 2007]. Decisions on the selection of such patterns and styles concern the design process. They can be managed according to sophisticated decision frameworks [Davide Falessi, Cantone, Kazman, et al. 2011].

Implementation

Developers are concerned with the *implementation* of the software when they create code to realize the system design [Sommerville 2010]. This activity highly depends on the personal experience and preferences of developers, so there is no common or general process for implementation [Sommerville 2010]. For instance, developers may start implementing those components first which are best understood. Other developers might start with coding unfamiliar objects, because they can better estimate the effort necessary for the known components [Sommerville 2010]. However, developers typically use IDEs to develop code and implement object-oriented designs. A prominent example for such an IDE is Eclipse [*Eclipse Project* 2016], which is very popular support tool for developing Java programs [Geer 2005]. Furthermore, code is usually stored in version control systems integrated with an IDE, such as SVN [*Apache Subversion* 2016] or git [*Git* 2016]. Then, each developer creates individual code, and merges and shares the implementation results with the team. Also, issue tracking systems can be used to document and manage knowledge on ongoing tasks and their progress during the implementation process [Ko and Chilana 2011]. Furthermore, developers may document important knowledge directly within their code using annotations, such as Javadoc [*Javadoc Documentation by Oracle* 2016]. With these annotations, developers can create structured comments, which contain information about the annotated code artifacts. An example is the textual explanation for parameters of a given function. Although there is no standardized overall process for implementation, still many important decisions are made during this activity. For instance, developers have to decide on the optimal implementation of an algorithm, or on how to adapt a given function to address changed requirements [Lougher and Rodden 1993].

2.3 Decision Making Strategies

In general, decision making strategies describe how a solution for a given decision problem is determined [Paech, Delater, and Hesse 2014]. *Rational decision making* and *naturalistic decision making* can be distinguished as two different kinds of decision making strategies [Lipshitz et al. 2001; Zannier, Chiasson, and Maurer 2007]. This means that both terms subsume different concrete approaches and theories of decision making.

Rational decision making (RDM)
The origin of most RDM approaches is the development of social sciences in the 1950s, when research on decision making significantly increased [Maule 2010]. Research on RDM dominated the overall investigation of decision making for decades [Lipshitz et al. 2001]. According to Lipshitz et al., typical approaches for RDM are classical decision making, behavioral decision theory, judgement and decision making, and organizational decision making. All RDM approaches share the important assumption that humans are rational decision makers who search systematically for relevant information and evaluate each solution option with regard to this information [Hesse, Lerche, et al. 2016; Jonassen 2012]. Based on this assumption, the RDM strategies have three strategy elements in common [Zannier, Chiasson, and Maurer 2007]. First, RDM strategies aim to select the optimal solution by intentional *choice* from a set of available alternatives. Thus, the goal of RDM is *optimizing* the solution outcome in relation to the given criteria [Zannier, Chiasson, and Maurer 2007]. Second, RDM is concerned with processing and comparing the prerequisites for and outcomes of different alternatives. Therefore, a *criterion evaluation* for all given alternatives and criteria is performed in order to determine the optimal solution [Zannier, Chiasson, and Maurer 2007]. Third, RDM typically relies on a formalism to develop context-free and abstract representations of a decision [Paech, Delater, and Hesse 2014]. Thereby, RDM approaches aim to achieve a comprehensive decision making process. This means that determining the optimal solution should happen intentionally and analytically [Lipshitz et al. 2001]. Thus, the solution selection within RDM is a process of *consequential choice*.

A prominent example of an RDM strategy is the Analytic Hierarchy Process (AHP) by Saaty. Its core steps are depicted in Figure 2.1. First, an analysis of the decision problem and the required kind knowledge to solve the decision problem takes place. Then, the goal is defined, which needs to be achieved in order to solve the decision problem. This goal is broken down into quantifiable objectives, which serve as criteria to identify and evaluate all potential alternatives. After all knowledge is acquired, comparison matrices are created for the criteria, and all alternatives are evaluated pairwise for each criterion. Finally, overall priorities are calculated for each alternative based on all of its evaluation results. The alternative with the highest priority represents the optimal solution and

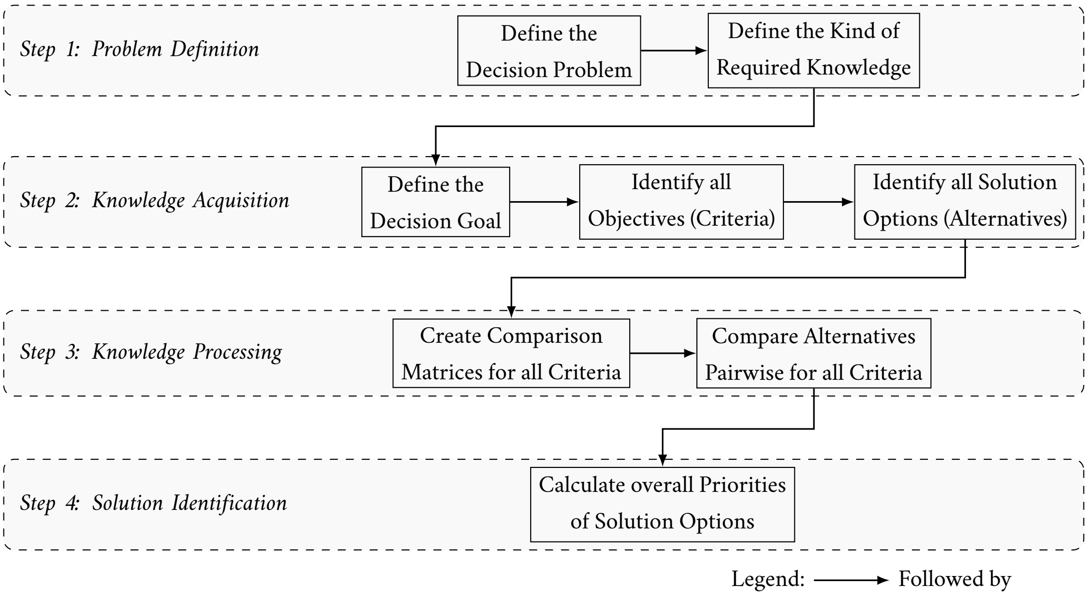

Step 1: Problem Definition
Define the Decision Problem
Define the Kind of Required Knowledge
Step 2: Knowledge Acquisition
Define the Decision Goal
Identify all Objectives (Criteria)
Identify all Solution Options (Alternatives)
Step 3: Knowledge Processing
Create Comparison Matrices for all Criteria
Compare Alternatives Pairwise for all Criteria
Step 4: Solution Identification
Calculate overall Priorities of Solution Options
Legend: → Followed by

Figure 2.1: The Analytic Hierarchy Process according to [Saaty 2008]

should be chosen according to AHP [Saaty 2008].

It should be noted, that for RDM both the addressed decision problem and the related decision maker need to fulfill several prerequisites. First, the decision maker requires accessible knowledge about all relevant solution options, their outcomes and the probabilities of these outcomes [Zannier, Chiasson, and Maurer 2007]. In addition, decision makers are expected to actually strive for selecting the optimal solution to a given decision problem [Hesse, Lerche, et al. 2016]. Moreover, it is important that there are no time limitations for acquiring all relevant knowledge and performing the evaluation of alternatives [Zannier, Chiasson, and Maurer 2007]. As these assumptions clearly restrict the applicability of RDM approaches in practice, doubts were raised by researchers concerning RDM. Most importantly, Tversky and Kahneman investigated different real-world decision problems in the 1970s. Their study showed that decision makers often fail to make rational decisions as prescribed by RDM approaches [Hesse, Lerche, et al. 2016; Tversky and Kahneman 1974]. Instead of performing a thorough evaluation of solution options, decision makers used heuristics and shortcuts for their decisions. Based on this finding, Kahneman developed the "Two Systems" approach. Here, decision makers may vary between rational decision making as "System 1" and a more intuitive decision making with heuristics and biases as "System 2" [Kahneman 2011]. This approach was investigated in detail for decisions during software design in recent studies [Razavian et al. 2016; Tang, Aleti, et al. 2010].

In general, three fundamental concerns emerged towards RDM. First, as described by Tversky and

Kahneman, humans struggle to reason as prescribed by RDM due to their limited cognition and memory [Maule 2010]. Instead, they are often influenced by biases and heuristics [Tversky and Kahneman 1974]. Second, decision makers often do not aim to make the optimal decision, but instead strive to find and select a satisfying solution [Klein 2008; Zannier, Chiasson, and Maurer 2007]. Then, they need less cognitive energy for their decision, as a satisfactory solution typically can be determined easier than the optimal one [Jonassen 2012]. Third, Klein, Calderwood, and Clinton-Cirocco criticize that RDM does not contribute to the understanding of decision making in real-world situations, because RDM is typically investigated under laboratory settings [Klein, Calderwood, and Clinton-Cirocco 2010]. These settings are standardized, and, therefore, cannot reflect contextual and situational factors of individual decisions. Thus, these decisions are less likely to create an actual impact on study participants [Hesse, Lerche, et al. 2016].

Naturalistic decision making (NDM)

The research on NDM originates from an initial conference in 1989 on the shortcomings of RDM research [Lipshitz et al. 2001]. In particular, the difficulties with applying RDM approaches on real-world decision situations were discussed. Thus, an important aim of NDM research is to address the criticism concerning RDM within its approaches. This is reflected in the definition of Klein, that natural decision making is the commitment of a decision maker to a course of action with plausible alternatives, even if these alternatives were not identified or compared [Klein 2008]. Like for RDM, a variety of different decision making approaches and theories form the term NDM. Most NDM approaches are derived from findings of field studies, such as the observation of decision problems in an emergency or military context [Lipshitz et al. 2001]. Examples are the recognition-primed decision model, decision making in natural contexts, "Image Theory", decision making as argument-driven action, and explanation-based decision making [Orasanu and Connolly 1993].

Also NDM strategies share similar strategy elements. The goal of the decision making process is shifted from identifying the optimal solution to determining a *sufficient* one [Hesse, Lerche, et al. 2016; Klein 2008]. To do so, decision makers use their experience, heuristics, and their personal knowledge [Zannier, Chiasson, and Maurer 2007]. Thus, it is unlikely that all existing solution options are identified and compared. Instead, typically a *singular evaluation* of a past decision situation and its related solution is performed. Hereby, the former situation is *assessed* and *matched* to the current decision situation by its characteristics [Klein 2008]. To enable decision makers to match situations, NDM approaches also aim to describe the prerequisites and rules for matching different decision situations. Therefore, also informal models with incomplete information on the decision problem are accepted [Paech, Delater, and Hesse 2014].

The recognition-primed decision model (RPD) by Klein is a prominent NDM strategy example. Its steps are depicted in Figure 2.2. The focus of the RDM model is to describe how decision makers make use of their experience within decision making. To do so, the current decision situation is assessed for any pattern, which can be matched to previous decisions [Klein 2008]. Then, a mental simulation is performed whether the formerly applied solution can be applied also for the current decision. In particular, previous solutions can be modified to fit to the current situation in case the mental simulation shows that adaptions are necessary. Then, an incremental process of solution refinement and mental simulation is performed. However, the mental simulation may be omitted if time pressure or other resource limitations force the decision maker to take a decision immediately [Klein 2008]. Obviously, the process of pattern matching implies that a decision is made without searching for all potential solution options, as the decision makers is likely to choose the first working solution, which passes the situation assessment and pattern matching as well as the mental simulation [Klein 2008]. Thus, RPD promotes decisions with a sufficient solution.

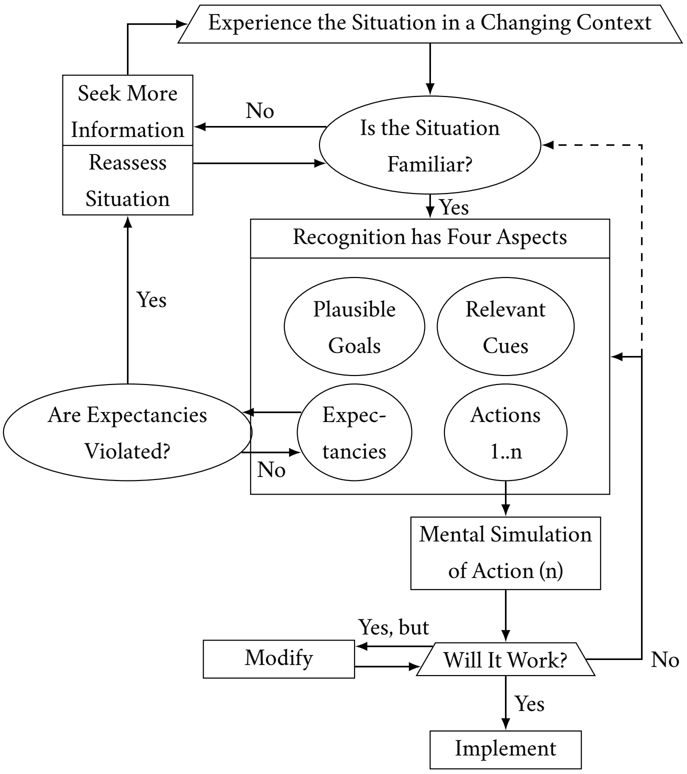

Experience the Situation in a Changing Context
Seek More Information
Reassess Situation
No
Is the Situation Familiar?
Yes
Recognition has Four Aspects
Plausible Goals
Relevant Cues
Expec-tancies
Actions 1..n
Yes
Are Expectancies Violated?
No
Mental Simulation of Action (n)
Yes, but
Modify
Will It Work?
No
Yes
Implement

Figure 2.2: The Recognition-primed Decision Model according to [Klein 2008]

The applicability of NDM is grounded on several assumptions towards decision makers and decision problems [Orasanu and Connolly 1993; Klein and Klinger 1991]. Most importantly, NDM expects humans to act in real-time to solve a decision problem under dynamically changing conditions. These conditions may include uncertainty, the possibility of contradictory goals, time stress, and potentially far-reaching consequences of the decision. Thus, decision makers typically are required to be experienced or even experts for the considered kind of decision problem [Lipshitz et al. 2001]. With respect to these assumptions, Gore et al. criticized NDM for being based mainly on qualitative field studies. Therefore, NDM approaches highly depend on the setup and methodology of the studies they are built on [Gore et al. 2006].

Comparison of RDM and NDM

The different strategy elements of RDM and NDM are summarized in Table 2.2. If identified in a decision making process, these elements indicate that activities of decision makers belong to the respective kind of strategy. RDM and NDM strategies differ according to the selection mechanism for and the promoted quality of solutions, and the process steps required for this selection. Whereas RDM enforces the selection of the optimal solution according to the identified criteria, NDM promotes the application of a sufficient solution according to the personal knowledge of the decision maker and experiences from previous decisions. The term *sufficient* means that the solution is applicable and working for the current decision problem. Then, only a singular evaluation of this solution needs to be performed. These decision making elements are the basis for the coding of decision making elements in the study described in Chapter 3.

Kind of Strategy Element	**RDM**	**NDM**
Selection Mechanism for Solution	Choice for Optimal Solution	Match for Sufficient Solution
Problem Exploration	Identification of all relevant criteria and solution options, evaluation of all options according to these criteria	Assessment of and pattern matching for current and past decision situations
Solution Identification	Consequential choice based on evaluation results	Evaluation of a singular solution, e.g. by mental simulation

Table 2.2: Comparison of Strategy Elements for RDM and NDM

2.4 Decision Knowledge

All knowledge related to a decision problem is referred to as *Decision Knowledge* (abbreviated as *DK*). Smaller parts of decision knowledge concerning a particular aspect of a decision are called *Decision Knowledge Elements* (abbreviated as *DKE*). Knowledge on a concrete decision problem consists of various DKE. Fundamental sets of DKE forming the overall decision knowledge on a decision are depicted in Figure 2.3. Most basically, decision knowledge contains elements on the problem description [Ngo and Ruhe 2005], which are represented by *Problem* DKE. Also, decision knowledge comprises one or more *Solution* DKE, such as different alternatives to solve the problem [Ngo and Ruhe 2005]. In addition, the *Context* DKE relate the decision problem to potential solutions, e.g. by enabling developers to evaluate different solutions for their feasibility and appropriateness to solve the given problem. A basic example for context knowledge is a criterion [Ngo and Ruhe 2005]. Moreover, these elements can be enriched by *Rationales*. For instance, rationales may justify the problem description, argue for an alternative, or highlight context information. Typically, different DKEs can form a hierarchy according to their degree of abstraction [Tang, Jin, and Han 2007]. As an example, consider the DKE *Argument* as a more fine-grained sub-element of the DKE *Rationale*. Finally, entire decision problems can be related to each other in various ways. For instance, Kruchten, Lago, and Vliet outline different kinds of relationships for design decisions, such as *depends on*, *conflicts with*, and *bound to* between two decisions [Kruchten, Lago, and Vliet 2006]. Also, DKEs can be linked with other DKEs. One example is the argument: it might *strengthen* or *weaken* other DKE, like alternatives [MacLean et al. 1991].

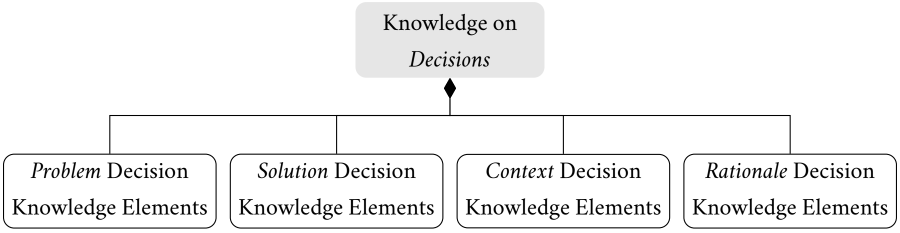

Knowledge on
Decisions
Problem Decision Knowledge Elements
Solution Decision Knowledge Elements
Context Decision Knowledge Elements
Rationale Decision Knowledge Elements

Figure 2.3: Basic Decision Knowledge Elements

It is important to note that not all relevant DKE are available to or accessible by decision makers by default [Zannier, Chiasson, and Maurer 2007]. Instead, relevant decision knowledge may be more or less structured according to the kind of decision problem (cf. Section 2.1). Then, it has to be uncovered and processed by decision makers when following one or multiple decision making strategies (cf. Section 2.3).

Documentation of Decision Knowledge

The first well-recognized approach for documenting development decisions for software are *Issue-based Informations Systems* (IBIS) by Kunz and Rittel in the 1970s. The conceptual IBIS approach aims to support problem-solving in groups by representing decision problems as *issues* [Kunz and Rittel 1970]. When discussing these issues, *arguments* are constructed to support or attack different positions [Kunz and Rittel 1970]. In particular, additional information may be requested from experts or additional documentation to further substantiate issues and arguments. Later, this approach was implemented as the hypertext tool *gIBIS* by Conklin and Begeman to actually enable developers to document and explore decision knowledge within a software tool [Conklin and Begeman 1988].

In the early 1990s, two fundamental meta-models for decision documentation were presented: the *Questions, Options and Criteria* (QOC) approach by MacLean et al., and the *Decision Representation*

Language (DRL) by Lee. QOC consists of six important elements [Paech, Delater, and Hesse 2014; MacLean et al. 1991]. *Questions* are used to capture and structure the problem definition. *Options* represent all considered solution alternatives. *Criteria* are used to evaluate and rank these options. A concrete evaluation of an option by criteria is captured in an *assessment*. In addition, *arguments* can be used to attack or support all of these elements. Finally, the selected option to solve the given question is captured as *decision*. Similar elements are proposed by DRL [Paech, Delater, and Hesse 2014; Lee 1991; Lee 1989]. Here, a *goal* describes a particular aspect of a decision problem, whereas *alternatives* represent different solution options. *Claims* are used in the way of arguments. In addition, DRL provides further context representations, such as *questions* to represent uncertainty, and *procedures* to describe courses of action. However, it should be noted that the name and meaning of similar documentation elements for decision knowledge may vary between different approaches. This is illustrated by a third important documentation approach, the *Scenario-based Claims Analysis* (SCA) proposed by Carroll and Rosson in 1992. In this approach, claims are not primarily used as arguments. Instead, they represent a particular solution alternative, which can be backed up or challenged by further individual *arguments*. Also, there is no explicit problem definition for a decision, but a description of the decision situation as *scenario*.

In the last decades, many further meta-models for decision documentation during software development emerged. Typically, they address decision documentation within selected development activities. For instance, Aurum, Wohlin, and Porter describe in their case study which kinds of decisions can be captured during the requirements engineering process [Aurum, Wohlin, and Porter 2006]. Also, they highlight important decision knowledge that should be documented for each kind of decision, such as knowledge on business processes, requirement priorities, release plans, and related features. For decisions during design, various documentation approaches exist. Many of them were investigated in comparative studies [Paech, Delater, and Hesse 2014; Tang, Avgeriou, et al. 2010; Ali Babar, Boer, et al. 2007]. A prominent example is the documentation template of Tyree and Akerman, as it is the foundation for many other documentation approaches for design decisions [Davide Falessi, Cantone, Kazman, et al. 2011]. The template proposes to describe the actual design decision, but also its assumptions, related requirements, and impacts on the design [Tyree and Akerman 2005]. Implementation decisions can be documented typically as structured comments directly within the code [Burge and Brown 2008; Canfora, Casazza, and De Lucia 2000; Lougher and Rodden 1993] by adding a textual explanation of the decision near the code realizing it. Besides the decision description, links to external knowledge bases may exist, e.g. to relate documented decisions with corresponding requirements [Burge and Brown 2008]. A more detailed comparison of current documentation approaches for decisions in different development activities is given in the literature review in Chapter 4.

Depending on their documentation elements, documentation approaches may be more or less suitable for documenting decision knowledge originating from a particular decision making strategy [Paech, Delater, and Hesse 2014]. For instance, QOC fits better to knowledge acquired from RDM, as different solution options and a set of criteria can be captured explicitly. In contrary, SCA fits better to knowledge acquired from NDM, as the situational description of decisions is covered, and single solutions with their related argumentation are highlighted. However, it should be noted that no approach explicitly addresses knowledge resulting from NDM. Also, current approaches do not cover knowledge resulting from a mix of decision making strategies.

Management of Decision Knowledge

Documenting decision knowledge is an important part of decision knowledge management. Typically, software tools are required to enable and support approaches for decision knowledge management in general, and for documentation of decisions in particular [Arnott and Pervan 2008]. Thus, documentation approaches for decision knowledge often describe both, documentation meta-models and software to allow for decision documentation according to this meta-model. Also, decision knowledge management is an important field within research on *Decision Support Systems* (abbreviated as DSS), which are IT-based systems to support decision processes [Arnott and Pervan 2008]. Other major fields of DSS research are *Personal* and *Group Support Systems* to support personal and group decision tasks, *Data Warehousing* to provide an infrastructure for decision-related data analysis in large scales, and *Enterprise Reporting and Analysis Systems* to provide performance management and business intelligence [Arnott and Pervan 2008]. Whereas these major fields already had a significant presence in practice [Arnott and Pervan 2008], knowledge management-based systems still need to be improved in order to increase their value for application in practice. Most importantly, knowledge management for decision knowledge needs to reflect the decision making strategies actually applied by the developers [Maule 2010; Zannier, Chiasson, and Maurer 2007].

Chan and Song describe further characteristics shared by successful DSS [Chan and Song 2010]. They should be *easy to use* and provide an appropriate *presentation format* for the available information and functionality, so that there is a cognitive fit between the presentation and the user. In addition, such systems should offer *decisional guidance* by providing assistance during the decision-making process and allow for enough *user interaction*. However, the system should also *limit the available options* within the decision-making process in an appropriate way and give *feedback* to the user about these restrictions.

Documentation of decision knowledge can be supported by different kinds of software tools. First, *specialized* tools may be used. These tools typically implement documentation support for one particular decision documentation meta-model. They can be realized as *stand-alone* software, like ADDSS [Capilla, Nava, Pérez, et al. 2006], or are *integrated* with other tools used by developers. For instance, the SEURAT tool is implemented as plugin for Eclipse [Burge and Brown 2008]. Also *hybrid* approaches, such as Knowledge Architect [Jansen, Avgeriou, and Ven 2009], exist, which provide a stand-alone software as well as plugins for existing development tools. Second, *general* documentation tools can be used for documenting decisions during the development process. Such tools are not dedicated to capture decision knowledge, but may cover a broad range of different kinds of artifacts and knowledge. For instance, decisions may be documented in requirements specifications using a text editor [Jansen, Avgeriou, and Ven 2009], or in comments to issue reports in issue tracking systems [Ko and Chilana 2011].

Both, specialized and general documentation tools, may be closely integrated with IDEs. For instance, the specialized SEURAT tool is a plugin for the Eclipse IDE. Another example is the knowledge management tool UNICASE [*UNICASE Project* 2016]. This tool is also an extension to Eclipse. It will be enhanced with additional features to support decision documentation in a specialized way, implementing the approach presented in this thesis. The fundamental components of UNICASE are depicted in Figure 2.4. The core of UNICASE is a set of client plugins to support a knowledge model integrating project and system knowledge [Bruegge et al. 2008]. A variety of model elements are provided for documentation, such as Use Cases, UML diagrams, and action items. Instances of the model are versioned and exchanged between distributed development sites using the model versioning system EMFStore [*EMF Store* 2016]. In addition, the tool offers a generic editing support for this model as well as for any model extensions through the EMF Client Platform [*EMF Client Platform* 2016].

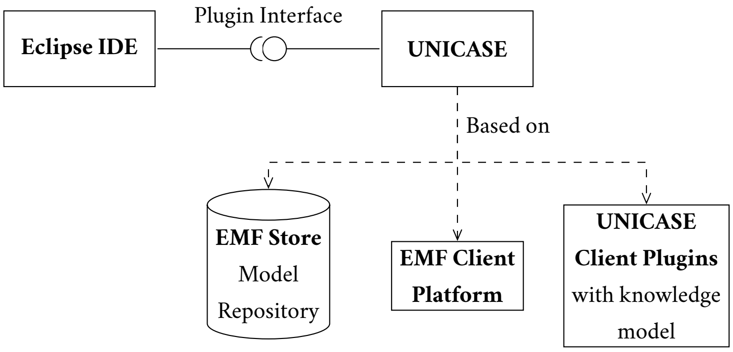


Figure 2.4: Fundamental Components of UNICASE

Part II

Problem Analysis

3 State of Practice for Decision Making Strategies

In this chapter, an empirical study on the documentation of decision making strategies in practice is presented. The study investigates comments to issue reports within the Firefox project in order to investigate how developers document knowledge resulting from their decision making strategies in practice. A full report on the study was published in [Hesse, Lerche, et al. 2016]. First, the Firefox project is introduced as study subject with the study goal. Also, relations between existing empirical studies and this study are discussed. Second, the research process of the study is described, including preparation, data coding, and analysis applied during the investigation of issue comments. Third, the results of data analysis are presented in detail. Fourth, these results are discussed in relation to requirements, which can be derived from the results for developing the documentation presented in this thesis. Finally, potential threats to validity are discussed together with mitigation measures applied during the study.

3.1 Study Foundations

In the following paragraphs, the Firefox project with its issue reports as the study subject and the study goal are introduced. In addition, the similarities and differences of this study and related existing empirical studies are discussed.

Study Subject and Goal

Developers in open source software projects typically document their development progress in issue tracking systems. There, developers can file issue reports to document recent development tasks and share the results with other developers and users. Also, issue reports may be created by the system users to inform the developers about problems, or to request improvements. Two different types of issue reports may be distinguished: *Bug reports* are concerned with system errors or corrections of functionality, whereas *feature requests* describe new or extended functionality. In consequence, issue reports typically indicate specific needs for changes and improvements ranging from the entire system to particular functionality. As developers address the reports in their development activities, they make decisions when implementing or rejecting the requested changes. These decisions are documented as comments to issue reports, so that decision knowledge is distributed to and shared between all involved developers and users. This is particularly important for large and globally distributed development teams with a heterogeneous user community, like the Firefox project. Prominent examples are decisions documented in issue reports of the Firefox project and the Linux kernel [Ko and Chilana 2011].

The goal of the study presented in this chapter is described using the Goal Question Metric (GQM) approach [Basili, Caldiera, and Rombach 1994]. GQM provides a structured template to formulate study goals, including the aim of investigation, the investigated matter, the purpose of investigating the data, the study context, and the viewpoint of examination. Table 3.1 shows the goal of this study, structured according to GQM.

GQM Template	**Goal Description**
Determine	significant quantitative effects
with respect to	documentation of knowledge according to the decision making strategies applied by developers
for the purpose of	improving the knowledge documentation in relation to decision making strategies used by developers
in the context of	development decisions in comments to issue reports for the open source project Firefox
from the viewpoint of	researchers.

Table 3.1: Description of Study Goal with GQM

The presented study aims to investigate documented decision making by identifying documented strategy elements and examining their quantity. In detail, the quantity of NDM and RDM as well as their strategy elements and relations between these elements are investigated. Thereby, the study contributes insights to answer the open questions for research goal 1. The identification of strategy elements was performed by manually reading and categorizing the content of each comment to an issue report. This investigation step is called *coding* of the data. Thereby, the presented study complements existing empirical studies on decision making (cf. the next paragraph for a detailed analysis of related studies). First, current studies on decision making in software development focus on particular kinds of decisions, such as design decisions (e.g., in [Zannier, Chiasson, and Maurer 2007; Tang, Aleti, et al. 2010]). Second, their method of investigation is either to observe decision making processes of developers [Tang, Aleti, et al. 2010], or to perform interviews with developers for a retrospective examination of decision knowledge [Zannier, Chiasson, and Maurer 2007]. As a consequence, most of the existing studies perform only a qualitative analysis (cf. the studies of [Zannier, Chiasson, and Maurer 2007; Mentis et al. 2009; Tang, Aleti, et al. 2010]). In contrast, the presented study investigates documentation of the Firefox project with topical decisions not only on design, but also on requirements and implementation of the Firefox software. In addition, the coding of the comments provides a fine-grained data set of identified strategy elements, which are analyzed quantitatively. This allows for testing concrete hypotheses, as well as for exploratory analysis of the data.

It should be noted that this study aims to investigate documented decisions without restrictions regarding a given specific decision making process. This requires investigating a detailed documentation of decisions from a large software project, such as the Firefox project. According to Ko and Chilana, the Firefox project does not enforce a particular kind of decision making strategy and documentation method for decision knowledge for its developers [Ko and Chilana 2011]. Also, the documentation in the project results from real-world and complex development activities with their related decisions. In issue tracking systems, this documentation is made *explicit* and *available* for the study through issue reports with their comments [Hesse, Lerche, et al. 2016]. The documentation is explicit, because the members of development teams in huge open source projects typically do not work together in the same place, but are spread globally. Therefore, discussions on development issues and their resulting decisions need to be visible for all team members, so that they are shared in an explicit way through issue tracking systems. The documentation is available, because open source projects typically need to attract and integrate new developers continuously. Therefore, developers already involved in the project benefit from documenting discussions and decisions in a comprehensive way through issue tracking systems. In consequence, new developers can easily access and explore current and previous development results and the related decision knowledge.

Related Studies

Several studies have been performed to investigate decision making and documentation of decisions by software developers. Most importantly, Ko and Chilana also investigated discussions on development decisions formed by comments to issue reports within the Firefox project [Ko and Chilana 2011]. They analyzed quantitatively which rhetorical structures of argumentation emerged within these comments over time. Therefore, Ko and Chilana marked the contribution of a comment, for instance information clarifying the scope, dimension, or process of the discussed topic. The study findings suggest that the investigated discussions were mostly concerned with exploring the design space for potential issue solution in order to prepare the related decisions. Among other aspects, this exploration typically consists of discussions between the developers to clarify the usage options and qualities of solutions. Whereas these findings highlight the close relation between decision making and documentation, Ko and Chilana mainly investigate the argumentation structures in issue comments to reason about the discussion outcome. Thus, their codes and data analyses do not aim to provide detailed insights on the decision making strategies documented by developers. In contrast, the study presented in this thesis uses fine-grained codes to capture different kinds of decision making strategies in the given documentation.

Other related studies focus on decisions concerning software architecture and design. A well-recognized qualitative study on decision making was performed by Zannier, Chiasson, and Maurer. The study investigated how decision making strategies were applied by 25 professional developers in different software projects [Zannier, Chiasson, and Maurer 2007]. Developers were found to mix different kinds of decision making strategies for one decision. Also, this finding was confirmed for agile projects [Zannier and Maurer 2006]. Another study was performed by Mentis et al. to investigate rational decision making in groups with planning tasks. Therefore, three discussion rounds of 36 undergraduate and graduate students within 12 teams were captured [Mentis et al. 2009]. All statements of each discussion were categorized either as information sharing, interpretation, arguing, or summarizing. The authors found that the results varied between newly created and established teams. Whereas new teams were mostly concerned with information sharing, established teams used most statements for arguing. The beginning of the discussions was dominated by information sharing, whereas arguing dominated the end of the discussions [Mentis et al. 2009]. In 2010, a set of studies was performed on three recorded design sessions with different teams of two professional designers working on the same design task. The studies can be found in special issues of the journals *Design Studies* in 2010 and *Software* in 2012 as well as in a book presenting all original studies on this data set [Petre and Hoek 2013]. Among these studies is the work of Tang, Aleti, et al. who investigated the evolution of problems and solutions during decision making processes in two design sessions. As they investigate documented decision making by coding the designers statements, their study is related to the study presented in this chapter. The authors found that the identical design task was approached very differently in both sessions [Tang, Aleti, et al. 2010]. Whereas one team used a problem-driven approach to guide their design space exploration, the other team applied a solution-driven approach without defined structures. Recently, a study was presented by Tang and Vliet on the reasoning process of software designers to make decisions on design problems. Within the study, 32 students and 40 professionals were asked to make a design decision and state the considered rationale [Tang and Vliet 2015]. [Tang and Vliet 2015] found software designers to rely on satisficing solutions instead of searching for optimal solutions, which indicates the usage of NDM strategies.

However, none of the aforementioned qualitative studies provides a quantitative analysis of strategy elements to investigate the decision making processes of developers in a fine-grained way. Instead, the studies describe the identified strategy elements only in a textual summary. Regarding the study subject, only the studies of Ko and Chilana and Zannier, Chiasson, and Maurer investigate real-world decisions, whereas all other described studies use an experimental setup with prepared decision tasks. These setups are either academic example projects or well-defined industry cases. However, the decision making behavior of developers deviates when making decisions under real-world conditions [Klein, Calderwood, and Clinton-Cirocco 2010]. In the study of Zannier, Chiasson, and Maurer, interviews were performed to extract knowledge on decision making in retrospect. In consequence, developers might not remember or not state all relevant details from their past decision making, so that the acquired knowledge within the study may be incomplete or erroneous [Hesse, Lerche, et al. 2016]. In contrast, the study presented in this thesis relies on comments to issue reports, which can be added by developers over time. In these comments, developers capture the available knowledge continuously over time. Also, it is possible to state questions and clarifications via comments. Thus, decision knowledge within the issue comments is more likely to be comprehensive and complete than interviews on decisions in retrospect.

3.2 Research Process

In this section, the research process with details on study execution and data analysis for the presented empirical study are described. The research process is depicted in Figure 3.1. The study was carried out as case study research according to the guidelines of Runeson et al. for case studies [Runeson et al. 2012]. A pilot study was performed to explore documented decision making in issue comments of one Firefox branch within a master thesis [Knoess 2014]. Based on the insights of this pilot study, the research process for this study was developed.

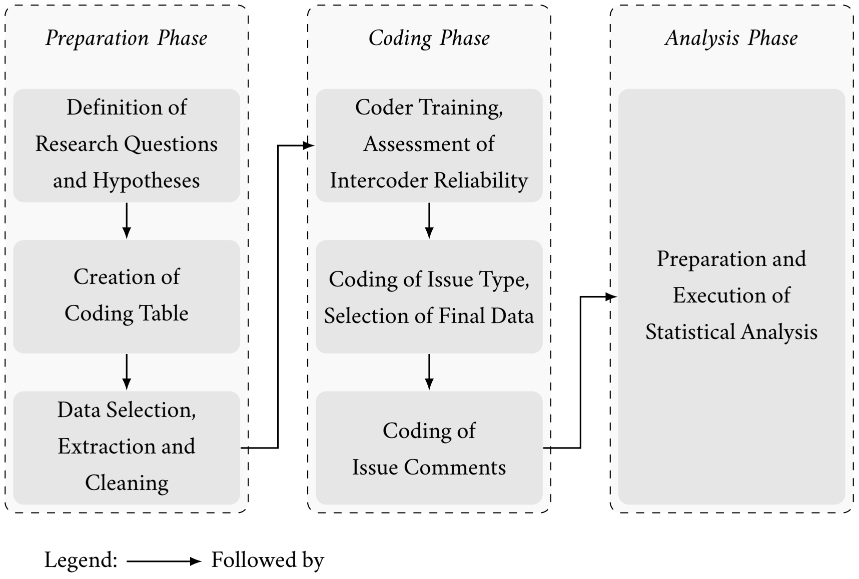

Preparation Phase
Definition of Research Questions and Hypotheses
Creation of Coding Table
Data Selection, Extraction and Cleaning
Coding Phase
Coder Training, Assessment of Intercoder Reliability
Coding of Issue Type, Selection of Final Data
Coding of Issue Comments
Analysis Phase
Preparation and Execution of Statistical Analysis
Legend:
Followed by

Figure 3.1: Overview of the Research Process

In the first phase, the open questions for research goal 1 (cf. Chapter 1) were refined to concrete research questions for this study. For selected research questions, hypotheses were defined based on the existing studies and decision making literature. Next, a coding table was created to classify issue comments for their documentation of one or multiple strategy elements. Then, the data selection, extraction and cleaning was performed for issue reports of two different Firefox branches. In the second phase, the actual coding of issue comments took place by two coders [Hesse, Lerche, et al. 2016]. Therefore, the coders were trained regarding the semantics and application of the defined codes. This was assessed by calculating the intercoder reliability for a training data set. Afterwards, all selected issue reports were classified either as bug report or feature request according to their content. This also lead to further balancing of the data set to ensure an optimal data quality. Finally, all issue comments were coded according to the coding table. In third phase, the statistical analysis of the coded comments was prepared and executed [Hesse, Lerche, et al. 2016].

3.2.1 Preparation Phase

Definition of Research Questions and Hypotheses

The research questions and hypotheses for this study were developed to provide insights for all open questions related to research goal 1 (cf. Section 1.1). These research questions were investigated by testing explicit hypotheses or by explorative analyses. In this study, it is assumed that developers of the Firefox project mainly document knowledge according to the kind of decision making strategy they use intuitively. In consequence, using a particular kind of decision making strategy should be reflected in the related comment to an issue report.

The first open question for research goal 1 addresses the actual quantity of NDM used and documented by developers in practice. Previous studies already found NDM to be a substantial part of decision making by developers in practice [Tang, Aleti, et al. 2010; Zannier, Chiasson, and Maurer 2007], whereas the actual amount of NDM used and documented by developers remains unclear. However, it is important to quantify this amount of applied and documented NDM, so that the knowledge documentation support for these decisions can be improved. For instance, if NDM is used in large quantities, documentation approaches should consider the iterative nature of NDM more closely during documentation. Also, NDM-specific documentation elements would be required to match knowledge originating from the particular strategy elements of NDM. Thus, *research question 1* (abbreviated as *RQ1*) is defined as: *Which percentages of NDM and RDM are documented?* Regarding RQ1, Zannier, Chiasson, and Maurer point out that for design decisions typically one kind of decision making strategy dominates the decision making of one developer [Zannier, Chiasson, and Maurer 2007]. This makes it less likely that both kinds of strategies are applied in a balanced mix. Also, NDM occurs more often than RDM under real-world conditions with different situational factors [Klein 2008; Orasanu and Connolly 1993]. Such factors can be "multiple deciders" [Orasanu and Connolly 1993], like different developers participating in issue discussions, or an "uncertain environment" [Orasanu and Connolly 1993], like incomplete and changing information in issue reports. These factors apply for the open source project Firefox with various, globally spread developers and time pressure through short release cycles [Khomh et al. 2012]. In consequence, *NDM is expected to dominate RDM in the investigated comments (hypothesis H1)*.

The second open question for research goal 1 asks for strategy elements of NDM (cf. definitions in Section 2.3) actually documented by developers and their relation to RDM strategy elements. Existing studies did not reveal in detail, which strategy elements of NDM and RDM are documented by developers in which quantity. However, this insight is essential to provide appropriate documentation entities to support decision documentation. Otherwise, the provided entities do not match those actually required by the developers. Therefore, it is also important to examine the relation of NDM to RDM strategy elements during documentation. Then, a mix of both strategies can be supported within the documentation model and its related tool support. Thus, *research question 2* (abbreviated as *RQ2*) is defined as: *Which quantities are found for each strategy element and which relationships between strategy elements can be identified?*. To investigate this research question, explorative analyses are performed without hypothesis formation for RQ2.

The third open question for research goal 1 is concerned with specific distributions for the percentage of NDM according to specific kinds of decision situations. For this study, different kinds of decision situations map to different types of issue reports due to the different characteristics of feature requests and bug reports with their related decisions. In consequence, it should be investigated whether the percentage of NDM to RDM differs for these issue types. This helps to clarify whether developers address different types of decision situations with different kinds of decision making strategies. Thus, *research question 3* (abbreviated as *RQ3*) is defined as: *Is the proportion of NDM to RDM related to the issue type?*. In detail, bug reports typically require developers to improve or adapt an existing functionality. Then, the original solution and a claim for the intended change are already given. Thus, development activities such as testing and debugging are performed to analyze the current situation and determine applicable solutions to realize the intended change. This fits to the situation analysis and matching procedure of NDM [Klein 2008] and could result in a higher percentage of NDM for bug reports. In contrast, feature requests either address functionality that has to be newly developed, or relate to given functionality that has to be extended significantly. Thereby, it is not clearly defined for developers how a solution can be achieved due to various potential implementations for the same functionality. To compare these implementations, developers will have to identify the relevant criteria by investigating the decision problem in more detail. This procedure is in line with criterion evaluation and consequential choice of RDM [Zannier, Chiasson, and Maurer 2007]. In consequence, the percentage of NDM for feature requests should be lower than for bug reports. Overall, *a higher* *percentage of NDM for bug reports than for feature requests is expected (hypothesis H3).*

Creation of Coding Table

A coding table was developed to highlight different strategy elements within comments to issue reports. All codes are presented in Table 3.2 with their definition and a corresponding example text

Source	***Code***	***Definition***	***Comment Example***
Naturalistic Decision Making			
[Klein 2008; Zannier, Chiasson, and Maurer 2007]	***Sat*: Satisficing**	Alternatives are better or worse, a satisfactory/workable solution is aspired.	"[...] In the mean time, this user style appears to work around the problem [...]"
[Klein 2008; Zannier, Chiasson, and Maurer 2007]	***SE*: Singular Evaluation**	Further alternatives are not considered or they are evaluated serially.	"It works on today's Nightly, so I guess the problem was fixed in the meantime."
[Klein 2008]	***SA*: Situation Assessment**	Retrieval of attributes that make the current situation comparable to others.	"[...] are you saying that this only happens when the 3 of them are installed at the same time?"
[Klein 2008]	***Mat*: Matching**	The current decision problem or situation is linked to another situation.	"In my case [...] the back button is not activated, so there is no way to restore the session. See bug 928626. [...]"
Rational Decision Making			
[Klein 2008; Zannier, Chiasson, and Maurer 2007]	***Opt*: Optimizing**	Alternatives are right or wrong, the best possible solution is desired.	"[...] Hence I think there is probably a better way to solve this - how about using Firebug's ability to break on DOM events to watch what's going on?"
[Klein 2008; Zannier, Chiasson, and Maurer 2007]	***CC*: Consequential Choice**	Further alternatives are considered, one option is selected from a list of others or options are evaluated concurrently.	"[...] For Beta with irregular builds and minimal functional changes it seems a lot less important. [...]"
[Lipshitz et al. 2001; Zannier, Chiasson, and Maurer 2007]	***CE*: Criterion Evaluation**	Criteria linked to alternatives are considered, reasons/rationales for choosing an alternative are provided.	"[...] Also there are probably cases where what we really need is the state at onload or some other milestone rather than what goes across the wire. [...]"

Table 3.2: Coding Table for Strategy Elements of Decision Making [Hesse, Lerche, et al. 2016]

from a comment. The codes were created according to the principles of content analysis [Mayring 2010]. One code was created for each strategy element of RDM and NDM, which were introduced in Section 2.3. The codes were tested during coding training. Also, the definition for applying a code was refined in the second training session in case that misunderstandings or ambiguities were recognized by the coders.

The textual description of an issue report was not included for coding strategy elements, because reporters were required to use a default structure for writing the report. This structure contains the expected and actual behavior of the software as well as steps to reproduce the actual behavior. In consequence, codes like “Situation Assessment” or “Matching” would have been assigned to many issue reports by default.

Data Selection, Extraction and Cleaning

The study was performed using issue reports from two different branches of the Firefox project, branch Firefox 6 and 27. They will be referred to as *branch 6* and *branch 27*. By using two different branches, it shall be ensured that the analysis of the coded comments is not biased by branch-specific development actions or external events. Branch 6 was developed in 2011, branch 27 in 2014. Both branches provide a suitable number of issue reports for investigation: 642 issue reports belong to branch 6, and 559 issue reports belong to branch 27. It should be noted that the release model in the Firefox project changed in the last years to a rapid release model. Whereas branch 1 to 4 were developed in traditional release cycles with a typical cycle length of one or more years, later branches were developed in shorter release cycles of 6 weeks length. This change affects the total amount of issues contained within one branch as well as the overall variety of development actions performed within one branch [Khomh et al. 2012]. In consequence, we selected only branches developed according to the rapid release model. Therefore, the amount of issues within branch 6 and 27 is comparable. However, the decisions within both branches differ, as branch 27 was developed three years after branch 6.

From the issue tracking system of Firefox [Firefox project 2015], 1201 issues were retrieved in total. All issue reports with their comments were exported and imported as raw data to the spreadsheet program Excel. Within this program, the textual contents of issue reports and comments were formatted visually, so that the initial report could be distinguished from its metadata and its comments. To avoid misinterpretations of the textual contents, artifacts resulting from the export and import process were removed, such as misinterpreted characters and wrongly wrapped lines.

3.2.2 Coding Phase

Coder Training and Intercoder Reliability Assessment

Before all issue reports and their comments were coded, the two coders performed a training to test and harmonize the code assignment to comments. Therefore, two training sessions of coding were performed with two training data sets of 50 randomly selected issue reports. Two benefits were achieved by the training sessions. First, both coders reached a high level of agreement which code to use for which content within an issue comment. Second, after each training session both coders gave feedback on the coding table. Thereby, the codes were further improved, and ambiguities in the code definition were clarified. The initial training session took place before the *Issue Type* coding, the second training was performed afterwards. Each of the two coders assigned codes to the training data sets individually. Both coders examined and discussed each other's codes after a training session. Different interpretations of content and differently coded comments were analyzed and resolved. The codes of the second training session were used to calculate the intercoder reliability . Therefore, the second set of training data had the same proportion of bug reports to feature requests as the final data set. After the second training session, no further training was performed due to the good values for intercoder reliability.

The intercoder reliability measures the similarity of code assignments between different coders. As concrete measure, the intraclass correlation coefficient (ICC) [Shrout and Fleiss 1979] was calculated for the amount of each code per issue for the codes of the second training data set. In detail, the two-way random single measures ICC(2,1) were calculated for agreement between ratings with `icc` out of the `irr` package of `R` [Gamer et al. 2012; R Core Team 2014]. As described by Cicchetti and Sparrow, an ICC value below 0.40 is *poor*, between 0.40 and 0.59 *fair*, between 0.60 and 0.74 *good*, and above 0.74 *excellent*. The ICC values after the second training session are given in Table 3.3. Whereas the ICC results for RDM are already good, the values for NDM strategy elements are excellent. An important reason for this difference is that the total amount of assigned RDM codes was lower than the amount of NDM codes. In consequence, differences between the coders weigh higher for RDM codes. However, the ICC values for RDM strategy elements are still good. This justifies that only one coder coded the final data set of each branch.

Strategy	**NDM**				**RDM**		
Element	*Sat*	*SE*	*SA*	*Mat*	*Opt*	*CC*	*CE*
ICC value	0.96	0.88	0.86	0.78	0.61	0.66	0.61

Table 3.3: Intraclass Correlation Coefficients (ICC) for Strategy Element Codes

Coding of Issue Type and Final Data Selection

The *Issue Type* of all issues from branch 6 and 27 was coded either as *feature* for feature requests, *bug* for bug reports, or *spam* for issue reports without useful content. In detail, one coder coded all issue reports contained within one branch. To determine the appropriate code type of an issue report, its headline and description text were examined by the coder. Bug reports typically contained a description of an error or intended behavior of the software, whereas feature requests consisted of a claim or request to newly develop or extend a functionality of the software.

The coding results for the *Issue Type* were used to select and balance the final data set for coding the strategy elements. Therefore, all issue reports coded as "spam" (about 4% in total) and all issue reports containing no comments (about 9% in total) were excluded. Also, some issues were marked as irrelevant by one or multiple commentators, which lead to a stop in discussion (about 2% in total). These issue reports were also excluded, because no codes could be assigned. In both branches, much more bug reports than feature requests were identified. Thus, all feature reports were coded from both branches in order to have a sufficient number of feature reports for analyzing a possible influence of the *Issue Type* on the percentage of NDM to RDM (RQ3) and their corresponding hypothesis. Then, four times as many bug reports as feature requests were selected for coding the comments to provide a larger overall sample for analyses not considering the *Issue Type*. All bug reports in the final data set were selected randomly from each branch. The final data set consisted of 260 issue reports with 52 feature requests and 208 bug reports. Of these, 100 issue reports originated from branch 6, and 160 reports were derived from branch 27.

Coding of Issue Comments

Each of the two coders analyzed and coded the issue reports and comments of one branch. If a comment contained links to further resources, such as screenshots, code files, or patch notes, the linked resources were also used to determine the appropriate codes for the respective comment.

3.2.3 Analysis Phase

Preparation and Execution of Analysis

Besides the main dependent variables (i.e., the strategy elements of decision making), several additional variables from the issue report meta-data were examined. Variables recording meta-data of issue reports are called *issue dimensions*. By evaluating issue dimensions, it can be checked whether relationships exist between meta-data and the main variables. Major issue dimensions were *Issue Type* with the values "bug" or "feature" and *Branch* with the values "6" or "27". Also, the technical *Component* assigned to each issue report and the total *Number of Comments* per issue report were recorded and analyzed. Moreover, the *Issue Duration* was captured as the difference between the creation date of the last comment and the creation date of the issue report. These issue dimensions describe the environment and context of the decision making documented in the issue reports. In addition, they can be recorded reliably for most issue reports in both branches with a broad range of different values. Thus, these dimensions were selected for analysis.

After recording all issue dimensions and coding the strategy elements, both descriptive and inferential statistics were applied on the data. As descriptive statistics, means (*M*), standard deviations (*SD*), and relative frequencies were used. As inferential statistics, χ^2 tests were used for analyzing relationships between two categorical variables, whereas Pearson correlations coefficients were used to investigate relationships between two metric variables. Also, t-tests and ANOVAs were applied to examine the mean differences between two ore more groups. All tests performed are two-sided. It should be noted that relatively small effects may lead to significant results with high sample sizes. Therefore, an effect size measure was calculated for each test with significant results. Depending on the test type, *r*, η^2, ${\eta_p}^2$ or Cramer's *V* was used. According to Cohen, values of 0.1 indicate *small*, 0.3 *medium*, and 0.5 *large* effects for *r* and Cramer's *V* (df = 1) [Cohen 1988]. For η^2 and ${\eta_p}^2$, 0.01 corresponds to a small, 0.06 to a medium, and 0.14 to a large effect.

3.3 Results and Discussion

In this section, analysis results for the coded data are presented and discussed. First, general findings for all recorded issue dimensions are briefly described to identify potential reasons for differences in the applied decision making strategies between branch 6 and 27. Then, the major results for each of the three research questions are presented. For each research question, the respective results are discussed with respect to requirements for a new documentation approach for decision knowledge. All results of explorative analyses are presented in detail in result tables. Additionally, identified relationships within these analyses are summarized in graphs for discussion. The graphs highlight identified relationships and also provide an overview on relationships which have not been found to be significant in this study. Thus, such insignificant relationships might serve as a subject for further studies. However, for explorative analyses, interpretation and discussion of relationships are focused on those of large and medium effect size.

3.3.1 Results for Issue Dimensions

In total, 36 different values for *Component* were identified and grouped into three categories: issue reports may be related to specifically named components (such as “location bar”; category is called “specific”, n=114), to unnamed but defined components (category is called “untriaged”, n=80), or to general components (category is called “general”, n=66). The *Number of Comments* ranged from 1 to 48 with a mean of 5.53 comments per issue report. For all investigated issue reports, the *Issue Duration* ranged from 65 seconds to more than 4 years. The mean *Issue Duration* is 268 days. It should be noted that Firefox’ official timeline for branch development does not necessarily fit the actual relation between issue reports and branches. For instance, 28% of all investigated issue reports had a starting time before the initiation of their respective branch. This indicates that some functionality and error discussions were started earlier, but later became major topics for their respective branch. Also, 76% of all issue reports exceeded their branch development time. This indicates that often issue reports were not entirely solved during development of their associated branch. Then, the issue status “closed” or “resolved” is a good indicator for an end of discussion. However, discussion on these issue reports may have ended already, even if this status was not set. This was observed for many issues with longer duration and no such status. But discussion may also start again in case former solutions do not work out or do not address changed environmental or system conditions appropriately. These uncertainties should be reflected when interpreting the results for *Issue Duration*.

The relationships between all investigated issue dimensions as the result of an explorative analysis are summarized in Table 3.4. The table depicts significant relationships according to their related effect size value as “no”, “small”, “medium”, or “large”.

Issue Dimensions	**Issue Type**	**Branch**	**Component**	**Issue Duration**	**Number of Comments**
Issue Type	–	no	small	small	no
Branch		–	large	large	small
Component			–	large	small
Issue Duration				–	small
Number of Comments					–

Table 3.4: Summary of Relationships between Issue Dimensions

Due to the selection of the final data, *Issue Type* and *Branch* are completely independent. In contrast, *Issue Type* is related to *Component* ($\chi^2[2] = 20.54, p < 0.001, V = 0.28$): The category “specific” for *Component* predominates the other two categories stronger for features than for bugs. In addition, the value for *Issue Duration* is significantly higher for bugs ($M = 296.16, SD = 366.72$) than for features ($M = 156.71, SD = 297.87, t[258] = 2.54, p < 0.05, r = 0.16$). Significant relationships were also found between *Branch* and *Component* ($\chi^2 = 90.39, p < 0.001, V = 0.59$). In branch 6, mostly issue reports with “general” (54%) and “specific” component categories (42%) are given. In branch 27, “untriaged” (48%) and “specific” (45%) component categories are predominant for issue reports. Also, significant effects of *Branch* on the *Issue Duration* ($t[258] = 16.23, p < 0.001, r = 0.71$) and the *Number of Comments* ($t[258] = 2.01;\ p < 0.05, r = 0.12$) were found. In detail, the average *Issue Duration* in branch 6 is longer (in days) (*M* = 589.38, *SD* = 395.59 vs. *M* = 67.57, *SD* = 76.19), and the *Number of Comments* is higher for branch 6 than for branch 27 (*M* = 6.42, *SD* = 6.51 vs. *M* = 4.97, *SD* = 5.06).

Moreover, ANOVAs uncovered significant effects of *Component* on *Issue Duration* ($F[2, 257] = 49.48, p < 0.001, \eta^2 = 0.28$) and the *Number of Comments* ($F[2, 257] = 4.60, p < 0.05, \eta^2 = 0.03$). Here, discussions on issue reports addressing "general" components took longest (in days) (*M* = 565.98, *SD* = 408.42) and comprised the highest number of comments (*M* = 6.82, *SD* = 7.48), followed by "specific" (*Issue Duration*: *M* = 237.69, *SD* = 336.21; *Number of Comments*: *M* = 5.81, *SD* = 5.28), and "untriaged" (*Issue Duration*: *M* = 66.22, *SD* = 65.03; *Number of Comments*: *M* = 4.06, *SD* = 4.10). Finally, also *Issue Duration* and the *Number of Comments* correlate positively ($r = 0.24, p < 0.001$): a higher number of comments is related to a longer issue duration.

3.3.2 Discussion of Results for Issue Dimensions

An overview of the relationships between the different issue dimensions according to Table 3.4 is given in Figure 3.2. Large relationships between *Branch, Component,* and *Issue Duration* probably were found because the age of both branches differs significantly. Of course, some restrictions for the interpretation of *Issue Durations* apply, as mentioned before. However, it is likely that issue reports in branch 6 are less structured than in branch 27, for which development started later. So, reporters and

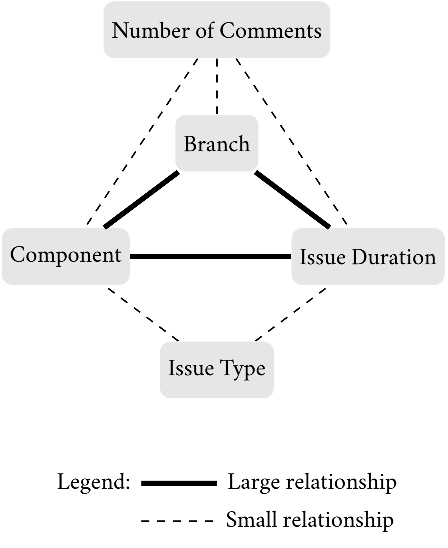


Figure 3.2: Relationships between Issue Dimensions

developers could work with many existing features and the related knowledge from earlier branches within branch 27. This interpretation is supported by the finding that mostly "general" was found as category for *Component* in branch 6 in comparison to mostly "specific" for branch 27. Unspecific issue reports also might be one important reason for the longer issue duration for branch 6, because these reports required more discussion. This is in line with the slightly higher *Number of Comments* for branch 6 (M = 6.42) than for branch 27 (M = 4.97). Also, developers would then struggle to resolve and close bug reports, when they cannot easily structure them to identify the described problem and criteria to assess potential solutions. This is backed up by the longer *Issue Duration* for bug reports than for feature requests.

However, this interpretation does not necessarily imply that the decision making behavior of developers differed significantly between branch 6 and 27. On the one hand, branch 27 appears to be better structured, because more issue reports are assigned to specific components. Thereby, they match better with well-structured decision problems, so the application of RDM is more likely in this branch. On the other hand, issue reports in branch 27 were built upon more project knowledge and experience of the developers due to the longer project duration Firefox had reached when the branch started. In consequence, developers might also tend to use more NDM due to their increased experience with former decision situations from earlier branches that could be matched to current ones. In case the percentage of NDM to RDM strategy elements differs significantly between the branches, one of these explanations may be reasonable.

3.3.3 Results for RQ1: Dominance of NDM

In total, 98.16% of all codes were NDM strategy elements, whereas only 1.84% RDM codes were set. The mean number of NDM codes was compared with the mean number of RDM codes in a t-test for dependent samples. Thereby, a significantly higher number of NDM strategy elements (M = 5.47, SD = 5.94) than RDM strategy elements (M = 0.18, SD = 0.57, $t[259]$ = 15.06, $p < 0.001$, r = 0.68) was observed. This clearly shows the dominance of NDM over RDM in the investigated comments to issue reports and confirms hypothesis H1.

3.3.4 Discussion of Requirement A: Documentation of RDM and NDM Decisions

The confirmation of H1 highlights that NDM plays an important role in decision making of developers in practice. For the investigated data, NDM dominated RDM clearly in both branches. This result challenges the common understanding that developers and particularly designers should apply an engineering approach that is driven by RDM [Davide Falessi, Cantone, Kazman, et al. 2011]. RDM may be the kind of decision making strategy desired and advised by researchers and academia, because it allows for determining the optimal solution. This is often the gold standard for decisions from a theoretical perspective. However, in practice, not only a second kind of decision making strategies is used, but it also outnumbers by far the actual percentage of RDM. The results of previous qualitative studies already pointed in this direction. Nevertheless, this study is the first to confirm quantitatively that developers use large amounts of NDM in practice.

Also, this finding implies that NDM should be considered more closely when creating and providing documentation support for decisions. Developers in the Firefox project were not required to document their decisions in a guided or prescribed way within their comments. Thus, it is likely that they documented the decision making they actually applied, as they could have added more RDM documentation at any time. That they did not add more RDM may have two different reasons: On the one hand, developers might have found the given NDM documentation to be sufficient. Then, more documentation on RDM was probably not requested for most of the decisions related to issue reports. In consequence, RDM should not be enforced during documentation by default. On the other hand, developers might have wanted to document more RDM, but were hindered by high documentation effort. Then, RDM documentation needs better support.

It should be noted that the presented study did not evaluate the development outcome of the investigated issue reports and their related decisions. Thus, the result for hypothesis H1 does not imply that the usage and documentation of NDM leads to better decision outcomes than using and documenting RDM. However, the finding clearly indicates that both RDM and NDM should be supported during documentation, as both are used by developers in practice. In consequence, the documentation approach developed in this thesis should address the documentation of both RDM and NDM. In detail, this requires specific documentation entities for decision knowledge resulting from NDM. These entities should be captured in structures supporting a mix of RDM and NDM, as different developers may work on the same decision, but apply different decision making strategies.

3.3.5 Results for RQ2: Distribution of Strategy Elements

The mean frequencies of each strategy element per issue are presented in Table 3.5. In addition, the mean frequency of each strategy element per issue was divided by the mean total number of strategy elements per issue. The resulting proportions are also shown in Table 3.5. In line with the findings for RQ1, these proportions highlight a clear predominance of NDM strategy elements over RDM strategy elements according to the proportions. It should be noted that the number of codes to an issue report depends significantly on the *Number of Comments* of the respective issue

Strategy Element	***M* (*SD*)**	**Proportion (in %)**	**Sat**	**SE**	**SA**	**Mat**	**Opt**	**CC**	**CE**
Satisficing (Sat)	0.63 (0.99)	12.15	–	0.02	-0.23***	-0.10	0.06	0.01	-0.09
Singular Evaluation (SE)	0.82 (1.41)	13.80		–	-0.26***	-0.25***	0.14*	0.17**	0.04
Situation Assessment (SA)	2.88 (3.89)	47.88			–	-0.50***	0.00	0.00	-0.07
Matching (Mat)	1.13 (1.47)	24.33				–	0.01	0.01	0.01
Optimizing (Opt)	0.03 (0.19)	0.35					–	0.14*	0.21***
Consequential Choice (CC)	0.06 (0.26)	0.72						–	0.21***
Criterion Evaluation (CE)	0.08 (0.34)	0.76							–

Legend: * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table 3.5: Means and Standard Deviations of the Frequency of each Strategy Element per Issue Report, Proportions of Strategy Elements and Intercorrelations between the Frequencies of Strategy Elements per Comment [Hesse, Lerche, et al. 2016]

report (r = 0.94, $p < 0.001$). In consequence, the proportion of each code per comment instead of the absolute frequencies was analyzed and reported for the correlational analyses in Table 3.5. Thereby, a positive correlation sign indicates that the higher one variable, the higher is the other variable. Consequently, a negative correlation reveals that the higher one variable, the smaller is the other one. For instance, the higher the proportion of *Satisficing (Sat)* codes, the smaller is the proportion of *Matching (Mat)* codes (r = -0.26, $p < 0.001$). In the remainder of this chapter, the *Percentage of NDM* as a combined measure of the sum of all NDM codes divided by the total number of all codes per issue report is reported and discussed. This facilitates the comparison of amounts of NDM and RDM codes per issue report. In total, 6 significant intercorrelations were found. Two of them were positive: the appearance of *Optimizing (Opt)* and *Consequential Choice (CC)* were positively correlated with *Criterion Evaluation (CE)*. Furthermore, four negative correlations were identified: a higher amount of *Satisficing (Sat)* and *Singular Evaluation (SE)* was found, when the amount of *Situation Assessment (SA)* was smaller. Also, higher amounts of *Singular Evaluation (SE)* and *Situation Assessment (SA)* were observed, when less *Matching (Mat)* codes were set.

3.3.6 Discussion of Requirement B: Iterative Decision Documentation

The intercorrelations between the frequencies of strategy elements per comment presented in Section 3.3.5 uncover relationships between strategy elements, as depicted in Figure 3.3. Significant relationships were found only between strategy elements belonging to the same kind of decision making strategy. Most importantly, a large negative correlation between *Situation Assessment* and *Matching* was identified. Two different explanations may be relevant for this finding. First, an assessment of the decision situation might be documented more often if developers cannot match the given situation to former ones in order to identify potential solutions. Second, it is also possible that more matching of decision situations is documented when the developers feel they have clarified the decision situation sufficiently to compare it to others.

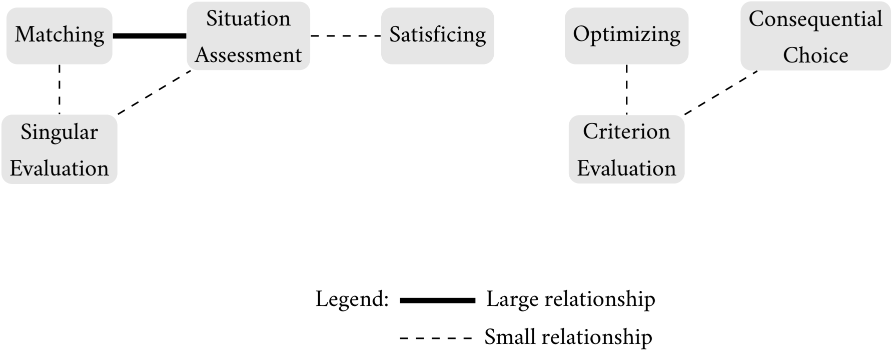


Figure 3.3: Relationships between Strategy Elements

An iterative documentation behavior for decision knowledge by developers is already reflected by the comments added by developers over time. Thereby, the developers document the available decision knowledge with adaptions and extensions incrementally over time. Whereas this behavior could be leveraged by the comment functionality of the issue tracking system, it would have also been possible for developers to document all decision knowledge at once within their issue reports. However, the distributed decision making within large open source projects and the broad nature of the upcoming decisions increase the likelihood of an iterative decision making behavior [Ko, DeLine, and Venolia 2007], and, therefore, also the need of developers to document newly emerging decision knowledge in an iterative way.

The identified relationships between the different strategy elements indicate that this iterative decision making and documentation behavior does apply for both kinds of decision making strategies and their strategy elements. However, the actual behavior differs. On the one hand, the documentation of RDM strategy elements increases the documentation of other RDM elements in further comments. Examples are the positive correlations of *Criterion Evaluation* with *Optimizing* and *Consequential*

Choice. So, the documentation of RDM elements seems to encourage developers to document further RDM knowledge within different RDM strategy elements. On the other hand, the documentation of NDM strategy elements decreases the documentation of other NDM elements. For instance, negative correlations were found between *Situation Assessment* and *Matching* or between *Singular Evaluation* and *Situation Assessment*. Thus, developers appear to document less NDM knowledge in different NDM strategy elements when a set of NDM strategy elements of a particular type is already given. Moreover, no correlations were found between the documentation of strategy elements belonging to different kinds of decision making strategies. In consequence, the mix of both strategies within one decision appears to be less likely for documentation, although it is possible in practice [Zannier, Chiasson, and Maurer 2007].

These characteristics result in an important requirement for the documentation approach presented in this thesis: The approach should provide iterative documentation structures for decision knowledge, which can be used consistently for each kind of decision making strategy. For RDM, the documentation of knowledge resulting from multiple strategy element types should be supported, whereas documenting knowledge originating from NDM requires support for larger amounts of knowledge resulting from particular strategy element types. The mix of both kinds of decision making strategies during documentation should be supported due to results from existing studies (e.g., as described by [Zannier, Chiasson, and Maurer 2007; Tang, Aleti, et al. 2010]), but was not found to be significant within this study.

In addition, the overall amounts of strategy elements provide an important insight on the knowledge actually documented by developers. These amounts are highest for *Situation Assessment* and *Matching*. Both elements concern the decision situation, which is characterized by information on the decision problem and its context [Zannier, Chiasson, and Maurer 2007]. In consequence, the documentation approach should provide differentiated entities to capture and represent knowledge on decision problems and context as precisely as possible.

3.3.7 Results for RQ3: Differences between Feature Requests and Bug Reports

An ANOVA was conducted with *Issue Type* and *Branch* as between-subject factors and the *Percentage of NDM* as dependent variable. The branch was included in order to check if the pattern of results is similar for the two branches. This 2 (bug vs. feature) by 2 (branch 6 vs. branch 27) ANOVA yielded a main effect of *Issue Type* (F[1,256] = 24.10, $p < 0.001$, $\eta_p{}^2 = 0.09$). The *Percentage of NDM* for bugs (M = 0.99, SD = 0.03) is significantly higher than for features (M = 0.94, SD = 0.13). This finding confirms H3. In addition, there is no main effect of *Branch* (F[1,256] = 1.31, p = 0.25). Thus, the two branches do not differ regarding the percentage of NDM elements. Also, there is no interaction of *Issue Type* and *Branch* ($F < 1$) which reveals that the pattern (higher percentage of NDM elements for bugs than for features) is independent of the branch (see also Figure 3.4). However, it should be noted that the observed difference between feature requests and bug reports is significant, but it is not large in size as NDM clearly predominated RDM.

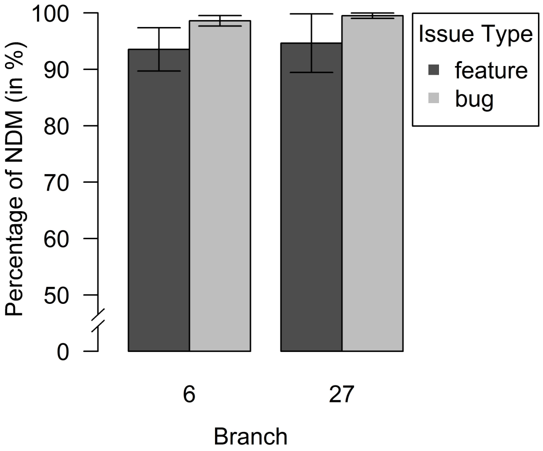


Figure 3.4: *Percentage of NDM* as a Function of *Branch* and *Issue Type*. Bars represent the means and error bars the 95% confidence intervals of the means.

3.3.8 Discussion of Additions to Requirement A

The confirmation of H3 is in line with different decision situations and thereby different development activity structures for feature requests and bug reports. For feature requests, developers document slightly more RDM, as they exploit the design space to initially develop features. For bug reports, developers tend to document more NDM when they carry out established development tasks to fix errors. Both aspects might indicate different levels of reflection for decisions on feature requests and bug reports by developers. It is possible that more RDM elements for decisions on new features are documented to make these decisions more comprehensible to all developers. This reasoning is supported by the small-sized correlation between *Issue Type* and *Issue Duration*, where a longer *Issue Duration* for bug reports than for feature requests was observed. Thus, feature requests appear to consist of better substantiated solutions, as developers need less time to discuss them.

However, the relation between the type of decision and the decision making performed by developers is not yet comprehensive. Further investigation is required to understand how different project setups and settings influence this relation. For instance, the observed difference between feature requests and bug reports might be more pronounced in projects with higher degrees of uncertainty and fewer experts involved. Based on the current finding, the requirement for documenting RDM and NDM knowledge can be refined. At least, it should be possible to capture the type of decision when documenting decision knowledge. Therefore, the type of decision should be reflected within the overall decision description, and may be refined by adding specific knowledge entities describing the decision problem.

3.4 Threats to Validity

Four different categories of threats to validity have to be considered for this study according to Runeson et al. [Runeson et al. 2012]. These categories are described and discussed in the following paragraphs.

Internal Validity
Threats to internal validity concern the correlation between the investigated factors and other factors [Runeson et al. 2012]. All retrieved issues were associated to their respective branch manually by reporters or developers. Thus, associations between issue reports and branches might have been incorrect due to inconclusive or misleading issue descriptions. This problem was addressed by the coders, as they checked the content of all issue descriptions for determining the *Issue Type*. Also, issue reports belonging to one of the investigated branches might have been missed during extraction, if they were not associated properly to their respective branch within the issue tracking system. Therefore, subsets of all issue reports were selected for investigation randomly from both branches. Nevertheless, the actual ratio of bug reports to feature requests might deviate from our calculated ratio. An important reason are potentially different report styles according to the role of their reporters. For instance, it is possible that within issue reports Firefox developers focus on other aspects than end users. In consequence, also issue discussions within the comments could have been determined by the reporter. This might affect the issue type, dimensions, and coding results of the comments. However, the guidelines of the Firefox project for its issue tracking system mitigated this threat, because every new issue report is required to provide a set of standardized description contents (i.e., for instance steps to reproduce, actual results, expected results). Finally, not all information relevant for a decision might be documented within the issue tracking system. Instead, such information could be retrieved from other sources for documentation, such as mailing lists or forums. However, if these sources would have been used extensively by project members, they would have likely referenced or copied information from these sources in order to make it accessible to others. Some references to such external sources, like links to mailing lists or code revisions, were found in issue comments and considered during coding.

Construct Validity
Threats to construct validity concern any gaps between intended and actual observations of the

researchers [Runeson et al. 2012]. This study does not assess the quality and actual outcome of the documented decisions, but only focuses on the documented decision making. Also, only documented strategy elements are covered in this investigation. In contrast, developers may stick to one kind of decision making, but document their thinking according to another one. For instance, documenting a claim for a single solution might be the result of a consequential choice between multiple solution options. Whereas such a situation might happen frequently for individual developers, the shared documentation is less comprehensive for others due to the omitted information. Therefore, a team of developers of an open source project is more likely to document the actual thinking to avoid misunderstandings and diverging interpretations of the given documentation. Thus, this study is focused on documented decision making only. However, there is the risk that the coding tables applied within the study could have identified something else than strategy elements or might not have covered the investigated strategy elements properly. This threat was addressed by creating the coding tables based on a comprehensive overview of theories and fundamental approaches for decision making strategies.

Reliability Validity

Threats to reliability validity concern the degree to which data and analyses of a study are dependent on specific researchers [Runeson et al. 2012]. All codings were performed manually by two human coders. Thus, codes might have been missed or were applied inappropriately. Also, the two coders coded one branch each. In consequence, discrepancies in applying the codes could have exist between the coders, but were not revealed. Both threats were mitigated by performing two rounds of coding training, where both coders coded the same issues independently, and discussed any observed differences. The calculated values for the intercoder reliability, which range from good to excellent, demonstrate the positive effect of this measure.

External Validity

Threats to external validity concern the degree to which the results of our study can be generalized [Runeson et al. 2012]. The investigated issue reports originate from only one open source project. In consequence, the presented findings might not be comparable to results acquired from other open source projects. However, the Firefox project is a well-established study subject in many other issue-related studies [Ko and Chilana 2011; Zaman, Adams, and Hassan 2011; Souza, Chavez, and Bittencourt 2014]. Thus, this project was also chosen for the presented study, as the results should be at least applicable to other open source projects with similar size and team distribution. Next, issue reports within branches 6 and 27 might not be representative for issue reports originating from other branches of the project. This problem exists for all subsets of branches, unless every branch is investigated. This issue was mitigated by choosing two branches to cover both an early and a late project stage. In consequence, it is not likely, that significant shifts or adaptions during issue documentation have been missed. Only a small subset of all issues registered in the issue tracking system could be analyzed, because more than 173k issue reports existed in April 2017. Nevertheless, the issue sample size of $N = 260$ is large from a statistical point of view, so that the identified relationships are well grounded.

4 State of the Art for Decision Knowledge Documentation

In this chapter, the scientific state of the art for documentation of decision knowledge is investigated based on the literature. Therefore, a literature review is performed to identify and compare current approaches with their knowledge models and tools for documenting decision knowledge. A previous version of this review was published in [Paech, Delater, and Hesse 2014]. First, *study foundations*, like the goal of the review and related studies, are presented. Second, the *research process* of the review with the definition of research questions, search terms, and search procedures is introduced. Third, the *results* of the review are presented and discussed with regard to resulting requirements for the documentation approach of this thesis. Finally, the *threats to validity* for the literature review are described and discussed.

4.1 Study Foundations

In this section, the *study goal and search focus* of the performed literature review are described. Also, *related studies* brought up by the presented review are discussed with regard to their search focus, results, and limitations.

Study Goal and Search Focus

The requirements derived from the results of the Firefox study (cf. Section 3.3) facilitate a documentation of decisions according to different DM strategies applied by developers. However, the study also indicated that developers do not make important decisions individually. Instead, they typically collaborate to refine and adapt decisions over time during different development activities. Thus, an investigation is required on how decision knowledge can be captured, structured and linked to related artifacts across these activities.

Various approaches and tools already exist to support the documentation of decision knowledge, for instance during the negotiation of requirements or the specification of an architecture. All these approaches apply specific models to structure decision knowledge. In addition, they often come with specialized tools to support capturing, linking, and exploiting decision knowledge according to a proposed model. By identifying major contributions of current approaches regarding their knowledge models and tools, gaps in covering the collaborative documentation of decisions can be examined. Thereby, requirements are derived to address research goal 2 (cf. Section 1.1). This literature review aims to provide such an overview on these approaches and tools with resulting requirements for the documentation approach proposed in this thesis.

Similar to the goal of the presented Firefox study (cf. Section 3.1), also the goal for the literature review is described using the Goal Question Metric (GQM) approach [Basili, Caldiera, and Rombach 1994]. The goal is given within the structured template of GQM in Table 4.1.

GQM Template	**Goal Description**
Determine	existing approaches and tools for decision knowledge documentation
with respect to	supported knowledge structures, links, and usage purposes for documented knowledge
for the purpose of	improving the documentation for decision knowledge elements
in the context of	development decisions during different development activities
from the viewpoint of	researchers.

Table 4.1: Description of Study Goal with GQM

In order to achieve this goal, the search for relevant publications focuses on automatic search engines of large scientific publishers, selected individual venues, and a manual search. The goal of the search is to uncover publications presenting a documentation approach for decision knowledge with respect to requirements engineering, design, and implementation. However, the automatic search was broadened to cover also other development activities to ensure that no relevant references were missed.

Related Studies

Several other reviews and mapping studies on decision documentation were uncovered during the search process. Two different groups of related studies can be distinguished: studies focusing on *concepts and knowledge models* for documentation approaches, and studies comparing a selection of *tools* for documenting decision knowledge.

Regarding concepts and knowledge models, two mapping studies were presented in [Li, Liang, and Avgeriou 2013] and [Tofan et al. 2014]. Li, Liang, and Avgeriou investigated 55 papers on knowledge-based approaches for enhancing and documenting software architecture. Among others, these approaches proposed to support architecturs during knowledge capture and representation, sharing, recovery, and reuse [Li, Liang, and Avgeriou 2013]. The study found that most papers covered knowledge capture and representation (42 out of 55), whereas only few approaches address knowledge recovery (3 out of 55) [Li, Liang, and Avgeriou 2013]. However, the study is concerned with architectural knowledge in general, within which decision knowledge is covered only implicitly. Thus, this study does not provide insights for documentation of decision knowledge in particular. In contrast, Tofan et al. identified 120 papers concerned with architectural decisions. The authors examined the papers to answer six different research questions, within which the first research questions explicitly asked for the documentation of architectural decisions. Within the 120 papers related to documentation in principal, even 26 approaches with individual tool support were found [Tofan et al. 2014]. Nevertheless, the study of Tofan et al. focuses on architectural decisions. Knowledge on decisions within other development activities, such as requirements engineering or implementation, was not investigated. In consequence, differences in knowledge structures, links, and usage purposes for decision knowledge originating from these activities remain uncovered. Furthermore, several literature reviews on documenting decision knowledge already exist. Ding et al. address decision documentation as part of their literature review on knowledge-based approaches for software documentation. Therefore, they examined 60 different publications for cost, benefit, and quality attributes of software documentation. In general, the study finds several attributes to be important for documented decisions, such as consistency or correctness [Ding et al. 2014]. However, no detailed investigation of decision documentation is performed, so that the identified quality attributes for documented decisions are not related to means to facilitate them within decision documentation. Thus, the study does not provide insights on how decision documentation may be improved in order to address the identified quality attributes. The literature review of Weinreich and Groher examines 56 approaches for architectural knowledge management and their empirical investigation. An important finding of their study is that capturing architectural knowledge is still an unsolved problem [Weinreich and Groher 2016]. In addition, maintaining and sharing architectural knowledge is addressed by only a few approaches [Weinreich and Groher 2016]. As approaches for architectural knowledge management typically include the management of knowledge on architectural decisions [Jansen and Bosch 2005], these findings are likely to apply also for decision documentation approaches. Nevertheless, documented decisions are not covered explicitly within the study. Alexeeva, Perez-Palacin, and Mirandola focus on investigating approaches for documenting design decisions. They examine 96 papers by six different classification dimensions [Alexeeva, Perez-Palacin, and Mirandola 2016]. In detail, documentation approaches are examined regarding their goal, formalisation, extent of decision capture, context, tool-support, and evaluation. Based on this examination, different open questions for design decision documentation are presented [Alexeeva, Perez-Palacin, and Mirandola 2016]. Whereas the classification dimensions for documentation approaches are a major contribution of the study, the investigation results for each approach are not described in detail, but only summarized quantitatively and highlighted by examples. Moreover, the study focuses on design decisions, and covers decisions during other activities only implicitly within the context dimension. Thus, it does not cover the state of the art for decision documentation in general.

In addition to the discussed literature studies, three individual studies comparing different tools for knowledge documentation were found. First, Ali Babar, Boer, et al. examine a selection of eight different tools for architectural knowledge management from academia and industry with regard to core functionality. The authors observe a gap between theories on knowledge management in a scientific context and their practical implementation within industrial tools [Ali Babar, Boer, et al. 2007]. For instance, scientific tools focus on codification of knowledge, whereas industrial solutions mostly support knowledge personalization [Ali Babar, Boer, et al. 2007]. However, the study does not investigate the described tools in detail. In addition, the limited number of tools does not allow for more general conclusions. Second, Tang, Avgeriou, et al. present an extensive comparison of five different tools for architectural knowledge management, which focus on knowledge from architectural decisions. The tools are compared regarding their support for knowledge representation, relationships between knowledge elements, and activities to maintain and share documented knowledge [Tang, Avgeriou, et al. 2010]. The study presents detailed results for each tool, but again investigates only a small number of existing tools. Third, a recent study was presented by Capilla, Jansen, et al. to reflect the last decade in architectural knowledge management. Therefore, three different generations of knowledge management tools are identified and described. The study discusses selected attributes of each approach, such as supported knowledge structures. The authors highlight the evolution of tools beginning with a focus on knowledge representation and capturing, continuing with knowledge sharing, and finally addressing the collaborative creation and documentation of architectural knowledge [Capilla, Jansen, et al. 2016]. This goal of the third generation of tools fits very well with both research goals of this thesis. However, no structured comparison of the described tools according to defined criteria is performed.

Overall, a variety of studies on approaches for documenting decision knowledge has been published. Altogether, these studies indicate a growing amount of approaches and tools for documenting knowledge, whereas the explicit focus on decision knowledge within the individual approaches and tools varies. Also, there is a focus on knowledge management and decision documentation during software architecture and design. In consequence, an overview on the state of the art for decision documentation is still required to compare different approaches and tools regarding (i) their representation for decision knowledge, (ii) links between knowledge elements within these representations, and (iii) the actual capturing and usage of decision knowledge during different development activities.

4.2 Research Process

In the following paragraphs, the research process for the conducted literature review is explained. An overview of this process is depicted in Figure 4.1. In detail, the guidelines of Kitchenham and Charters were applied for preparing and executing the search, as well as for documenting the results of the review [Kitchenham and Charters 2007]. However, the presented review was not strictly conducted as a systematic literature review, as the literature was assessed by one researcher only.

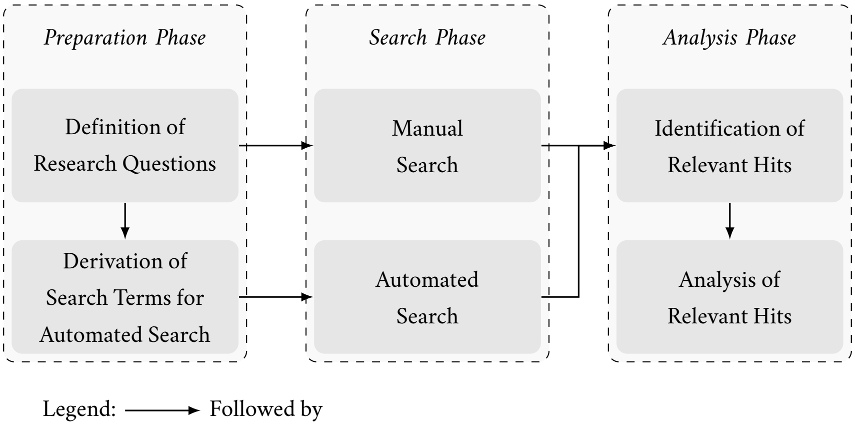

Preparation Phase
Definition of Research Questions
Derivation of Search Terms for Automated Search
Search Phase
Manual Search
Automated Search
Analysis Phase
Identification of Relevant Hits
Analysis of Relevant Hits
Legend:
Followed by

Figure 4.1: Overview of the Literature Review Process

4.2.1 Preparation Phase

First, the literature review was prepared by *defining the research questions* to be answered. Then, appropriate *search terms for automated search* of relevant publications were derived based on the research questions.

Definition of Research Questions

For the literature review, the following research questions (abbreviated as *RQ*) were defined with regard to research goal 2, the investigation of decision documentation during different development activities, and its open questions (cf. Section 1.1):

RQ1: How can decision knowledge be structured?

RQ2: Which tools are used to capture decision knowledge in relation to further knowledge?

RQ3: How is decision knowledge used?

In detail, RQ1 aims to investigate, which decision knowledge elements belong to a core set of elements for documenting decision knowledge. Therefore, it is necessary to examine decision knowledge elements and their related structures proposed by current documentation approaches. However, this requires to also investigate the links between these elements, as linking knowledge elements is an important mechanism for structuring them (cf. Section 2.4). Thereby, RQ1 contributes to the first open question for research goal 2 by uncovering which knowledge elements should be contained in a core set for documentation during requirements engineering, design and implementation. In contrast, RQ2 aims to create a deeper understanding of the tools for documenting decision knowledge. Regarding the currently available tools for documentation, capturing decision knowledge is an important functionality according to the findings of the related studies presented in the previous section. In particular, this requires capturing decisions in relation to development activities from which they originate. Therefore, also relations between decision knowledge and other development knowledge and artifacts need to be examined. Moreover, captured knowledge should be explored and used by developers, which is addressed by RQ3. Together, RQ2 and RQ3 contribute to the second open question for research goal 2 by providing insights on specific capturing mechanisms for decision knowledge during different development activities, and how this knowledge could be used by developers.

Derivation of Search Terms for Automated Search

In order to perform an automated search to investigate the presented research questions, search terms for the automated search were derived. These terms are given in Table 4.2.

Search Term	Definition of Search Term	Restriction
Term 1a	"Decision knowledge" OR "decision rationale" OR "decision motivation" OR "decision intention"	Title, abstract, keywords
Term 1b	"Decision knowledge" OR "decision rationale" OR "decision motivation" OR "decision intention" OR "design decision" OR "requirements decision" OR "implementation decision" OR "test decision" OR "maintenance decision" OR ("management decision" AND "software project")	Title, abstract, keywords
Term 2	Model OR representation OR information OR capture OR link OR benefit OR advantage	Title, abstract, keywords

Table 4.2: Overview of Search Terms for the Literature Review [Paech, Delater, and Hesse 2014]

Term 1a and term 1b were used to identify literature related to decision knowledge. Such literature may address documented knowledge directly, drivers for decisions, or decisions during different development activities. Therefore, various synonyms for decision knowledge and kinds of decisions, such as "decision rationale" or "design decision", were used alternatively in these terms. If term 1a yielded not enough hits for a particular search engine, term 1b was used instead, as it provided more alternative search strings for decision knowledge. However, searching solely for “decision” or “design decision” produced far too many results. Thus, term 2 addressed the specific contents required to answer the research questions. The literature was searched for contributions to knowledge models or representations as well as for capturing and linking concepts for decision knowledge. Also, approaches using unstructured representations were covered by adding “information” to the search term. Furthermore, the usage of decision knowledge was described more specifically by “benefit” and “advantage”. During the automated search, a combination of term 2 and either term 1a or term 1b was applied to the search request.

4.2.2 Search Phase

The actual literature search was performed twice. The first search was carried out in October 2012, as reported in [Paech, Delater, and Hesse 2014]. The second search was performed in May 2017 to enrich the previous search results with relevant literature published since the first search. In both searches, an *automated* and a *manual* search was carried out.

Automated Search
During the automated search, the sources *IEEE* [*IEEExplore* 2017], *ACM* [*ACM Digital Library* 2017], *SpringerLink* [*SpringerLink* 2017], and *ScienceDirect* [*Elsevier ScienceDirect* 2017] were searched using the defined search terms. In addition, also the *International Journal of Software Engineering and Knowledge Engineering* (IJSEKE) [*The International Journal of Software Engineering and Knowledge Engineering* 2017] was included in the search, because this venue is focused on publishing knowledge-based approaches for research problems in software engineering. Table 4.3 shows the search requests within each source.

Sources	**Automated Search Request**	**Restriction**
IEEE, ACM	Term 1a AND Term 2	None
SpringerLink, ScienceDirect	Term 1a AND Term 2	Category “Computer Science”
IJSEKE	Term 1b AND Term 2	None

Table 4.3: Search Requests Applied for Different Sources

In general, IEEE, ACM, and IJSEKE focus on publications belonging primarily to the discipline of computer science. Therefore, it was not necessary to apply any search restrictions within the search requests for these sources. In contrast, SpringerLink and ScienceDirect offer a broad variety of publications from other disciplines, such as engineering, or medicine. Thus, the search for these two sources was limited to the field of “Computer Science”.

Manual Search
The manual search complemented the automated search by assessing papers given in the references of relevant hits by their title and abstract. Based on the hits of the automated search and the existing related studies, approaches for documenting design decisions were explored in more detail with this search. In addition, other topics related to decision documentation, such as decision science or management, were partially addressed within the manual search. Whereas relevant hits for these topics contributed to other chapters of the thesis, e.g. to enrich the background on decision making (c.f. Section 2.3), these hits were not considered within the results of the literature review.

4.2.3 Analysis Phase

Finally, all hits retrieved by automated and manual search from the two literature searches were assessed for relevant hits. Therefore, relevant hits were *identified* according to standardized criteria, and afterwards *analyzed* for their contribution to the results.

Identification of Relevant Hits
An overview of the amounts of hits from the different searches and exclusion rounds is given in Figure 4.2. In total, the automated search in 2012 provided 520 hits, whereas the automated search in 2017 added another 200 hits. Also, the manual search produced 27 further hits. Based on these hits, two exclusion rounds were performed. Hits were considered relevant if they explicitly addressed decision knowledge and its content structures, links, capturing mechanisms, documentation tools, or usage of decision knowledge. Hits were not considered if their full text was not retrievable, or if hits were duplicates. In the first round, all hits were examined according to their title, and their

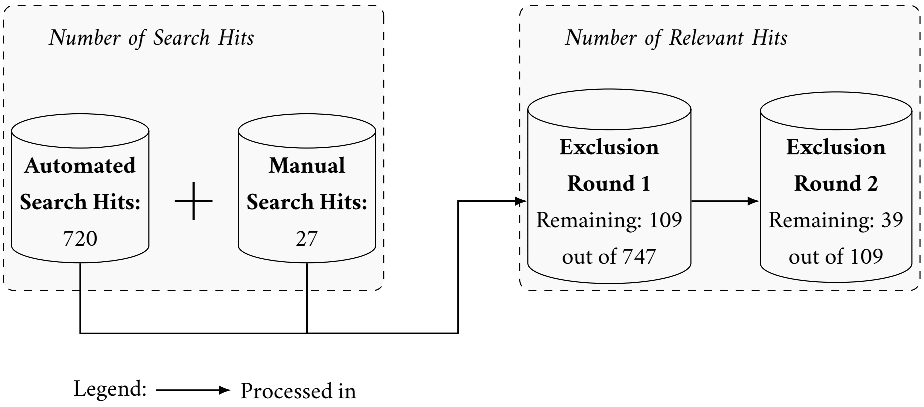

Number of Search Hits
Automated Search Hits: 720
+
Manual Search Hits: 27
Number of Relevant Hits
Exclusion Round 1 Remaining: 109 out of 747
Exclusion Round 2 Remaining: 39 out of 109
Legend:
Processed in

Figure 4.2: Search Hits and Relevant Hits from Searches Performed in the Review

abstracts, if necessary. Thereby, 109 hits were identified as potentially relevant. In the second round, all abstracts were read and the full text of promising publications publications was examined carefully for contributions. This revealed 39 publications to be relevant for the review.

A detailed overview of all publications resulting from exclusion round 2 can be found in Appendix A. In detail, the automated search using term 1a produced 22 relevant publications, term 1b yielded 2 relevant sources, and manual search contributed another 15 relevant publications. Documentation approaches and tools are addressed by 31 relevant publications, whereas 8 sources are reviews and mapping studies. Compared to the previous review published in [Paech, Delater, and Hesse 2014], six additional publications on documentation approaches were uncovered, and six additional related existing studies were found.

Analysis of Relevant Hits

All relevant hits were carefully examined in detail for their contribution regarding the given research questions. The resulting analysis regarding these research questions is presented in the following Section 4.3. However, if a publication was found silent with regard to the research questions, it was excluded from the aforementioned final set of relevant hits.

4.3 Results and Discussion

In the following sections, results for the research questions are described considering all relevant publications. Requirements for the documentation approach presented in this thesis were either newly derived from these results or, if appropriate, existing requirements were extended. It should be noted, that the approach “Decision, goals, and alternatives” (DGA) was described in four different publications [Davide Falessi, Cantone, and Becker 2006; Davide Falessi, Becker, and Cantone 2006; David Falessi, Cantone, and Kruchten 2008; Davide Falessi, Capilla, and Cantone 2008], but is referenced in the following sections by the first source only. Also, some approaches use the term *meta-model* to distinguish a generalized knowledge model from its instances used to structure and document specific decisions. In contrast, other approaches were found using the term *knowledge model* for an abstract model for documenting decision knowledge. Both terms will be used synonymously in the following sections.

4.3.1 Results for RQ1: Decision Knowledge Structures

All documentation approaches for decision knowledge revealed by this literature review propose and employ their own meta-models for structuring decision knowledge. Nevertheless, these approaches typically refer to basic representation concepts for decisions, such as QOC or DRL (cf. Section 2.4), and adopt previous documentation approaches. An important and fundamental meta-model for documenting decision knowledge is the documentation model by Tyree and Akerman. Like many other current documentation approaches, this model aims to capture and structure knowledge originating from design decisions [Tyree and Akerman 2005]. The knowledge elements of the model are depicted in Table 4.4.

Knowledge Element	**Considered Content**
Issue	Open questions addressed by the decision
Decision	The alternative finally chosen in the decision
Status	A state describing the current decision condition, like pending, approved, or rejected
Group	The category the decision belongs to
Assumptions	Assumptions concerning the context of the decision and their influences on the alternatives considered
Constraints	Limitations that result from the chosen alternative
Positions	Alternatives considered in the decision
Argument	Rationale supporting the selected position
Implications	The consequences which arise from the decision, like the need to adapt an artifact
Related decisions	Decisions related to the one described, e.g., due to influences or dependencies
Related requirements	Software requirements that set or influence the objectives for the described decision
Related artifacts	Other artifacts being concerned by the decision or concerning it
Related principles	Institutional principles that concerned or influenced the decision
Notes	Further notes related to the decision process

Table 4.4: Decision Knowledge Elements of the Model by Tyree and Akerman

The model of Tyree and Akerman is structured as a documentation template for decisions with predefined contents. For instance, the *Decision* itself is captured together with its underlying Issue as a description of the decision problem. Context knowledge is covered by *Assumptions*, *Constraints*, and *Implications*. Also different solution options may be stated by documenting different *Positions* with their related *Arguments*. The decisions' *Status* can be used to describe the actual condition of the decision, whereas the *Group* classifies the topic addressed by the decision. Relations may be specified between decisions, and to other related development knowledge, such as fundamental *Principles*, *Requirements* and development *Artifacts*.

Whereas the model of Tyree and Akerman already provides 13 different knowledge elements, other documentation approaches uncovered by the literature review add further knowledge elements or refine existing ones. A summary of the characteristics of these documentation approaches and their additions to the model of Tyree and Akerman is given in Table 4.5. Many approaches extend the documentation by refining specific knowledge elements. For instance, some approaches provide

Approach	Characteristics	Additions to Tyree and Akerman
Cooperative conceptual maintenance model (CM^2) [Canfora, Casazza, and De Lucia 2000]	Focus on decision rationale in the software maintenance process; comments are used to state rationales on source files	Refinement of related artifacts
RATSpeak [Burge and Brown 2004]	Extends DRL; emphasis on position and argument element	Background information like trade-offs, argument ontology
[Jansen and Bosch 2005]	Focus on architecture decisions	Refinement of implications through architectural modifications
[Smith, Bohner, and McCrickard 2005]	Decisions and related knowledge from development and project management	Claims, risks
DAMSAK [Ali Babar, Gorton, and Kitchenham 2006]	Focus on design decision and rationale	Scenario descriptions for decisions
Decision, goal, and alternatives (DGA) [Davide Falessi, Becker, and Cantone 2006]	Extends Tyree and Akerman	Project objectives
[Kruchten, Lago, and Vliet 2006]	Focus on design decisions; Many relationships between decisions such as “constrains”, “forbids”, “enables”	Enhanced state model, decision scope
[Capilla, Nava, and Duenas 2007; Capilla, Zimmermann, et al. 2011]	Distinguishes optional and mandatory attributes; Focus on relationships between decisions and other knowledge	Attributes for decision evolution, Relationships to quality attributes
[Zimmermann, Gschwind, et al. 2007]	Extends QOC	Involved persons such as decision identifier, responsible, taker
Architecture rationale and elements linkage (AREL) [Tang, Jin, and Han 2007]	Focus on design decisions and rationale	Motivational reasons for decisions
ADDRA [Jansen, Bosch, and Avgeriou 2008]	Focus on design decisions, identified by the delta between two architectural states	Problem causes
[Jansen, Avgeriou, and Ven 2009]	Focus on design decisions and knowledge domain modeling	Tailored knowledge meta-model
[Konemann 2009]	Focus on design decisions	UML design models
[Rockwell et al. 2009]	Focus on engineering design	Extended description of alternatives
[Buchgeher and Weinreich 2011]	Focus on tracing decisions from architecture to code	Code resulting from decisions
Toeska rationale extraction (TREx) [López et al. 2012]	Focus on realization of non-functional requirements (NFR)	Ontology for architecture and NFR; Decision goals
[Cleland-Huang et al. 2013]*	Focus on architectural goals and concerns of decisions	Quality goals
[Nowak and Pautasso 2013]*	Focus on position-based argumentation and situational awareness	Architecture elements
[Gaubatz, Lytra, and Zdun 2015]*	Focus on collaborative decision making	Formalized decision constraints
[Manteuffel, Tofan, Avgeriou, et al. 2016]*	Focus on decision viewpoints, like relationships, details, chronology	Decision history, involved stakeholders, UML design models

*Additional hit compared to the previous review published in [Paech, Delater, and Hesse 2014]

Table 4.5: Comparison of Revealed Decision Documentation Approaches

more sophisticated models for rationales, like RATSpeak with an argument ontology [Burge and Brown 2004] or AREL with motivations for taking a decision [Tang, Jin, and Han 2007]. Other approaches provide an extended problem description by adding problem causes [Jansen, Bosch, and Avgeriou 2008], decision goals [López et al. 2012], and quality goals [Cleland-Huang et al. 2013]. The solution description is enhanced by providing a more sophisticated knowledge structure for decision alternatives [Rockwell et al. 2009]. Furthermore, documentation approaches also propose a specialized documentation of decision knowledge related to particular development activities. For instance, Jansen and Bosch provide a refinement for implications by adding the resulting architectural modification [Jansen and Bosch 2005]. In contrast, architectural elements can be added directly to decision knowledge [Nowak and Pautasso 2013], or linked to UML diagrams [Konemann 2009; Manteuffel, Tofan, Avgeriou, et al. 2016]. Similarly, requirements are integrated with links to project objectives [Davide Falessi, Becker, and Cantone 2006] and non-functional requirements [López et al. 2012]. Moreover, the content and status model of documented decisions may be more formalized [Gaubatz, Lytra, and Zdun 2015; Kruchten, Lago, and Vliet 2006], as well as the evolution of decisions can be covered explicitly by attributes [Capilla, Nava, and Duenas 2007].

A detailed examination of the knowledge models suggested by the different documentation approaches is given in Table 4.6. Several characteristics of the uncovered documentation approaches are summarized in this table. First, the *model structure* is investigated, because this structure offers fundamental insights for answering the first research question of this review. Second, *iteration support* and *refinement support* within these model structures are highlighted, as both aspects need to be covered in order to support developers in documenting decisions collaboratively. Third, the *strategy support* of each approach is investigated. Thereby, it is determined whether both RDM and NDM strategies are supported.

In general, the model structure of documentation approaches may be *monolithic* like static text templates, a meta-model with *fixed* relations, or *flexible* with compositions and aggregations of knowledge elements. In detail, it was also investigated whether approaches support developers to extend their documented decisions in multiple iterations over time. For instance, this requires a versioning of knowledge elements or attributes to highlight evolved knowledge. Also, the approaches were checked for any kind of refinement structures for documented knowledge, such as fine-grained *refinement elements* or *refinement relations* within their meta-model. Finally, the approaches were assessed for the support of different decision making strategies according to their documentation support of decision solutions. The documentation could focus either on *choice* for RDM or *match* for NDM according to the strategy's mechanism for solving decision problems [Zannier, Chiasson, and Maurer 2007].

Interestingly, RATSpeak is the only approach providing a flexible structure for its knowledge model

Approach	Model Structure	Iteration Support	Refinement Support	Strategy Support
CM^2 [Canfora, Casazza, and De Lucia 2000]	Fixed	Yes	No refinement	Choice
RATSpeak [Burge and Brown 2004]	Flexible	Yes	Refinement elements and relations	Choice
[Jansen and Bosch 2005]	Monolithic	No	No refinement	Choice
[Smith, Bohner, and McCrickard 2005]	Fixed	Yes	No refinement	Match
DAMSAK [Ali Babar, Gorton, and Kitchenham 2006]	Monolithic	No	No refinement	Match
DGA [Davide Falessi, Becker, and Cantone 2006]				
[Kruchten, Lago, and Vliet 2006]	Fixed	Yes	Refinement relations	Choice
[Capilla, Nava, and Duenas 2007; Capilla, Zimmermann, et al. 2011]	Fixed	Yes	Refinement elements, sub-types	Choice
[Zimmermann, Gschwind, et al. 2007]	Fixed	Yes	Refinement relations	Choice
AREL [Tang, Jin, and Han 2007]	Monolithic	Yes	Refinement elements	Choice
ADDRA [Jansen, Bosch, and Avgeriou 2008]	Fixed	Yes	No refinement	Choice
[Jansen, Avgeriou, and Ven 2009]	Monolithic	No	Refinement elements	Choice
[Konemann 2009]	Monolithic	No	No refinement	Choice
[Rockwell et al. 2009]	Fixed	No	Refinement relations	Choice
[Buchgeher and Weinreich 2011]	Monolithic	Yes	Refinement relations	Choice
TREx [López et al. 2012]	Fixed	Yes	Refinement elements	Choice
[Cleland-Huang et al. 2013]*	Monolithic	No	Sub-types	Choice
[Nowak and Pautasso 2013]*	Fixed	Yes	No refinement	Choice
[Gaubatz, Lytra, and Zdun 2015]*	Fixed	Yes	No refinement	Choice
[Manteuffel, Tofan, Avgeriou, et al. 2016]*	Fixed	Yes	No refinement	Choice

*Additional hit compared to the previous review published in [Paech, Delater, and Hesse 2014]

Table 4.6: Comparison of Knowledge Structures within the Documentation Approaches

with both refinement elements and relations [Burge and Brown 2004]. All other approaches either rely on a monolithic or a fixed structure. The majority of approaches (13 out of 19) provides support for an iterative documentation of decision knowledge. Nevertheless, this does not imply that these approaches also propose structures for refining given knowledge. For instance, [Smith, Bohner, and McCrickard 2005] and ADDRA [Jansen, Bosch, and Avgeriou 2008] do not support refinement of documented decision knowledge. In contrast, [Rockwell et al. 2009] and [Jansen, Avgeriou, and Ven 2009] do offer fine-grained knowledge elements and relations for structuring decisions, but lack support for distinguishing different documentation versions. Only the two approaches described in [Smith, Bohner, and McCrickard 2005] and [Ali Babar, Gorton, and Kitchenham 2006] support an NDM documentation by matching solutions according to a described decision situation. All other approaches focus on RDM documentation by choice of a solution, e.g. by extending QOC (cf. [Zimmermann, Gschwind, et al. 2007; Gaubatz, Lytra, and Zdun 2015]) or DRL (cf. [Burge and Brown 2004]). In summary, no approach was found which offers a flexible structure of its knowledge model together with support for iterative documentation, refinement of given knowledge, and support for documenting both RDM and NDM.

4.3.2 Discussion of Additions to Requirements A and B

In the following paragraphs, additions to the requirements specified in Section 3.3 are discussed based on the results for decision knowledge structures in current documentation approaches.

Additions to Requirement A: Documentation of RDM and NDM Decisions

The results of the Firefox study presented in Section 3.3.3 already highlight the necessity to capture knowledge from both, RDM and NDM. In addition, the results of the literature review for RQ1 outline the missing support for a combined documentation of RDM and NDM in current approaches. First, the majority of approaches supports decision knowledge resulting from RDM according to their employed knowledge model, whereas only two approaches were found to support NDM documentation. This shows a discrepancy between the actual decision making behavior of developers and the support for structuring decision knowledge according to the applied decision making strategy during documentation. Regarding the results of the Firefox study, this might indicate that decision knowledge is currently lost because developers often use NDM, but are typically not supported in documenting this knowledge. Second, no approach was found which actually integrates the documentation of decision knowledge resulting from both RDM and NDM. However, the results of the Firefox study and other studies (cf. [Zannier, Chiasson, and Maurer 2007; Tang, Aleti, et al. 2010]) indicate that different developers tend to mix decision making strategies within one decision. Thus, knowledge models for documentation are required to provide structures supporting both RDM and NDM documentation.

Regarding the required knowledge structures for such an integrated documentation of RDM and NDM, the results for RQ1 indicate that particularly knowledge elements capturing decision solutions from RDM and NDM need to be integrated. For instance, it should be possible for developers to document NDM-related claims as well as RDM-related solution alternatives for the same decision. In addition, context knowledge, such as scenario descriptions originating from NDM and criteria originating from RDM, should be structured to facilitate documentation within the same decision.

Additions to Requirement B: Iterative Decision Documentation

The results of the Firefox study indicate that developers extend decision documentation iteratively over time, as new decision knowledge arises, which is subsequently added to the documentation (cf. Section 3.3.5). Nevertheless, the results of the literature review regarding RQ1 indicate that decision documentation in an iterative way with flexible knowledge structures is not supported sufficiently. Whereas many approaches enable developers to add further decision knowledge iteratively, only a few approaches also provide structures that can be used for knowledge refinement or even extensions of the knowledge model. Moreover, no approach was found that integrates support for iterative refinement with flexible structures and support for RDM and NDM within a single knowledge model. Also, current documentation approaches struggle to represent changes in the content and structure of decision knowledge, such as new or extended knowledge elements, and adaptions in the decision making behavior of developers.

In consequence, a flexible knowledge model with refinement support is an important addition to requirement B. In detail, the documentation approach presented in this thesis is required to provide defined abstract knowledge elements, which can serve as extension points for the given knowledge structure. These defined abstract knowledge elements are an important improvement to the monolithic and fixed structures employed by most current approaches. As a foundation for iterative refinement of documented decisions, a versioning of decision knowledge is required. Furthermore, refinement elements and relations should be provided that allow for extending and adapting given knowledge elements. On the one hand, this requires fine-grained knowledge elements to enrich already given general knowledge elements over time. On the other hand, specific relations between related decision knowledge elements enable developers to add further dependencies, which emerge during decision evolution.

4.3.3 Results for RQ2: Tools for Capturing and Linking Decision Knowledge

All approaches were examined for their capturing support for decision knowledge, and their abilities to link decision knowledge to other related knowledge. A summary of this investigation is described in Table 4.7. In general, three major mechanisms for capturing decisions knowledge were identified:

Approach	**Tool**	**Capturing**	**Linked Development Knowledge**
CM^2 [Canfora, Casazza, and De Lucia 2000]	COMANCHE	Manual	Decisions and code files
RATSpeak [Burge and Brown 2004]	SEURAT, SEURATArchitecture [Wang and Burge 2010]	Manual	Decisions and requirements, code files
Archium [Jansen and Bosch 2005]	Archium	Manual	Decisions and architectural knowledge
[Smith, Bohner, and McCrickard 2005]	LINK-UP	Manual	Decisions and design knowledge, project management knowledge
[Tyree and Akerman 2005]	–	Manual	Decisions and other related decisions, requirements and artifacts
DAMSAK [Ali Babar, Gorton, and Kitchenham 2006]	PAKME [Ali Babar and Gorton 2007]	Manual	Decisions and requirement
DGA [Davide Falessi, Becker, and Cantone 2006]	–	Manual	Decisions and goals, other related decisions, requirements, and artifacts
[Kruchten, Lago, and Vliet 2006]	–	Manual	Decisions and architectural knowledge
[Capilla, Nava, and Duenas 2007; Capilla, Zimmermann, et al. 2011]	ADDSS	Manual	Decisions and other related decisions and architectural knowledge
AREL [Tang, Jin, and Han 2007]	AREL	Manual	Decisions and architectural knowledge
[Zimmermann, Gschwind, et al. 2007]	ADkwik	Hybrid	Decisions and architectural knowledge
ADDRA [Jansen, Bosch, and Avgeriou 2008]	–	Automated	Decisions and architectural knowledge
[Jansen, Avgeriou, and Ven 2009]	Knowledge architect	Hybrid	Decisions and architectural knowledge
[Konemann 2009]	–	Manual	Decisions and UML models
[Rockwell et al. 2009]	–	Manual	Decisions and requirements
[Buchgeher and Weinreich 2011]	LISA	Automated	Decisions and architectural knowledge, code files
TREx [López et al. 2012]	Plugins/Rationale repository	Automated	Decisions and text documents
[Cleland-Huang et al. 2013]*	Archie	Hybrid	Decisions and requirements, architectural knowledge, code files
[Nowak and Pautasso 2013]*	Architecture Warehouse	Manual	Decisions and architectural knowledge
[Gaubatz, Lytra, and Zdun 2015]*	CoCoADvISE	Manual	Decisions and architectural knowledge
[Manteuffel, Tofan, Avgeriou, et al. 2016]*	Decision Architect [Manteuffel, Tofan, Koziolek, et al. 2014]*	Manual	Decisions and UML models

*Additional hit compared to the previous review published in [Paech, Delater, and Hesse 2014]

Table 4.7: Comparison of Capturing Mechanisms and Linked Development Knowledge

manual elicitation, automated extraction, or a hybrid approach. Manual elicitation supports developers in documenting decisions manually, whereas the automated extraction of decision knowledge derives knowledge from existing sources by knowledge discovery techniques. Hybrid approaches combine both manual elicitation and automated extraction. Applying these capturing mechanisms, the investigated approaches create different kinds of links to related knowledge.

All early approaches up to AREL [Tang, Jin, and Han 2007] as well as the approaches of [Konemann 2009], [Rockwell et al. 2009], [Nowak and Pautasso 2013], [Gaubatz, Lytra, and Zdun 2015], and [Manteuffel, Tofan, Avgeriou, et al. 2016] support the manual elicitation of decision knowledge according to the input of developers. ADDRA [Jansen, Bosch, and Avgeriou 2008], LISA [Buchgeher and Weinreich 2011], and TREx [López et al. 2012] use an automated tool-based extraction of decision knowledge from given artifacts. In detail, ADDRA recovers architectural models from given source code and other documents. Deviations in architecture are identified, which indicate possible decisions. LISA logs change events for architectural models and within code files. Based on these changes, decisions are linked with related architectural model elements and their implementation. In contrast, TREx derives decision knowledge by text mining in given text documents. Furthermore, the hybrid approaches ADkwik [Zimmermann, Gschwind, et al. 2007], knowledge architect [Jansen, Avgeriou, and Ven 2009], and Archie [Cleland-Huang et al. 2013] allow developers to manually enter decision descriptions according to their respective meta-models. In addition, ADkwik is capable of creating an initial description of design decisions based on textual requirements. Knowledge architect supports developers in semi-automatically extracting a project-specific decision knowledge model. Archie supports developers in identifying decisions and trace links to requirements and code by assigning an abstract link model to concrete project-specific goals, design elements, and code files.

Links are established to related knowledge in many areas. Beside links between related decisions and their knowledge elements, several approaches link decisions to architectural knowledge or requirements from which they originate or which they impact. For instance, TREx creates links between decision rationale and its source in textual documents. In addition, the approaches of Konemann and Decision Architect [Manteuffel, Tofan, Koziolek, et al. 2014] enable developers to manually relate decisions to their corresponding model elements in UML models. The approach of Tyree and Akerman and DGA [Davide Falessi, Becker, and Cantone 2006] allow for linking any related development artifact with a decision by textual reference. COMANCHE [Canfora, Casazza, and De Lucia 2000], SEURAT [Burge and Brown 2004], LISA [Buchgeher and Weinreich 2011], and Archie [Cleland-Huang et al. 2013] enable developers to link decisions with their realization in code. The approach of Smith, Bohner, and McCrickard provides annotations for linking claims with project-related knowledge, such as project deadlines and resource plans.

In summary, most approaches focus on linking decision knowledge with related knowledge originating from one specific development activity, such as design or requirements engineering. Only a few approaches span their links across different development activities, such as the approach of [Tyree and Akerman 2005], DGA [Davide Falessi, Becker, and Cantone 2006], and LISA [Buchgeher and Weinreich 2011]. Archie [Cleland-Huang et al. 2013] even relates decisions with requirements, architectural knowledge, and code files. However, Archie and LISA do not provide extraction support for decision knowledge from given artifacts, but focus on creating links based on manual decision descriptions. Furthermore, the approaches of [Tyree and Akerman 2005] and DGA [Davide Falessi, Becker, and Cantone 2006] only offer to create links between decisions and other artifacts manually without any tool support.

4.3.4 Discussion of Requirement C: Capturing Decision Knowledge during Development

Several approaches use artifacts from one development activity to extract or import knowledge from these artifacts for related decisions. Hereby, it is important to align knowledge capturing and extraction with the type of artifact that is used. Approaches focusing on design decisions typically use architectural components and UML models as knowledge source. They link decisions specifically with these sources, and decision contents refer to these concrete design artifacts. In contrast, textual descriptions of requirements are analyzed for decision knowledge using text mining and knowledge discovery techniques. To cover the realization of decisions, typically code files are linked with decision knowledge elements.

However, the results for RQ2 show that currently no approach provides capturing mechanisms to support knowledge extraction for decisions from given artifacts during multiple development activities for one integrated decision documentation. Thus, decision knowledge originating from one development activity remains isolated and is not enhanced by further decision knowledge resulting from other development activities. For instance, decisions made during requirements engineering are not reflected during design and implementation due to missing integration of the related decision documentation. Even current documentation approaches addressing more than one development activity can be improved, as they do not import decision knowledge originating from these activities, but only link existing decisions with the related artifacts.

Thus, it is an important requirement for the documentation approach developed in this thesis to capture decision knowledge during different development activities. As outlined in Section 2.2, the activities requirements engineering, design, and implementation are most suitable for an integrated decision documentation. This is also backed up by the findings for RQ2, as all of the investigated approaches provide links to knowledge originating from at least one of these activities. The documentation approach is required to support a hybrid extraction of decision knowledge for these activities, where it is technically suitable and provides useful contributions to decision contents.

4.3.5 Discussion of Requirement D: Decision Knowledge Links

Whereas many current approaches support developers in creating links between decisions and related artifacts in automated or hybrid way, the granularity of these links varies. Some approaches, like the model of Tyree and Akerman and DAMSAK [Ali Babar, Gorton, and Kitchenham 2006], link decisions and their knowledge elements to entire artifacts. In contrast, Archie [Cleland-Huang et al. 2013] and the approach of Manteuffel, Tofan, Avgeriou, et al. link knowledge elements of decisions to specific design elements, such as UML classes. Although some approaches link decisions to multiple development activities, none of these approaches provides fine-grained links to all kinds of related knowledge. Thus, links for one decision might be not comprehensive, as they may be detailed in relation to design, but coarse-grained for requirements.

In consequence, the last requirement for the documentation approach presented in this thesis is to provide links from decisions to development knowledge in a fine-grained way. On the one hand, this complements the fine-grained knowledge elements within the decision knowledge model of the approach. On the other hand, it assures that developers can always assign links between decisions and related knowledge at the appropriate level of granularity within the target artifacts. For instance, it may be important for developers to differentiate between links from decisions to entire use cases or single actor steps within these use cases. This is also in line with the findings of the Firefox study, where developers were found to reference entire code commits as well as single screenshots in their comments to decisions.

4.3.6 Results for RQ3: Usage of Decision Knowledge

All approaches were investigated for usage support of decision knowledge. The results are presented in Table 4.8. All approaches make decision making more transparent to developers through their documentation. In addition, they provide a direct navigation to particular knowledge elements within documented decisions. Three approaches support the enforcement of decisions made. The approach of Zimmermann, Koehler, et al. proposes to inject decision contents during model transformation of architectural models and code generation. Similarly, Konemann suggests to apply given decision knowledge when new design decisions are captured in relation to their corresponding UML models. The approach of Gaubatz, Lytra, and Zdun supports developers in specifying constraints for their architectural decisions, which are evaluated for new decisions. In contrast, several approaches provide an impact analysis for decisions and decision changes on related artifacts. RATSpeak [Burge and Brown 2004] visualizes the impact of decision changes regarding the coverage of requirements and potential changes within the code. Also Buchgeher and Weinreich track decision to architecture and code in order to visualize the realization of decisions made. By evaluating dependencies from trace links, ADDRA [Jansen, Bosch, and Avgeriou 2008] and the approach of Jansen, Avgeriou, and Ven address the impact of decision changes on the architecture. The approaches of Cleland-Huang et al. and Manteuffel, Tofan, Avgeriou, et al. extend this idea by integrating requirements and decision forces as drivers for potential decision changes. Similarly, the AREL approach [Tang, Jin, and Han 2007] employs causes to trace decisions to other knowledge, so that developers can explore requirements or constraint as potential origins for decision knowledge elements. Only the approach of Smith, Bohner, and McCrickard explicitly addresses risk management. In detail, potential risks, such as delays or deviations from planned tasks, are identified in relation to claims within decisions.

Usage	Approaches
Transparent decision making	All approaches
Direct navigation	All approaches
Decision enforcement	[Zimmermann, Gschwind, et al. 2007], [Konemann 2009], [Gaubatz, Lytra, and Zdun 2015]*
Impact analysis	RATSpeak [Burge and Brown 2004], AREL [Tang, Jin, and Han 2007], ADDRA [Jansen, Bosch, and Avgeriou 2008], [Jansen, Avgeriou, and Ven 2009], [Buchgeher and Weinreich 2011]*, [Cleland-Huang et al. 2013]*, [Manteuffel, Tofan, Avgeriou, et al. 2016]*
Risk management	[Smith, Bohner, and McCrickard 2005]

*Additional hit compared to the previous review published in [Paech, Delater, and Hesse 2014]

Table 4.8: Comparison of Usage Support for Decision Knowledge

In summary, many approaches provide an impact analysis from decisions to related artifacts, and some approaches are also concerned with preserving decisions within the system under development. However, no approach was found that provides particular support for analyzing the impact of changes within development artifacts on decision knowledge. Also, quality management or communication of decision knowledge typically remain implicit, as they may be supported by the results from tracing decisions to quality requirements or from impact analysis.

4.3.7 Discussion of Additions to Requirements C and D

The results for RQ3 indicate the usage of decision knowledge is currently focused on how decisions influence related development activities. However, it could be beneficial for developers to use such relations in a bidirectional way. For instance, this would make it easier for developers to recognize that unclear or vague requirements lead to assumptions within their corresponding decisions [Hesse and Paech 2016]. Regarding requirement C, this finding raises the need for bidirectional links, which may originate from a development artifact to a decision and vice versa. This means that developers are enabled to create links during decision documentation and when working on other development artifacts, such as requirement descriptions, design models, and code. Regarding requirement D, links between decisions and other development knowledge should focus on artifacts that can actually impact decisions, and are not only affected by decisions themselves.

4.4 Threats to Validity

Four different categories of threats to validity have to be considered for this study according to Runeson et al. [Runeson et al. 2012]. These categories are described and discussed in the following paragraphs.

Internal Validity
Threats to internal validity concern the correlation between the investigated factors and other factors [Runeson et al. 2012]. There is the risk that documentation approaches were designed to serve other purposes than discovered within the review, for instance regarding the capturing and usage of decision knowledge. This threat was mitigated by searching not only for the decision documentation approach itself, but also for tools supporting the respective approach. Thereby, the intended design and actual realization of the approach could be cross-checked within the tool and its related publications.

Construct Validity
Threats to construct validity concern any gaps between intended and actual observations of the researchers [Runeson et al. 2012]. Whereas as many sources as feasible were included, still even more references from further conferences, journals, and publishers could have been searched. In particular, the manual search might be incomplete due to low coverage of relevant publications. Nevertheless, the automated search engines of IEEE, ACM, Elsevier, and Springer already include many important venues. Thus, most of the relevant literature should have been covered. As the manual search was based on the references of hits from automatic search, it is less likely that important publications were not found. Another threat could be that algorithms and data sources of automated search engines evolve over time, so that quantity and quality of search results might vary. This risk was mitigated by increasing the quality of the automated searches with exploratory search, and cross-checks for references within the manual search.

Reliability Validity

Threats to reliability validity concern the degree to which data and analyses of a study are dependent on specific researchers [Runeson et al. 2012]. As described in Section 4.2, no strict literature review according to [Kitchenham and Charters 2007] was conducted. In consequence, the assessment of literature might have been biased or its classification even might have been incorrectly due to a missing second reviewer. This threat was addressed by checking all classification results with the existing studies and examining publications twice, if deviations or different interpretations occurred.

External Validity

Threats to external validity concern the degree to which the results of our study can be generalized [Runeson et al. 2012]. Many investigated approaches address design decisions, whereas only a few approaches focus on decisions related to requirements or code. In consequence, the findings for the research questions might be specific to design decisions, so that the significance of these findings is limited for other kinds of decisions. However, this threat is mitigated by the findings for RQ1 and RQ2. These findings show that approaches addressing different kinds of decisions share similar knowledge structures and capturing mechanisms for decision knowledge. Next, approaches not discovered during the searches might influence the findings of the review. In this case, the results could not be generalized because important approaches were missed. This threat was addressed by employing the manual search to complement potential gaps of the automated search engines. Furthermore, related studies were used to check that all important approaches were covered by the search.

Part III

Solution Approach

5 An Incremental and Strategy-Independent Approach for Documenting Decisions

In this chapter, a documentation approach for decision knowledge is developed. This approach incremental documentation of decisions, and can be used with NDM as well as RDM. First, a *running example* is specified to demonstrate the usage of the different documentation elements provided by the approach. Next, the developed *Decision Documentation Model* is presented with its elements, their attributes, and relationships between the elements. Subsequently, it is highlighted how the model supports decision documentation during requirements engineering, design, and implementation by providing integrated decision knowledge representations and tailored knowledge capturing mechanisms for these development activities. Within the presentation of the model, it is explained how the model realizes the requirements identified in Chapters 3 and 4.

5.1 Running Example

For the introduction of the documentation model, a running example is used to illustrate the structure and usage of the model, its elements, and relationships. This example is derived from decision situations within the CoCoME project, which models a sales and storage management system for supermarket enterprises. In particular, the system modeled in the CoCoME project supports sales processes in single stores as well as an inventory management within the entire enterprise to optimize product orders from suppliers [Hesse, Kuehlwein, and Roehm 2016]. A detailed description of the modeled system and the CoCoME project as case study for different architecture and performance benchmark approaches is given in [Herold et al. 2008]. In this thesis, the modeled system and the CoCoME project are used synonymously and referred to as *CoCoME*.

For the running example, knowledge on a design decision concerned with migrating major parts of CoCoME to the cloud is used (abbreviated as *migration*). Whereas the migration impacts security requirements and the actual implementation of the system, it is beneficial for its cost-effectiveness and scalability [Hesse, Kuehlwein, and Roehm 2016]. In practice, decision knowledge typically arises over time in an unstructured and loosely related way, as indicated by the results of the Firefox study (cf. Section 3.3). This is reflected by the detailed presentation of the decision knowledge for the running example, as depicted in Figure 5.1. Within the example, it is assumed that different developers contribute decision knowledge to the migration. First, the architect *Alice* brings up the decision, because she is concerned about the cost-effectiveness of the current system. From her point of view, migrating sales and order services to the cloud would help to decrease costs significantly. She also states that this cannot be achieved by only optimizing the current architecture. However, requirements engineer *Bob* is worried that the cloud migration of services may impact system security. In particular, he wants to preserve secure interactions with customers, but assumes that personal

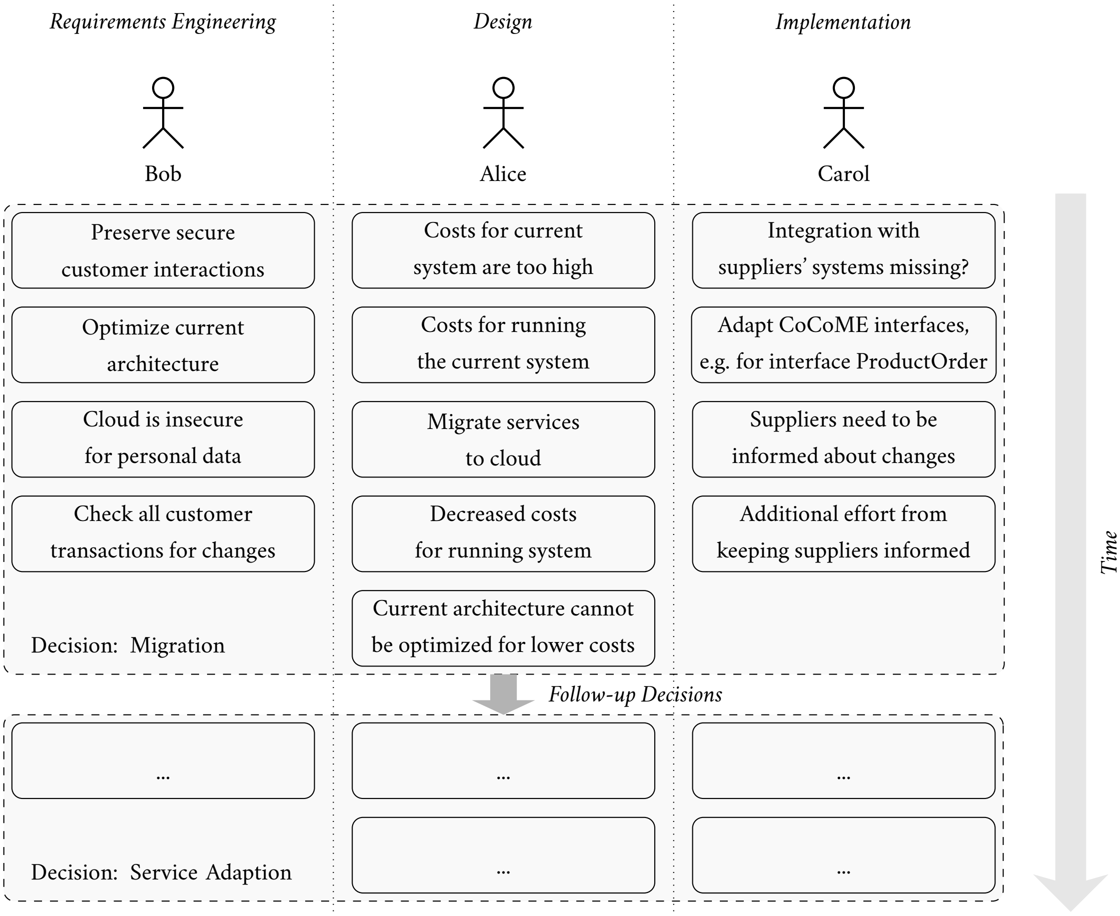

Requirements Engineering
Design
Implementation
Bob
Alice
Carol
Preserve secure customer interactions
Costs for current system are too high
Integration with suppliers' systems missing?
Optimize current architecture
Costs for running the current system
Adapt CoCoME interfaces, e.g. for interface ProductOrder
Cloud is insecure for personal data
Migrate services to cloud
Suppliers need to be informed about changes
Check all customer transactions for changes
Decreased costs for running system
Additional effort from keeping suppliers informed
Current architecture cannot be optimized for lower costs
Decision: Migration
Follow-up Decisions
...
...
...
...
...
Decision: Service Adaption
Time

Figure 5.1: Overview of Decision Knowledge for the Running Example

customer data may be exposed within the cloud. Third, developer *Carol* looks into the details of implementation. She recognizes a missing integration of external systems, e.g. from suppliers, with the changed services of CoCoME. This results in additional effort for communication and creates new dependencies between the systems. Thus, a follow-up decision is made to change the actual implementation of CoCoME services to address the identified security concerns and third-party dependencies.

5.2 Requirements Overview

Based on the study results presented in Sections 3.3 and 4.3, several requirements were derived for the documentation approach and its tool support. These requirements are summarized with regard to the documentation model in the following paragraphs:

- **Requirement A: Documentation of RDM and NDM Decisions**
  The documentation model shall support the documentation of decision knowledge resulting from both RDM and NDM. This includes:

  A.1 Entities to capture decision knowledge resulting from RDM and/or NDM (cf. Sections 3.3.4, and 4.3.2)

  A.2 Integration of RDM and NDM documentation within the same decision (cf. Section 4.3.2)

  A.3 Classification of different decision types (cf. Section 3.3.8)

- **Requirement B: Iterative Decision Documentation**
  The documentation model shall support an iterative documentation of decision knowledge for one decision over time. This includes:

  B.1 Consistent iterative documentation structures for each kind of DM strategy (cf. Section 3.3.6)

  B.2 Abstract knowledge elements as extension points (cf. Section 4.3.2)

  B.3 Fine-grained knowledge elements for capturing decision problem and context knowledge (cf. Section 3.3.6)

- **Requirement C: Capturing Decision Knowledge during Development**
  The documentation model shall provide capturing mechanisms for decision knowledge during different development activities. This includes:

C.1 Capturing of decision knowledge during requirements engineering (cf. Section 4.3.4)

C.2 Capturing of decision knowledge during design (cf. Section 4.3.4)

C.3 Capturing of decision knowledge during implementation (cf. Section 4.3.4)

- **Requirement D: Decision Knowledge Links**
  The documentation model shall offer different relations to link knowledge elements with each other, as well as with further development artifacts. This includes:

  D.1 Relations to refine knowledge elements (cf. Sections 4.3.2, and 4.3.5)

  D.2 Relations between knowledge elements and development artifacts (cf. Section 4.3.5)

5.3 Decision Documentation Model

In the following subsections, the documentation model elements with their attributes and relations are introduced. Therefore, all elements of the model are briefly introduced to provide an overview of the entire model. Then, every element is described in detail using the running example. To distinguish the different entities, *knowledge elements* are written in italics, whereas `attributes` and `relations` are written in Courier font. It should be noted that in the following sections *decision knowledge element* is used as a term to reference the general type of decision knowledge, whereas concrete decision knowledge documented for a given example decision is referred to as *instance*. However, both terms might be used synonymously if the context clarifies whether general or concrete decision knowledge is addressed. The graphical notation of the figures presented in the following sections is based on UML. In detail, knowledge elements are depicted as UML classes, whereas instances of the elements are shown as UML objects, having an underlined name and a knowledge element as type classifier. Also, the UML relationship "generalization" is used to visualize sub-types of knowledge elements by a white triangle arrow tip. Furthermore, the UML relationship "composition" shows that one knowledge element is contained within another one. This is depicted by a filled rhombus shape as arrow tip.

5.3.1 Overview

All model elements are depicted in Figure 5.2. The root element is *Decision*, which constitutes a container element for decision knowledge regarding one decision. This decision knowledge can be documented by adding multiple *DecisionComponents*. Thereby, a *Decision* may be a composition of different *DecisionComponents*, so that these components are contained within the *Decision*. This is depicted using the `Contained in`-relation. It should be noted that one *DecisionComponent* may also contain further components. Furthermore, both *Decision* and *DecisionComponent* are sub-types of the abstract *KnowledgeElement*, which represents a general artifact of software development, such as a requirement. This is visualized using the `Sub-type of`-relation.

Four major types of *DecisionComponents* are distinguished with different sub-types. First, a *Question* and its sub-type elements *Issue* and *Goal* may be used to capture knowledge on the decision problem. Second, *Solutions*, such as *Alternatives* and *Claims*, may be used to document different options to solve the decision problem. Third, *Context* elements capture knowledge on the decision context, like *Assumptions* about the decision situation, *Constraints* for the decision, or *Implications* resulting from choosing a particular solution. Finally, *Arguments* and their sub-type *Assessment* cover knowledge on rationales supporting or attacking other knowledge elements within the decision.

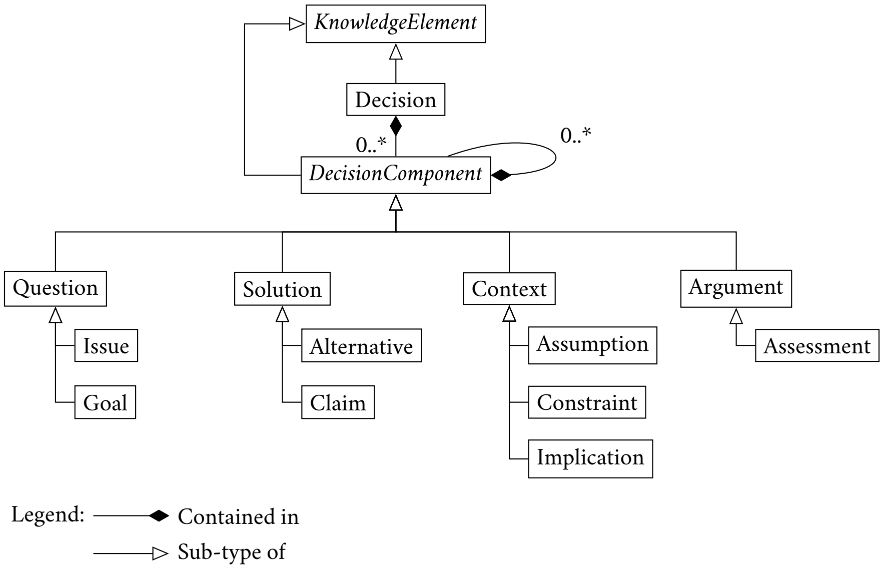


Figure 5.2: Overview of the Decision Documentation Model

The realization of the requirements for the documentation model is briefly introduced using the model overview. First, the documentation model supports knowledge documentation resulting from both RDM and NDM decisions (A.1), as knowledge elements for both kinds of DM strategies are given. For instance, *Claims* from NDM decisions as well as *Alternatives* from RDM decisions may be used to describe a solution option. Next, the model contains abstract knowledge elements as extension points for more detailed elements (B.2). These high-level elements are *Question*, *Solution*, *Context*, and *Argument*. They can be instantiated with concrete knowledge themselves or may be inherited by more specific elements. As an example, consider the refinement of *Questions* into *Issues* and *Goals*, as well as *Context* knowledge into *Assumption*, *Constraint*, and *Implication*. Thereby, fine-grained knowledge elements are provided (B.3). Finally, a *Decision* is composed of various *DecisionComponents*, which themselves may contain further *DecisionComponents*. Using this composition mechanism, the documentation for one decision can be extended over time in an iterative way (B.1). As all other decision knowledge elements are sub-types of *DecisionComponent*, the extension allows for using both RDM and NDM elements in a consistent way (A.2). More details on the realization of these requirements are given in the following subsections.

5.3.2 Decision and DecisionComponent

The core of the documentation model is formed by the elements *Decision* and *DecisionComponent*. These elements are depicted with their full specification in Figure 5.3. It should be noted that

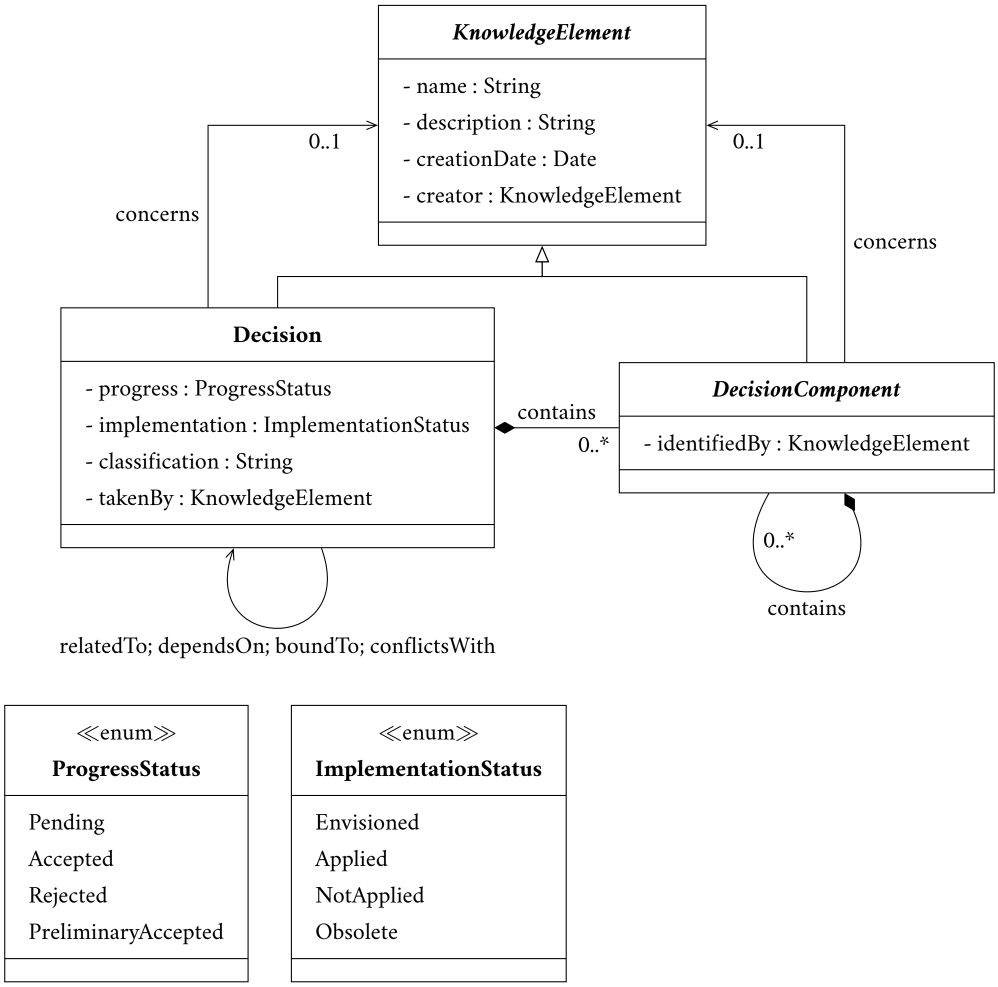


Figure 5.3: Details of *Decision* and *DecisionComponent*

KnowledgeElement is used as an abstraction for a basic knowledge element within the documentation model, which is not necessarily concerned with decision knowledge. It provides basic attributes for all other knowledge elements within the model, such as a `description` representing the content of the element, and a `name` to summarize its payload. In addition, the `creator` and the `creationDate` provide administrative information about the author and time when the knowledge element is created. Thereby, *KnowledgeElement* becomes the common foundation for all other documentation model elements.

Decision

This is a container element to store all knowledge elements a documented decision consists of. Therefore, it can hold any set of *DecisionComponents* using the composition `contains`. This implies that the existence of the contained *DecisionComponent* instances depends on the existence of the containing *Decision* instance. This dependency was implemented to ensure that *DecisionComponent* instances are documented individually for each decision with specific textual contents and links, so that a particular decision becomes more comprehensive through the documentation. At the same time, the existential dependency between *Decisions* and *DecisionComponents* supports the maintenance of documented decision knowledge, because links to *DecisionComponents* are removed when the containing *Decision* is no longer present. As an extension to the basic attributes of *KnowledgeElement*, several administrative attributes are provided. The `progress` represents the status of decision making for the decision. For instance, a decision may be `Accepted`, because the developers agreed upon the solution, or `Pending`, when the decision making process is not yet finished. Also, the decision may be `Rejected`, if no viable solutions were identified. In addition, is is possible to mark decisions as `PreliminaryAccepted`, which is particularly useful for NDM decisions. Then, a solution is accepted until a better one is discovered or the current solution is no longer applicable. Furthermore, the *Decision* provides the `ImplementationStatus`, which describes the decision outcome upon realization of the documented solution. This realization is `Envisioned` as long as the implementation of the solution is not yet finished. When a solution is fully implemented and working, developers can mark the decision as `Applied`. In contrast, `NotApplied` indicates that a taken decision is not realized. Moreover, a valid decision may become `Obsolete` over time, because it is overruled by follow-up decisions. To enable the classification of different decision types (requirement A.3), the attribute `classification` allows for specifying different keywords for one decision. Also, the developer actually taking the decision is stored within the attribute `takenBy`, as the decision might have been documented by a different developer.

Decisions can be linked to each other using different relations. With `relatedTo`, it is expressed that some kind of relation exists between two decisions, but the relation cannot be further specified. In contrast, `dependsOn` between two decisions X and Y expresses that Y depends on a successful decision making process for X. When a decision is made, it can be documented by setting `Accepted` for the respective decision (cf. ProgressStatus). Even more strict is the relation boundTo, which indicates that Y depends on the application of X. If this relation is used, Applied should be documented for X (cf. ImplementationStatus). Furthermore, an exclusion relation between two decisions X and Y can be expressed using conflictsWith. In this case, X and Y cannot be applied together. These relations help developers to refine their documented decisions over time, and thereby address requirement D.1. In Figure 5.4, two decisions from the running example are documented as *Decision* knowledge elements to illustrate the presented attributes and relations. In the example, the decision to adapt the service layer of CoCoME is bound to the decision to migrate the system into the cloud.

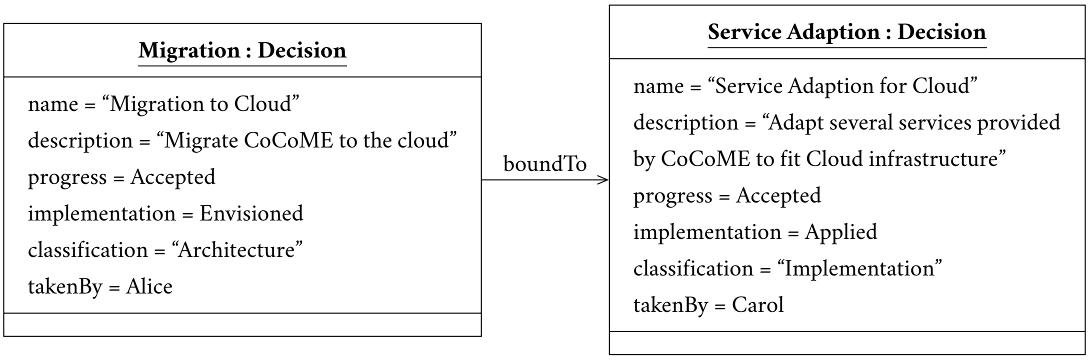

Migration : Decision
name = "Migration to Cloud"
description = "Migrate CoCoME to the cloud"
progress = Accepted
implementation = Envisioned
classification = "Architecture"
takenBy = Alice
boundTo
Service Adaption : Decision
name = "Service Adaption for Cloud"
description = "Adapt several services provided by CoCoME to fit Cloud infrastructure"
progress = Accepted
implementation = Applied
classification = "Implementation"
takenBy = Carol

Figure 5.4: Examples for *Decision*

DecisionComponent

The *DecisionComponent* represents the core knowledge element within the documentation model, as it is the parent element for all other decision knowledge elements. Like *Decision*, it also inherits the basic attributes of *KnowledgeElement*. In addition, it provides a link to the developer identifying the element using identifiedBy. One *DecisionComponent* may contain multiple further *DecisionComponents* via the contains-relation. Thereby, trees of *DecisionComponents* can be created within one *Decision* element. In this way, an iterative documentation structure is given (requirement B.1), which supports developers in adding more *DecisionComponents* either to a *Decision* or to existing *DecisionComponents* over time. Both *Decision* and *DecisionComponent* may be linked to any other knowledge element using the relation concerns. In particular, this enables developers to link development artifacts as a special kind of *KnowledgeElement* with decisions and their components (requirement D.2, cf. Section 5.4).

5.3.3 Question, Issue and Goal

An overview of the *Question* element and its sub-elements is given in Figure 5.5. Whereas *Question* serves as the general knowledge element to describe decision problems, *Issue* and *Goal* are more fine-grained and specific. In detail, *Issue* provides the attribute errorDetails to capture erroneous

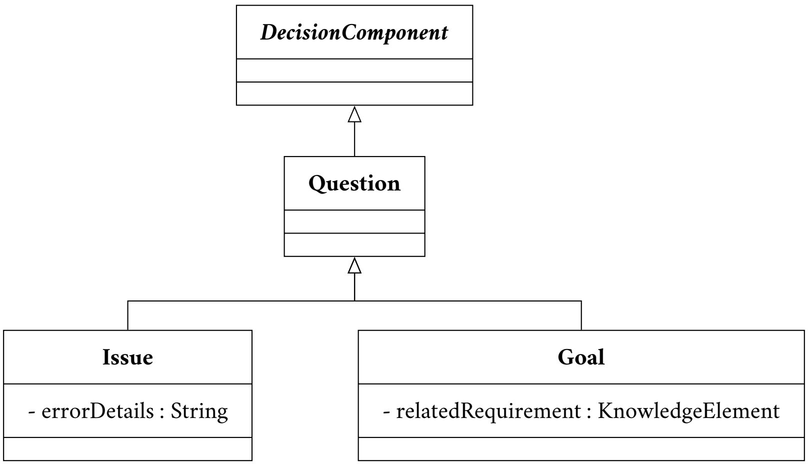


Figure 5.5: Details of *Question*, *Issue*, and *Goal*

situations as decision problems. For instance, this would be the case if a bug report is the starting point of a decision. In contrast, *Goal* links a `relatedRequirement` with a textual goal description. Thereby, a goal can be used to document a feature request, which is directed to extensions and adaptions of given functionality. In summary, these elements enable the fine-grained capturing of decision problems (requirement B.3). In addition, they cover decision problems regarding both RDM and NDM. However, a *Question* can be instantiated directly if none of the two sub-types fits a given problem description. This might be the case, when developers start documenting before the kind of a decision problem is fully determined. Furthermore, a *Question* is appropriate for documenting a rational problem description (cf. questions in the QOC approach in Section 2.4), whereas *Issues* and *Goals* may be used to capture NDM problems. It should be noted that *Question* can be related with *Solution* directly, which is described in Subsection 5.3.7.

Example instances of these three knowledge elements are depicted in Figure 5.6. The figure shows that the missing integration with the suppliers' systems from the running example was documented as *Question*, which was brought up by Carol. In addition, Alice raised the concern of increased costs, because the current ability to scale up with an increasing number of shops is limited. The performance of the current system appears to be insufficient, as costs increase dramatically for a relatively small number of shops. Thus, it was documented as *Issue* with specific details on the error. Furthermore, Bob strives for the *Goal* of secure interactions, so that the related security requirement is fulfilled. This issue type was chosen, because a single requirement needed to be addressed with a solution, although a potential erroneous behavior of the system was only anticipated, but not yet observed. In the example, no direct links exist between the aforementioned instances, so that no relations are depicted in Figure 5.6. However, these instances contain further instances of *DecisionComponents* regarding the entire decision documentation for the running example, which will be summarized in

Section 5.3.8.

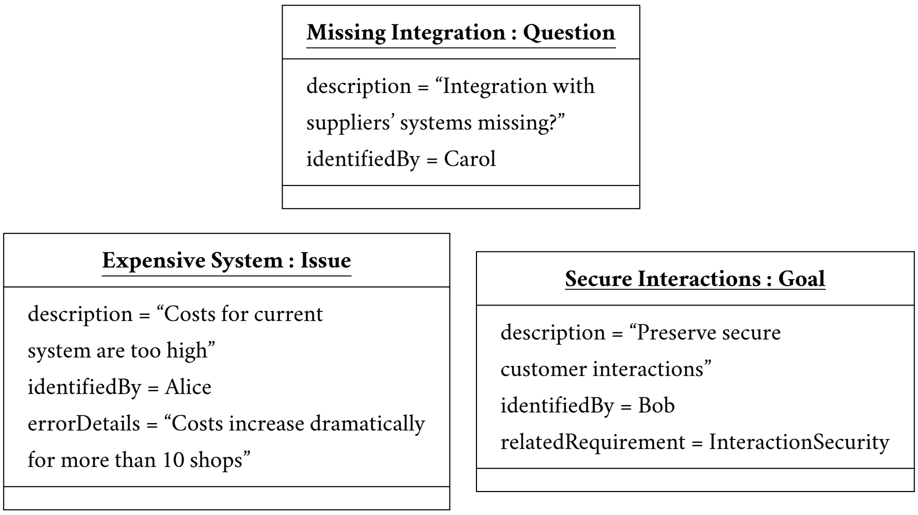

Missing Integration : Question
description = "Integration with suppliers' systems missing?"
identifiedBy = Carol
Expensive System : Issue
description = "Costs for current system are too high"
identifiedBy = Alice
errorDetails = "Costs increase dramatically for more than 10 shops"
Secure Interactions : Goal
description = "Preserve secure customer interactions"
identifiedBy = Bob
relatedRequirement = InteractionSecurity

Figure 5.6: Examples for *Question, Issue,* and *Goal*

5.3.4 Solution, Alternative and Claim

The knowledge element *Solution* as well as its child elements are depicted in Figure 5.7. As for *Question* and its children, also *Solution* is the general documentation element to cover all kinds of solutions within decisions. It is extended by the RDM-oriented *Alternative* (cf. options in the QOC approach in Section 2.4) and by the NDM-based *Claim*. For a *Claim*, the `type` can be specified, because these solution descriptions strongly depend on the scenario they are derived from. The content of a *Claim*

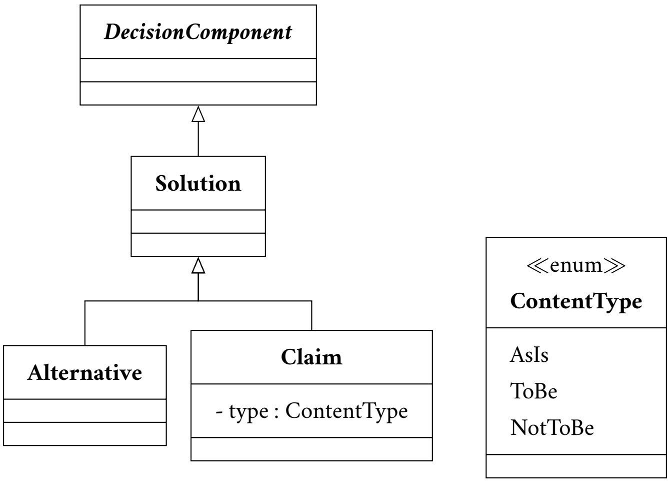


Figure 5.7: Details of *Solution, Alternative,* and *Claim*

may describe an intended and positive change by marking it with `ToBe`. Also, *Claims* may be used to simply state an existing solution by using `AsIs`, or indicate a negative outcome through `NotToBe`. Again, it should be noted that *Question* can be related with *Solution* directly, which is described in Subsection 5.3.7.

An example of different instances of *Solution* elements is depicted in Figure 5.8. Here, the more general idea of Alice to move the CoCoME services to the cloud is represented as *Solution*. Bobs proposal to optimize the current architecture is another option to solve the CoCoME cost and performance problems, so it is represented as *Alternative*. In contrast, Carols intended change of the CoCoME interface was modeled as a *Claim*, which is an intended improvement within the scenario of a cloud-based CoCoME.

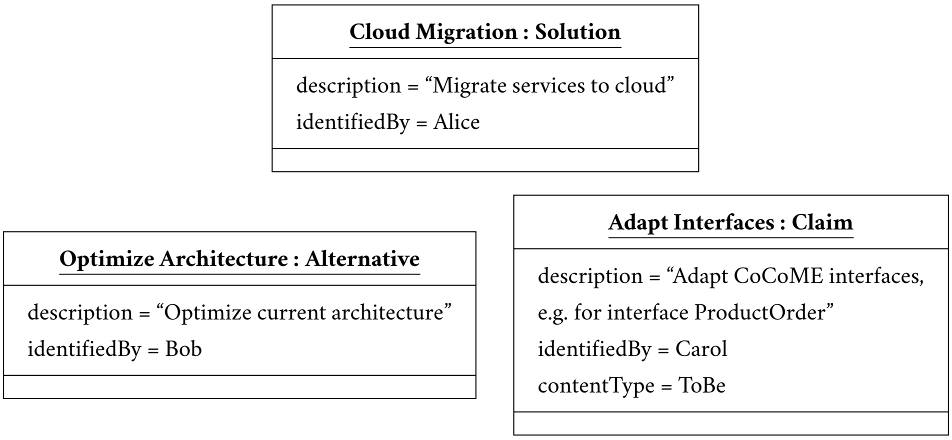

Cloud Migration : Solution
description = "Migrate services to cloud"
identifiedBy = Alice
Optimize Architecture : Alternative
description = "Optimize current architecture"
identifiedBy = Bob
Adapt Interfaces : Claim
description = "Adapt CoCoME interfaces,
e.g. for interface ProductOrder"
identifiedBy = Carol
contentType = ToBe

Figure 5.8: Examples for *Solution*, *Alternative*, and *Claim*

5.3.5 Context, Assumption, Constraint and Implication

An overview of all knowledge elements for documenting the decision context is given in Figure 5.9. With a variety of attributes, the *Context* element supports various types of information regarding the environment and situation surrounding a decision. First, the *Context* element may be used to cover knowledge regarding solutions. For this purpose, `isCriterion` provides a flag to indicate that context knowledge is useful for evaluating a given solution (cf. criterion in the QOC approach in Section 2.4). This can be refined by indicating the type of `measure` for the criterion. For instance, the evaluation might be performed regarding `Quantitative` or `Qualitative` aspects. Also, the `type` of context can be specified to mark the description as a current or future situation. The support for specifying a `scenario` is important to capture NDM decisions, because this represents a typical foundation for *Claims* (cf. scenario-based claim analysis in Section 2.4). Furthermore, context

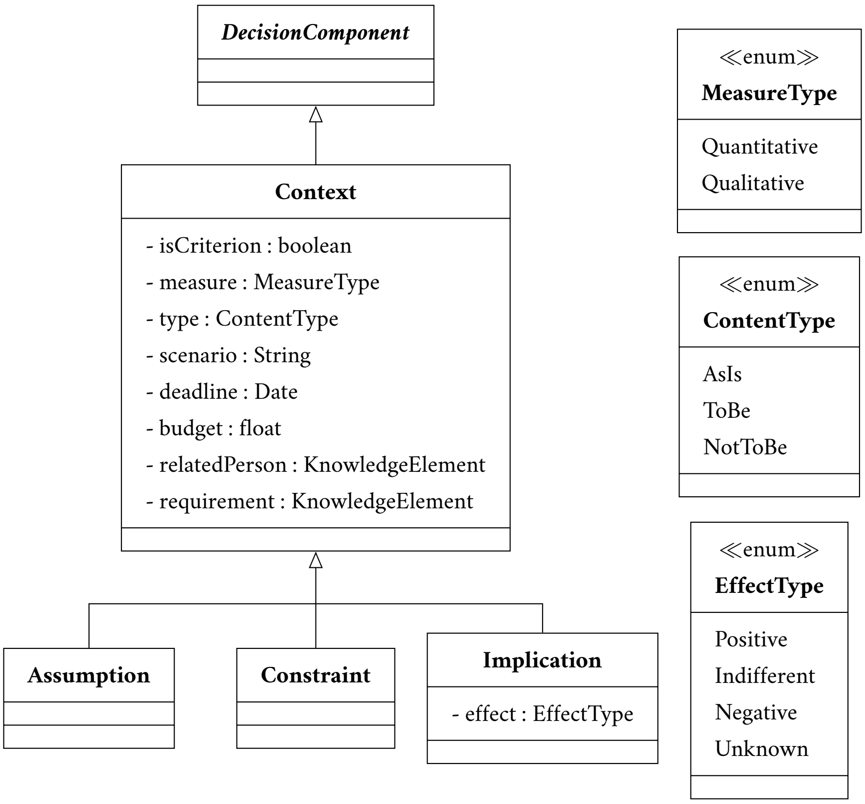


Figure 5.9: Details of *Context*, *Assumption*, *Constraint*, and *Implication*

knowledge may refer to timelines, which can be specified using `deadline`. Similarly, a `budget` can be captured. Finally, several references are provided. It is possible to document a `relatedPerson` as a potential source of implicit knowledge for the documented context. Also, a `requirement` addressed by this element can be specified, as requirements may be critical sources of *Context* knowledge. For instance, a security requirement may constrain the solution space of a decision problem. Therefore, it should be linked to the respective *Context* instance explicitly. It should be noted that relations between *Context* and other *DecisionComponents* are described in Section 5.3.7.

Several sub-types of *Context* exist. First, *Assumption* allows for capturing uncertain information related to a decision, such as suggestions and assumptions of developers. Second, restrictions and limitations regarding a decision may be documented using *Constraint* element. Third, consequences of making a decision, e.g. by choosing a particular alternative, can be expressed through the *Implication* element. Here, the `effect` can be specified as a qualitative assessment: consequences may be seen as `Positive`, `Indifferent`, or `Negative`. However, the actual effect of a consequence can also be `unknown` when the element is documented.

An example for the described knowledge elements is given in Figure 5.10. Alice has documented the costs for running CoCoME as an instance of*Context*. She marked it as criterion, and also specified the costs as quantitative measure with a deadline for evaluation. In contrast, Bob contributed an instance of *Assumption* for documenting his fear that a cloud solution would be insecure for personal data. In addition, he has specified potential transaction changes for the customers as an *Implication*. It is still unknown whether these changes will impact the transactions in a positive or negative way, so that he flags the `effect` as `Unknown`. Furthermore, Carol identified the need to inform the CoCoME suppliers as a *Constraint*.

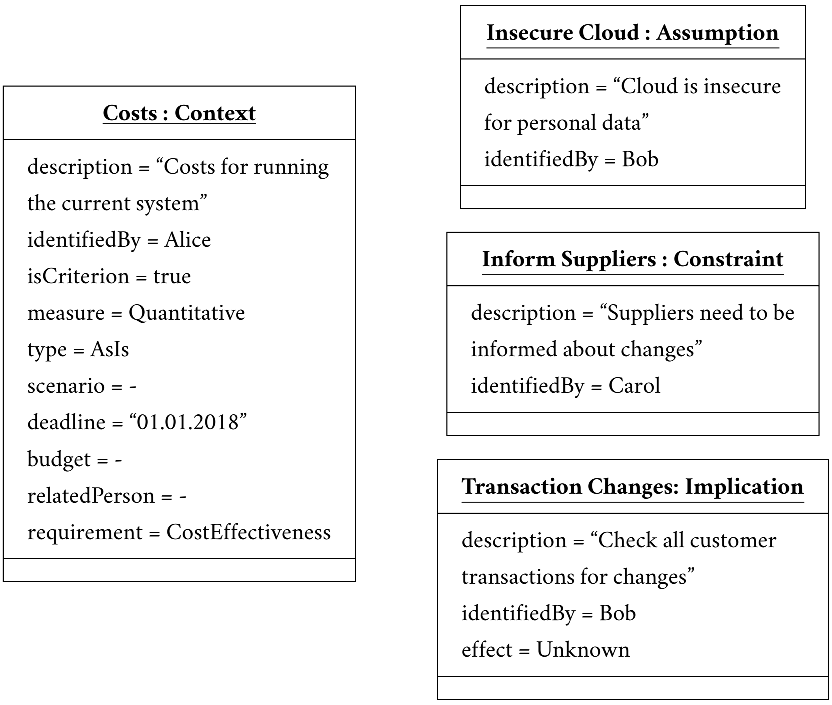

Insecure Cloud : Assumption
description = "Cloud is insecure for personal data"
identifiedBy = Bob
Costs : Context
description = "Costs for running the current system"
identifiedBy = Alice
isCriterion = true
measure = Quantitative
type = AsIs
scenario = -
deadline = "01.01.2018"
budget = -
relatedPerson = -
requirement = CostEffectiveness
Inform Suppliers : Constraint
description = "Suppliers need to be informed about changes"
identifiedBy = Carol
Transaction Changes: Implication
description = "Check all customer transactions for changes"
identifiedBy = Bob
effect = Unknown

Figure 5.10: Examples for *Context*, *Assumption*, *Constraint*, and *Implication*

Overall, the *Context* element and its sub-types are crucial for enriching documented decisions with specific knowledge. The intention behind the *Context* elements is to provide documentation structures with many attributes to enrich other *DecisionComponents* with contextual knowledge. Thereby, the documentation of knowledge on the decision problem and its solution can be separated from describing the context. First, this allows for adding context knowledge over time, as it is uncovered or emerges within the decision situation. Second, where attributes are given for *Context*, duplication of these attributes is not necessary within *Question* and *Solution* elements. Thus, it becomes easier for developers to maintain the content of these attributes. For instance, the *Claim* of Carol to adapt the product order interfaces in the CoCoME running example contains a *Constraint*, that suppliers need to be informed about the change. Thereby, the *Context* element and its sub-types address the requirement for fine-grained problem and context knowledge elements (B.3).

5.3.6 Argument and Assessment

In Figure 5.11, an overview of the elements for the documentation of rationale knowledge is given. As every rationale for a decision is either an argument backing up the solution or attacking the alternatives, *Argument* is the general knowledge element to cover rationales. A sub-type of *Argument* is *Assessment*, which can be used to evaluate solutions according to criteria. Then, it is possible to indicate whether the evaluation result is a certain `quantity` according to the meaning of the criterion or if a certain `quality` was found. Hereby, the quality types range from `VeryGood` to `VeryPoor` and `Inapplicable` in case that no quality could be determined.

Arguments can be linked to any other *DecisionComponent*. In detail, the `supports`-relation is used to express that an argument strengthens another *DecisionComponent*. In contrast, the `attacks`-relation indicates that an argument is opposed to a *DecisionComponent*. In case an argument is a neutral statement, the `comments`-relation can be applied. As *Argument* is a child of *DecisionComponent*, arguments can also be related to each other. This may lead to complex argumentation graphs with sophisticated syntax and semantics.

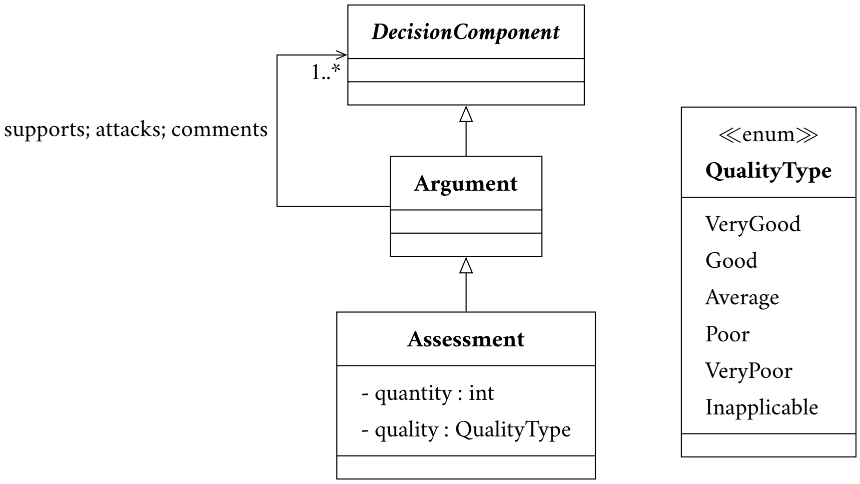


Figure 5.11: Details of *Argument* and *Assessment*

Argumentation Graphs

A prominent example for such argumentation graphs are *abstract bipolar argumentation systems*, which are described in detail in [Cayrol and Lagasquie-Schiex 2009]. The foundation for such systems are arguments that can be related with each other through a positive relation (here: the `supports`-relation) and a negative relation (here: the `attacks`-relation). It is important that an argument cannot support and attack another argument at the same time, but loops created with these relations are allowed. For instance, an argument could be self-attacking. Then, complex paths of arguments connected by the `attacks`- or `supports`-relation may arise. In detail, a path of `supports`-relations can end with an attack on a further argument. Then, the first argument of this path attacks the last one *indirectly* (referred to as *supported attack*). Also, sets of arguments can attack or support another argument, if a direct or supported path of attack or support exists from an argument within the set to the other argument. Based on this syntax, semantics can be defined to evaluate given argument graphs for coalitions, and victory regarding the documented relations.

Arguments from the running example are depicted in Figure 5.12. Here, Carol states that it is additional effort to keep the suppliers informed about interface changes. This *Argument* is opposed to the *Claim* for interface adaptions. In addition, Alice has uncovered that the current architecture of CoCoME cannot be optimized for lower costs. This *Assessment* of the criterion of costs has multiple relations to other knowledge elements. On the one hand, it `supports` the *Solution* of migrating CoCoME to the cloud. On the other hand, it *attacks* the *Alternative* to optimize the current architecture.

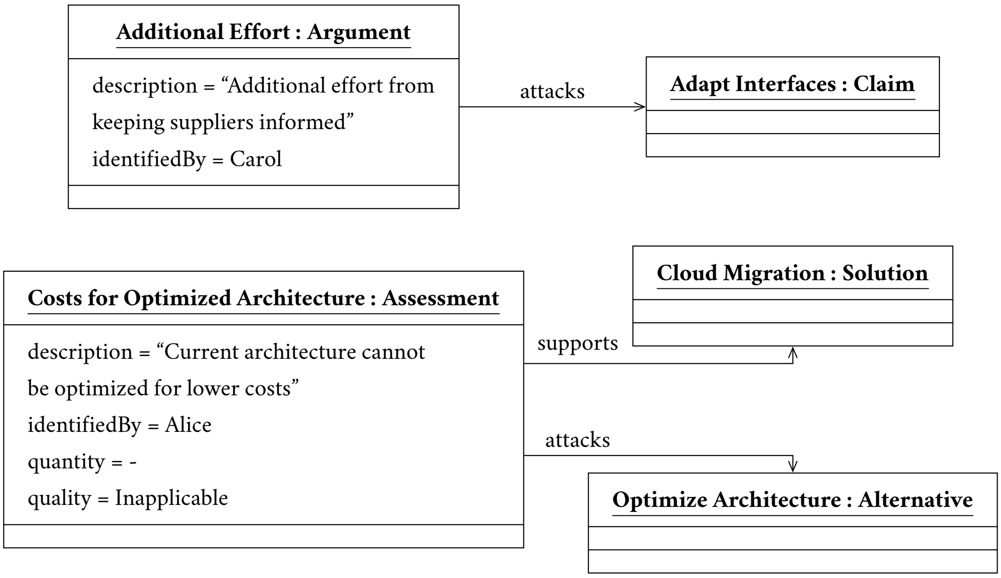

Additional Effort : Argument
description = "Additional effort from keeping suppliers informed"
identifiedBy = Carol
attacks
Adapt Interfaces : Claim
Costs for Optimized Architecture : Assessment
description = "Current architecture cannot be optimized for lower costs"
identifiedBy = Alice
quantity = -
quality = Inapplicable
supports
Cloud Migration : Solution
attacks
Optimize Architecture : Alternative

Figure 5.12: Examples for *Argument* and *Assessment*

5.3.7 Relations between DecisionComponents

Several types of links exist to specify relations between *DecisionComponents*. An overview is given in Figure 5.13. The principal idea behind these relations is to specify how the evaluation of a *Solution* towards a given *Question* can be documented. First, *Questions* may result from or depend on a certain *Context*, which represents a criterion for potential solutions to this question. To document this

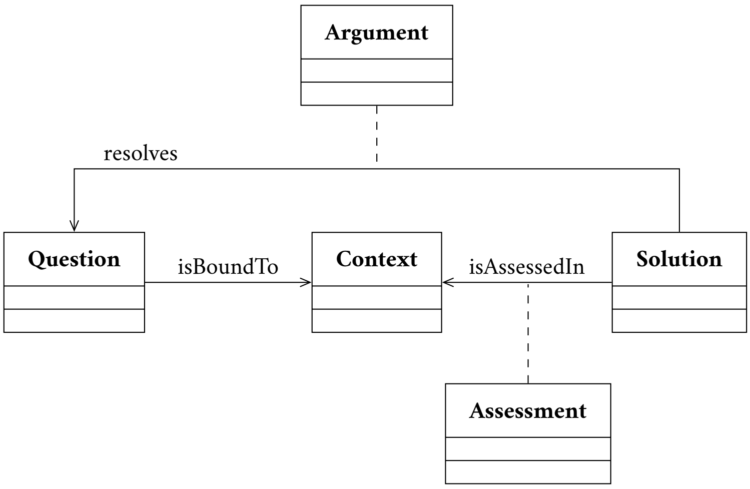


Figure 5.13: Relations between DecisionComponents

situation, the `isBoundTo`-relation can be used. Second, a *Solution* can be evaluated for its capability to fulfill a criterion. Then, it is linked via the *isAssessedIn*-relation to the respective *Context* element. Details on the assessment may be provided by associating an *Assessment* to this relation. In case that a solution for a given problem was identified and selected, the corresponding *Solution* element can be linked with the underlying *Question* using the `resolves`-relation. Any key reason supporting this selection can be documented through an associated *Argument*.

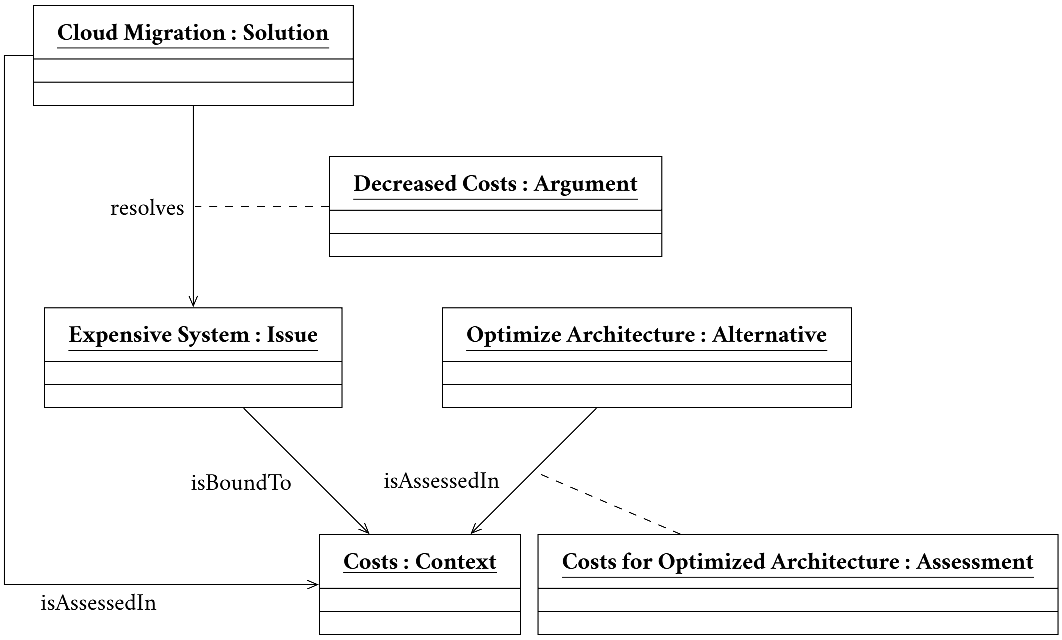

Cloud Migration : Solution
resolves
Decreased Costs : Argument
Expensive System : Issue
Optimize Architecture : Alternative
isBoundTo
isAssessedIn
Costs : Context
Costs for Optimized Architecture : Assessment
isAssessedIn

Figure 5.14: Example of Relations between DecisionComponents

Based on the running example, instances of knowledge elements with the presented relations are modeled in Figure 5.14. Here, the currently expensive CoCoME system is an *Issue*, which is obviously characterized by the *Context* of costs. Thus, a potential solution should reduce the costs for running the system. The *Solution* to migrate CoCoME to the cloud is assessed with regard to this criterion. It `resolves` the issue, as it is found to reduce the costs for CoCoME sufficiently. In contrast, the *Alternative* to optimize the current architecture does not fulfill the criterion. It should be noted that these relations are capable to capture any evaluation of *Solution* elements that was performed by developers. In consequence, knowledge on criteria and their related *Assessments* may become incomplete or inconsistent over time.

5.3.8 Model of Running Example

The detailed introduction of all documentation model elements and relations with their respective excerpts of the running example in the previous subsections is summarized by an overview of the entire example presented in Figure 5.15. Here, all instances of model elements with their relations are depicted based on the example. In addition, it is marked which developer contributed an element to the documentation. For instance, the original issue of CoCoME being to expensive with the current architecture was brought up by Alice. Thus, it is one of top level elements within the decision to migrate CoCoME to the cloud. Then, further elements are added to this issue, like context knowledge on the costs, a solution for migrating to the cloud, and so on. Thereby, the interplay between *Decision* and *DecisionComponent* via the composite relation `contains` is shown (visualized as composite relation). This highlights how the documentation model supports the iterative documentation of decision knowledge, as the model elements and their instances are able to form knowledge trees using `contains`. This nested tree structure represents the origin of the knowledge element instances in the context of the given decisions. As described in Section5.3.2, all instances of *DecsisionComponent* depend on the existence of their respective root element, which contains them. For instance, the *Constraint* to inform suppliers about interface adaptions is contained within the *Claim* to adapt the interfaces, as this constraint would not have been brought up and, therefore, could not exist without the related claim. The tree structure is complemented by relations between the knowledge elements, e.g. , that the *Assessment* of costs for an optimized architecture actually supports the *Solution* to migrate to the cloud. Thereby, instances contained in different root instances are related to each other. Thus, the documentation allows for a structural refinement of documented knowledge using the `contains`-relation, as well as refinement relations between given knowledge element instances are provided. Furthermore, it becomes visible how different developers add documented knowledge for the example decision collaboratively over time. For instance, Alice adds the *Assessment* on the costs for an optimized architecture after Bob has documented his *Alternative*, and Carol adds a *Question* on missing integration with the suppliers' systems to the *Solution* of a cloud migration identified by Alice.

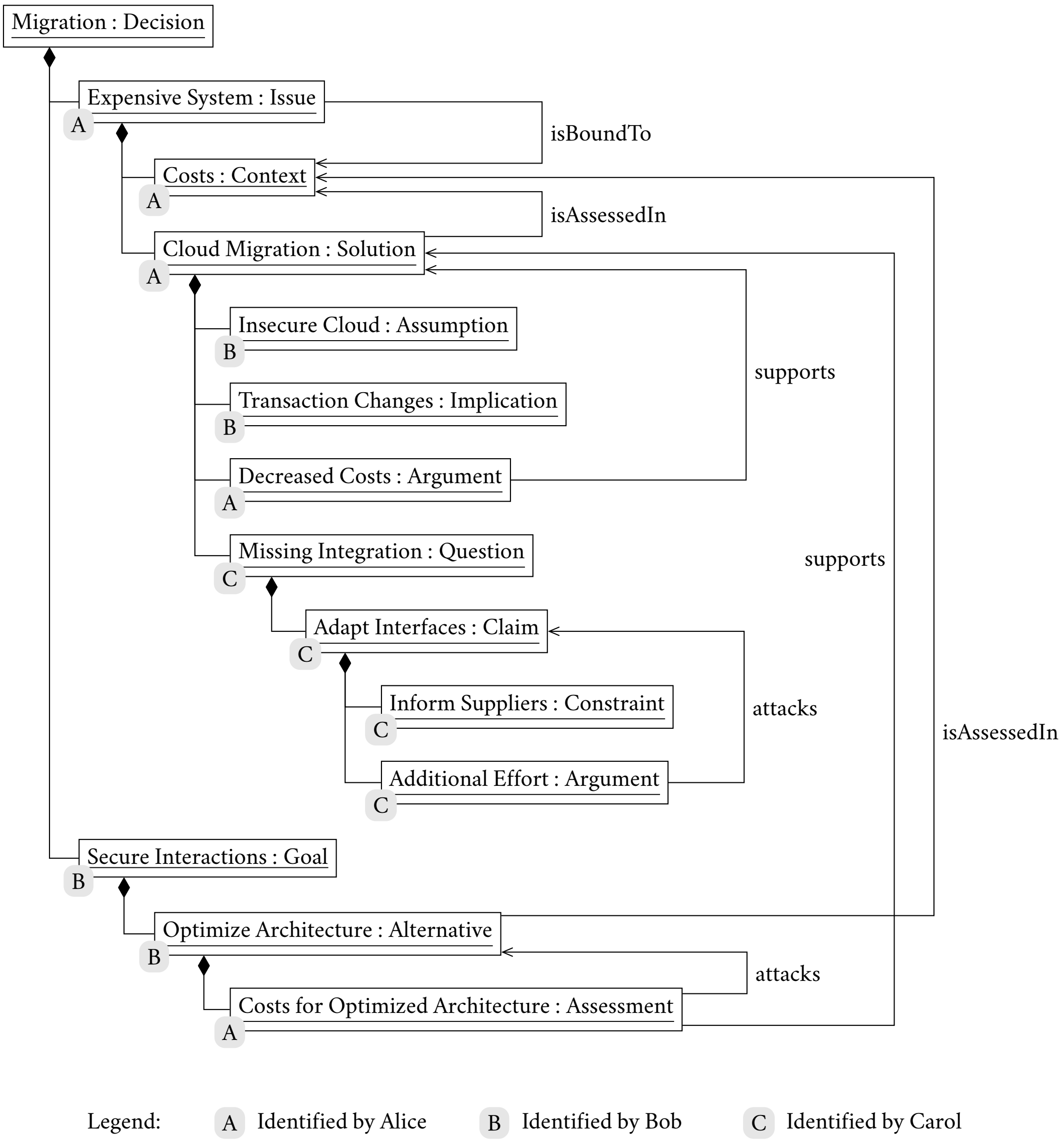

Migration : Decision
Expensive System : Issue
A
isBoundTo
Costs : Context
A
isAssessedIn
Cloud Migration : Solution
A
Insecure Cloud : Assumption
B
supports
Transaction Changes : Implication
B
Decreased Costs : Argument
A
Missing Integration : Question
C
supports
Adapt Interfaces : Claim
C
Inform Suppliers : Constraint
C
attacks
isAssessedIn
Additional Effort : Argument
C
Secure Interactions : Goal
B
Optimize Architecture : Alternative
B
attacks
Costs for Optimized Architecture : Assessment
A
Legend:
A Identified by Alice
B Identified by Bob
C Identified by Carol

Figure 5.15: Model of the Running Example (without Associations for Associated Classes)

As Alice, Bob, and Carol serve in different roles within the running example, they collaboratively document the migration decision when performing different development activities. The integration of the documentation model with these activities is described in the following section.

5.4 Integration with Development Activities

The documentation approach presented in this thesis is integrated with different development activities in two ways. First, the `concerns` relation is used to link *Decisions* and *DecisionComponents* with knowledge resulting from requirements engineering, design, and implementation. An overview of these links is given in Figure 5.16. Knowledge from development activities is typically codified within development artifacts. For instance, use cases may be the result of requirements engineering, UML diagrams can be created during design, and code files will most likely result from implementation (cf. Section 2.2). Thus, linking decision knowledge with different development artifacts is an important step towards the integration of decision documentation with the respective development activities. In consequence, the presented documentation approach is refined and adapted to provide these flexible and meaningful links in order to fulfill requirement C. Therefore, a semi-automatic capturing mechanism for decision knowledge from a heuristic security analysis of use cases and two manual capturing mechanisms for decisions during UML design and code implementation are presented in this thesis. It should be noted that these mechanisms demonstrate the principal idea of capturing decision knowledge to support developers. Similarly to these mechanisms, further approaches for knowledge capturing might be added using the documentation model in the future. Second, tool support for linking decision knowledge and development artifacts is required, so that developers benefit from these links by decreased documentation effort and easier collaboration during documentation. This is addressed in Section 6.2.

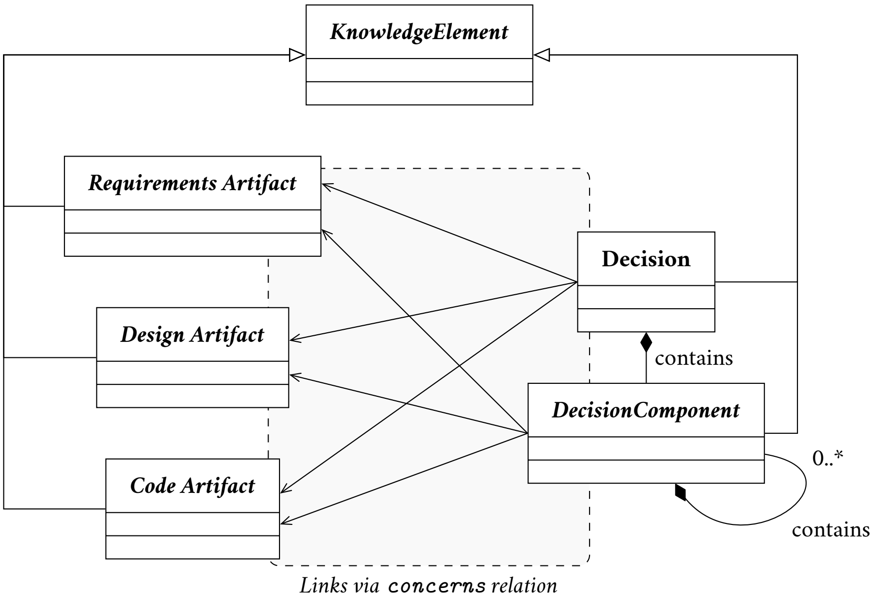


Figure 5.16: Integration of Documentation Model with Development Activity Artifacts

5.4.1 Requirements Engineering: Decisions on Security Requirements

The semi-automatic capturing of decision knowledge during requirements engineering is performed for security requirements. This approach was previously described in [Hesse, Gaertner, et al. 2014].

Security requirements are of crucial importance for long-living information systems, as they have to be checked and updated regarding new security issues. For instance, vulnerabilities discovered in systems built previously should be addressed in given security requirements for a new system. Developers need to address these security issues by making decisions on how to adapt and implement the related security requirements. A structured documentation of this decision knowledge helps developers to access and review the decisions, as security requirements are subject to monitoring and change even after deployment [Hesse, Gaertner, et al. 2014]. Also, new security issues are typically reflected in emerging expert knowledge. For instance, in order to mitigate these issues, legislation may change, or developers may adapt their patterns. In consequence, the amount of knowledge related to security requirements is large compared to other kinds of requirements. To keep track of this knowledge on security requirements, it is useful for developers to reflect them within decision documentation [Hesse, Gaertner, et al. 2014]. Thus, this kind of requirements was selected to be integrated with the presented decision documentation model.

Heuristic Analysis with HeRA

For natural language requirements, vulnerabilities can be identified using the security heuristics tool HeRA to analyze the textual specifications automatically. A detailed description of this tool, which is developed by the Software Engineering group at Leibniz Universität Hannover, can be found in [Gärtner et al. 2014]. HeRA searches use case descriptions for vulnerabilities based on reported incidents, which are attempts to violate security policies or to gain unauthorized access to data [Hansman and Hunt 2005]. Knowledge on incidents used for HeRA is modeled according to the knowledge structure given in Figure 5.17. Here, an incident is described by actions in order to gain access to assets via several entry points. The resulting vulnerabilities are assigned to incidents, which may refer to further expert security knowledge, such as textbook knowledge, security obligations, and experiences [Hesse, Gaertner, et al. 2014]. With these modeled incidents, HeRA assesses the

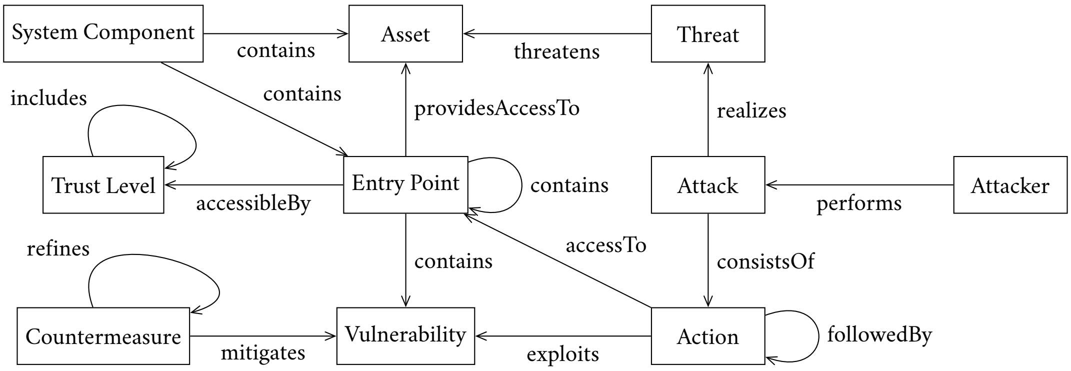


Figure 5.17: HeRA Knowledge Structure [Gärtner et al. 2014]

flow of events within use cases by scanning it for relevant elements of modeled incidents and their chronological order. Hereby, the elements found in the use case steps are matched heuristically with elements modeled for the incident. If the comparison yields a positive result, the modeled incident has been identified within the use case. This indicates a potential security vulnerability [Hesse, Gaertner, et al. 2014]. However, it should be noted that the developers need to review the findings in order to decide whether the identified use case steps are actually vulnerable to security breaches. In case a vulnerability was found by the developer, decisions should be taken based on the heuristic findings to mitigate the vulnerability. Then, decision knowledge is acquired, which can be imported semi-automatically using the HeRA results and the documentation approach.

Import Decision Knowledge from HeRA Results

After a use case was scanned with HeRA, the analysis results can be classified according to their outcome. This is shown in Table 5.1. In case that a modeled incident did not match the analyzed use case (negative match), obviously no further semi-automatic support can be provided, and further checks for missing identifications of security issues need be carried out manually. However, in case that the heuristic analysis produced positive matches with modeled incidents, the heuristic results highlight important decision points for addressing potential security issues. Of course, a positive match may be a false positive, which has to be dismissed during manual check-up. Then, typically no further action or decision is required. If, however, a true positive match was found (category 1A and 1B in Table 5.1), developers can be supported in capturing decision knowledge, as the heuristic findings highlight decision points regarding the analyzed use cases. Thereby, the documentation effort for decisions is decreased, because knowledge on incidents can be incorporated as decision knowledge.

Category	Machting Result	Actions by Developers
1A	True positive match: The use case contains a security issue, and all use case contents were matched correctly to incident knowledge	Decisions need to be made and documented in order to address the security issue
1B	Partial true positive match: The use case contains a security issue, but not all use case contents were matched appropriately to incident knowledge	Decisions need to be made and documented to address the security issue, but it is likely that the documentation structure derived from the incident knowledge needs to be adapted manually
2	False positive match: The security issue was identified for the use case by mistake	The result can be discarded, no decisions need to be taken
3	Negative match: The security issue was not identified within the use case, either because it is actually not relevant, or because the matching failed for the use case	Manual check after heuristic analysis for potentially unidentified security issues

Table 5.1: Developer Actions for a Heuristic Finding

The process of incorporating incident knowledge as documented decision knowledge is illustrated using the running example CoCoME. Therefore, a use case of CoCoME is considered, which specifies the sales process within physical stores using the CoCoME Trading System. Here, the customer pays for multiple goods from the store at a cash desk. Requirements engineer Bob is worried that moving CoCoME to the cloud will result in vulnerabilities for this use case, as personal data and money transactions are exchanged between the customer and the cashier within this process. Therefore, the cashier is required to be authenticated in the system and has to be authorized to perform these transactions. It is assumed that for this use case a heuristic analysis using HeRA was performed by requirements engineer Bob. Within HeRA, an incident regarding the missing authentication of sensitive persons was modeled. This incident was now identified within the use case analyzed by Bob. The results of the heuristic match are shown in Table 5.2.

Incident Knowledge	**Content for Running Example**
System Component	CoCoME Trading System
Asset	Personal data, Money
Entry Point	Sale Finished, Start New Sale, Cash Desk
Trust Level	Customer, Cashier
Threat	Authentication missing or unreliable
Attack	Unauthorized money transfer, Unauthorized access to personal data
Attacker	Inside or outside attacker (unknown)
Vulnerability	Invalid authentication
Countermeasure	Single sign-on using an external provider

Table 5.2: Heuristic Match for Running Example

To incoporate knowledge on identified security issues of category 1A and 1B into documented decision knowledge, three different steps are necessary [Hesse, Gaertner, et al. 2014]:

1. *Identify potential decisions:* All identified security issues are grouped according to their origin in the analyzed use case and their vulnerability.

2. *Map incident and decision knowledge:* Knowledge modeled for the incident within each match is mapped to appropriate decision knowledge elements.

3. *Complete decision documentation:* Missing knowledge for decisions on the identified security issues is elicited and added to related elements of the respective decision.

The first step is intended to ensure that the uncovered security issues are grouped into meaningful decisions and linked to all use cases they were identified in. Therefore, one decision is created automatically for each security issue, which is correct for all security issues belonging to category 1A and 1B. However, developers may need to adapt these decisions. In detail, developers may discard all decisions proposed for security issues belonging to category 2. Also, if decisions on similar security issues were already made, those decisions should be linked to the investigated use case instead of creating new decisions. Therefore, existing decisions need to be determined automatically, and the security issues are filtered accordingly. In the running example, Bob decides to document a new decision on the identified security issue of missing authentication within the sales process.

In the second step, the content of each heuristic finding is mapped to decision knowledge elements. Based on this mapping, instance proposals for these knowledge elements with textual descriptions and links to the related requirements are created. The default mapping with generated contents for the running example is presented in Table 5.3. The *Threat* to a *System Component* describes the decision related to the identified security issue, and is created as *Decision*. This decision is linked via the *concerns*-relation to the use case the heuristic finding originates from. *Attack* and *Attacker* provide details on the underlying decision problem based on the potential attack, and are mapped to *Issue*. This element is linked via the *concerns*-relation to all use case steps, which are vulnerable to the identified attack. A general *Context* of the decision is provided by *Entry Point* and *Asset*, as they describe the exploited shortcomings and target of a potential attack. Furthermore, an *Assumption* can be made on the circumstances of a potential attack, for which an attacker needs to exploit the identified *Entry Point* at a certain *Trust Level*. Finally, several *Alternatives* are given within the heuristic findings, as HeRA proposes *Countermeasures* based on the identified *Vulnerabilities*. It should be noted that developers need to decide in the second step, whether the default mapping is appropriate for each decision to be documented. Otherwise, changes to the mapping have to be made manually. For instance, Bop could decide to map *Entry Point* and *Trust Level* on a *Constraint* instead of an *Assumption*,

Incident Knowledge	**Decision Knowledge**	**Generated Content for Running Example**
Threat, System Component	Decision	*Authentication missing or unreliable* in *CoCoME Trading System*, *concerns*-relation to affected use case
Attack, Attackers	Issue	*Unauthorized money transfer, Unauthorized access to personal data* might be performed by *Inside or outside attacker (unknown)*, *concerns*-relation to vulnerable use case steps
Entry Point, Asset	Context	*Sale Finished, Start New Sale, Cash Desk* holds the asset(s) *Personal data, Money*
Entry Point, Trust Level	Assumption	*Sale Finished, Start New Sale, Cash Desk* is vulnerable by the role(s) *Customer, Cashier*
Vulnerability, Countermeasure	Alternative	*Invalid authentication* can be mitigated by *Single sign-on using an external provider*

Table 5.3: Default Mapping between Incident Knowledge and Decision Knowledge

because he is convinced that those are the only roles involved in a potential attack within this use case.

In the third step, the generated proposals for textual contents and links of decision knowledge elements can be adapted and refined manually if necessary. In particular, this is necessary if a decision is documented for a heuristic match belonging to category 1B. In this case, the heuristic matching between the incident and the examined use case was not appropriate for all use case steps. This may happen if the incident was not modeled in detail, or if a finding is not appropriate in all contexts due to language ambiguities. For instance, the term "enter" might have been found in a use case step. On the one hand, this could indicate the potential vulnerability of date entered by customer, for instance a PIN code for making a money transfer. On the other hand, also the cashier could simply enter a different mask to change the price of a sold good. To clarify such ambiguities, developers can check and decide finally in this step, which instance of decision knowledge elements are created and whether they are linked correctly. Furthermore, the generated descriptions might be incomplete, because not all relevant knowledge was modeled for the identified incident. For instance, if no particular entry point was specified for a system component, or if a countermeasure is missing, developers need to add this information manually.

In summary, the presented steps allow developers to import decision knowledge during requirements engineering, when they incorporate knowledge on incidents into their documented decisions. Thereby, the capturing of decision knowledge during requirements engineering (requirement C.1) is addressed. As HeRA uncovers security issues in given use cases and their flow of events, links between decisions to address security issues and their related use cases can be created semi-automatically. The `concerns`-relation can be used to link entire decisions with use cases, and to relate individual instances of decision knowledge elements with their corresponding use case steps. In consequence, links between decisions and development artifacts are created (requirement D.2).

5.4.2 Design: Decisions on UML Design Models

To integrate the documentation model for decision knowledge with design activities, an approach is presented to create and link decision knowledge within UML design models. As a prerequisite, all model elements within UML design models are considered to be knowledge elements hold in a central knowledge repository. For instance, such a representation of UML design models is provided by UNICASE and the EMFStore (cf. knowledge management tools in Section 2.4). Then, the integration of decision knowledge and UML model elements is achieved by linking them using the *concerns*-relation. Depending on the granularity of the available UML model elements, decisions can be linked to entire diagrams, structural entities, such as components, classes, attributes, and methods, as well as to relationships within the UML model.

Regarding the tool support, it is important to provide designers with a shortcut functionality to document decision knowledge directly within their respective UML editor. Thereby, it should be distinguished whether decisions are newly created or existing ones are linked to given UML model elements. Also, multiple UML model elements might be related to a design decision, which needs to be captured within the editor. Furthermore, an initial set of typically used documentation elements should be proposed to the designer for documentation. According to the value-based approach by Davide Falessi, Capilla, and Cantone, it is perceived as beneficial by developers to capture a design decision with its decision problem, its current solution, and a rationale regarding this decision [Davide Falessi, Capilla, and Cantone 2008]. Due to various and complex changes that developers may apply to UML diagrams over time, a manual check of the related decisions is necessary. If a UML model element is deleted, the corresponding reference to decision knowledge elements is removed. As design decisions shape and guide the development process [Jansen and Bosch 2005], the related decisions are not deleted automatically. Instead, developers should check whether the `implementationStatus` and documented content of the related decisions has to be changed.

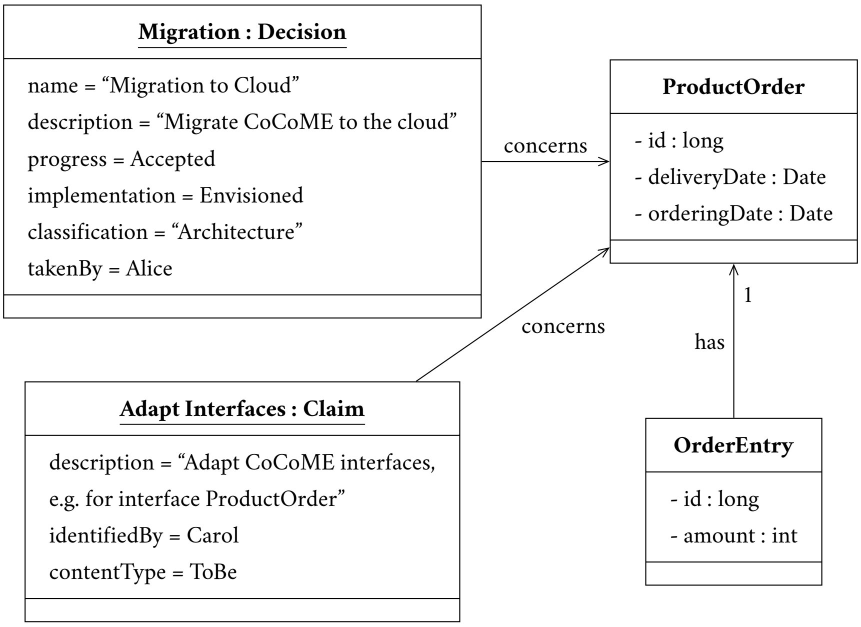


Figure 5.18: Decision Knowledge related to UML Class `ProductOrder` from CoCoME

Regarding the CoCoME running example, architect Alice and developer Carol may document their decision knowledge on moving CoCoME to the cloud within the related UML models of CoCoME. This is depicted in Figure 5.18. For instance, Alice could document the decision for all UML model elements within the current CoCoME architecture model, which cannot be optimized regarding better performance and lower costs. Using the `concerns`-relation, she can link her decision to both the entire component diagram `ProductDispatcher` in the application layer and selected UML model elements from data layer, such as the classes `ProductOrder` and `OrderEntry`. Later on, Carol may also link her *Claim* to the respective UML class.

Overall, capturing decision knowledge using UML diagrams provides direct links between decisions and their related design artifacts. Such links can be provided on different levels of granularity for both UML and decision knowledge elements. Thereby, requirements C.2 and D.2 are addressed.

5.4.3 Implementation: Decisions on Code

Based on the described documentation model, a model for code annotations is presented to integrate decision documentation into code created during implementation. This approach was previously presented in [Hesse, Kuehlwein, Paech, et al. 2015] and a related bachelor thesis [Kuehlwein 2014]. Such code annotations can be used by developers to document decisions when writing code. Thereby, a context switch between different views or even tools is avoided for developers. Moreover, documented decisions are directly embedded into the corresponding development artifact, so that they are accessible without access to a central knowledge repository and a sophisticated knowledge browser.

Regarding our running example, Carol identified the need to adapt the services of CoCoME to enable suppliers to access the new interfaces of the trading system. Thus, a follow-up decision is made to adapt the service implementation of CoCoME. For instance, this decision describes a change of the service for ordering a product, which requires a cloud-based data object `ProductOrder`. Hereby, Carol uses code annotations to document the related decision knowledge directly within the code, as depicted in Figure 5.19.

```
/**
* The ProductOrder class represents a ProductOrder of a Store in the database
* @author Carol
*
* @Decision Move sales and order services to the cloud.
* @Solution Migrate services to the cloud.
* @Question Integration with suppliers' systems missing?
* @Claim Adapt our connectors for processing product orders.
*/
```

Figure 5.19: Example of Decision Annotations in Class `ProductOrder`

Structure of Code Annotations
The principal idea is that all decision knowledge elements have a corresponding code annotation.

The only exception is *DecisionComponent*, which is an abstract knowledge element. Consequently, it cannot be instantiated directly and, therefore, does not require an annotation. Code annotations may be used in any inline code comment, including JavaDoc, to annotate class and method declarations as well as any code part within the method body. Thus, different levels of code granularity are covered. Code annotations may be employed by developers either to create a new instance of a decision knowledge element or to reference an existing one [Hesse, Kuehlwein, Paech, et al. 2015]. First, developers can type the presumed decision knowledge as text directly behind the annotation. This is depicted in Figure 5.19. Second, an annotation holds several internal attributes. These attributes are used to manage references to existing knowledge elements, to ensure persistent storage of the annotation, and to locate the annotation within code files.

To create new instances of decision knowledge elements, two different kinds of code annotations were established with different functional support for developers: core annotations and augmented annotations. *Core annotations* represent an instance of a decision knowledge element from the documentation model. For each decision knowledge element from the documentation model one core annotation is derived. For instance, `@Solution` can be used to document an *Solution*, and `@Issue` creates a new *Issue* instance. In contrast, *augmented annotations* represent one or more decision knowledge elements with default values for attributes and relations. Thereby, developers can create instances of decision knowledge elements using patterns codified within augmented annotations, such that the manual documentation effort is reduced by this shortcut. For instance, `@Contra` can be used to create a new argument as a child of the nearest *DecisionComponent* and to link the argument to this component using the `attacks`-relation. In contrast, `@Pro` creates a new argument with a `supports`-relation to its containing element.

To link code annotations with other knowledge elements and code files, administrative information is required. Therefore, both core annotations and augmented annotations inherit the *AbstractAnnotation* with several attributes for internal use, as depicted in Figure 5.20. Thus, annotations are knowledge elements themselves, and can be stored explicitly in a knowledge repository. Within *AbstractAnnotation*, the *unique identifier* of each annotation instance is stored. Furthermore, it contains the related code revision for links to code files, as well as the *knowledge revision* for links to other knowledge elements. These revisions are updated, whenever a change in code or knowledge management impacts an annotation. Thereby, consistency can be assured between annotations within a central knowledge repository and code files managed by a version control system.

It should be noted that this annotation structure suits an incremental and collaborative usage by developers [Hesse, Kuehlwein, Paech, et al. 2015]. First, if a developer misses documented decision knowledge for an already documented decision in the code, further annotations can be added easily. For instance, a new `@Alternative` could be added by Carol, when she realizes that there are further

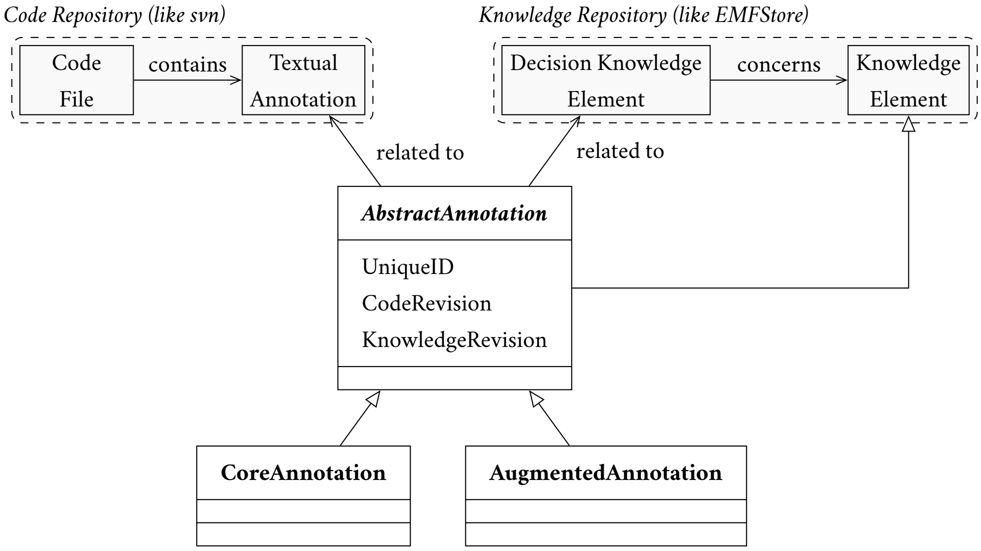


Figure 5.20: Annotation Model

solution options than only modifying the connectors within `ProductOrder`. Second, as every annotation instance can be linked to an underlying decision knowledge element, knowledge created by annotations is synchronized with the central knowledge repository. Thereby, decision knowledge originating from implementation becomes visible to developers in other roles. For instance, architect Alice might realize due to the documentation of Carol, that further constraints exist regarding the cloud migration of CoCoME.

Actions with Code Annotations

With the presented code annotations, developers may create, modify, or delete both the annotation and decision knowledge element instances [Hesse, Kuehlwein, Paech, et al. 2015]. This is summarized in Table 5.4. When a new annotation instance is added, developers may decide to either create the related instance of a decision knowledge element, or to link an existing one to the annotation instance. If, however, the instance of a decision knowledge element was created first, it can be linked to an annotation instance afterwards. Thus, there is no effect on annotations if knowledge elements are created first. In case the content of an annotation is modified within the code, the corresponding decision knowledge element needs to be updated. In turn, updates or removals of decision knowledge elements stored in the knowledge repository also require the contents of code annotations within the knowledge repository and code files to be modified or deleted. When developers delete annotation instances in code files, the related core or augmented annotation needs to be removed from the knowledge repository. In addition, the corresponding decision knowledge element is removed, if it was created through the annotation.

Action	Performed on Annotations	Performed on Decision Knowledge Elements
Create	Create new decision knowledge element or link existing one	No effect on annotation
Modify	Update decision knowledge element content and references	Update annotation and references
Delete	Delete annotation, decision knowledge element and references	Delete annotation and references

Table 5.4: Overview of Developer Actions with Code Annotations

In summary, the presented code annotations allow for documenting decision knowledge directly during the implementation of code. Thereby, requirement C.3 is addressed. In addition, the code annotations allow for linking decision knowledge with code files (requirement D.2). This is realized by inserting annotations directly into code and linking annotations with other knowledge elements.

6 DecDoc: Tool Support for the Documentation Approach

In this chapter, DecDoc as the tool support for the decision documentation model is presented. First, *requirements* for the tool support are derived from the documentation model described in Chapter 5. Second, *design and implementation* of DecDoc are outlined using the running example of CoCoME from Chapter 5. A first overview of DecDoc with several details regarding its architecture and usage was published in [Hesse, Kuehlwein, and Roehm 2016]. DecDoc can be installed via an Eclipse update site [*DecDoc Update Site* 2017]. The user manual is available online [*DecDoc User Manual* 2017].

6.1 Requirements Overview

DecDoc aims to support developers in documenting decision knowledge according to the documentation model presented in Chapter 5. Thus, the requirements for DecDoc are guided by the requirements for the documentation model and their origins in the study results described in Sections 3.3 and 4.3. In detail, the four major requirements for the documentation shall be supported within the tool DecDoc:

A. Support for the Documentation of RDM and NDM Decisions
DecDoc shall provide generic access and edit functionalities for all instances of RDM and NDM model elements, which were introduced by the documentation model (Requirement *A.1*). In detail, DecDoc is required to show an overview for a selected decision with the attributes of the *Decision* element, as well as the numbers of contained element instances within this decision. Next, it shall be possible to explore the actual structure of the contained elements by visualizing the *containedIn*-relations between the instances (Requirement *A.2*). In addition, the attributes of each further decision element need to be presented in an editable way, whenever an instance of this element is opened (Requirement *A.3*). This shall be possible for both RDM and NDM elements.

B. Support for Iterative Decision Documentation

The tool support needs to provide a version control for instances of decision model elements, such that multiple developers can document their decision knowledge within a shared knowledge repository in a consistent way (Requirement *B.1*). In addition, DecDoc is required to support basic extensions of the documentation model, so that its generic functionality for accessing and editing the documented knowledge can also be used for these extensions (Requirement *B.2*). Thereby, the abstract knowledge elements of the documentation model are supported as extension points of the model. Finally, the tool support shall provide a dedicated view for evaluating *Problem* instances in contrast to the related *Solution* instances. Open problems can thereby be identified and addressed by developers more easily (Requirement *B.3*).

C. Support for Capturing Decision Knowledge during Development

DecDoc is required to provide an integration with *HeRA*, as described in Section 5.4.1 (Requirement *C.1*). Developers are thus supported during requirements engineering, because they are enabled to import decision-related knowledge from HeRA analysis results on security-related requirements as a starting point for decision documentation. In addition, DecDoc shall support decision documentation when creating UML diagrams in Eclipse, as described in Section 5.4.2 (Requirement *C.2*). This enables developers to capture and document their design decisions as they emerge during the design activities. Moreover, DecDoc is required to enable the documentation of decision knowledge during the actual implementation of code by embedding this documentation using code annotations, as described in Section 5.4.3 (Requirement *C.3*).

D. Support for Decision Knowledge Links

The tool support shall provide a visualization functionality for both, links between decisions and links between their contained knowledge elements (Requirement *D.1*). Furthermore, DecDoc is required to propose the semi-automatic creation of links, where this is suitable and effective (Requirement *D.2*). In particular, such a semi-automatic creation shall be supported when capturing decision knowledge during requirements engineering, design, and implementation.

6.2 Design and Feature Overview of DecDoc

In principal, DecDoc is an extension to the model-based knowledge management tool *UNICASE* [*UNICASE Project* 2016], as shown in Figure 6.1. This is enabled by the plugin interface provided by the *Eclipse IDE* [*Eclipse Project* 2016]. While UNICASE itself consists of a set of plugins for the Eclipse IDE, it can be further extended by the DecDoc plugin set. Both UNICASE and DecDoc rely on the versioned model repository *EMFStore* [*EMF Store* 2016], such that knowledge element instances created by developers can be persisted, versioned, and shared. This addresses requirement B.1.

Furthermore, DecDoc is integrated with the *HeRA* plugin for Eclipse [Hesse, Gaertner, et al. 2014], so that decision-related knowledge originating from the heuristic analysis of use cases for security issues provided by HeRA may be imported to and documented in DecDoc. Also, the *Papyrus* UML editor [*Papyrus* 2016] for Eclipse is extended by DecDoc in order to document and link decision knowledge to UML design entities, such as classes or associations between classes. Finally, Eclipse *markers and code annotations* are used to document and highlight decision knowledge directly within the source code. To this end, decision knowledge versioned in the EMFStore is synchronized with annotations related to revisions from the SVN code repository.

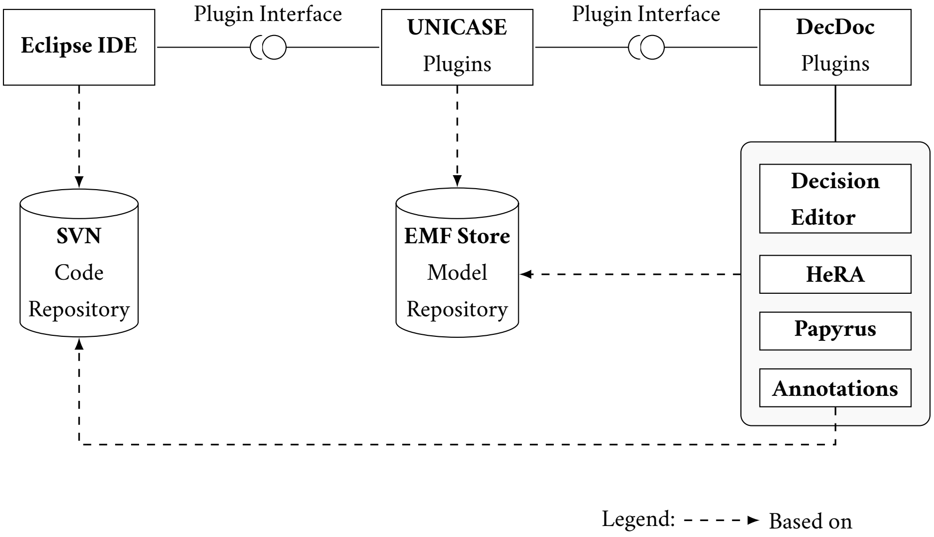


Figure 6.1: Architecture of DecDoc using the Eclipse IDE and UNICASE

Due to the architecture of DecDoc, its features can be grouped into four major parts. This feature structure is shown in Table 6.1. First, DecDoc offers a general *Knowledge Editor* with both a generic and a specialized editing support for DKEs. The editor includes the basic *Standard Editor* for all instances of DKEs, as well as a more sophisticated *Decision Editor* for instances of *Decision* elements. In addition, a graphical and statistical *Decision Overview* is provided. Furthermore, developers can explore *Question* and *Solution* instances to set *resolves*-relations in a dedicated *Solution Management* dialog for all decisions within a UNICASE project. Second, DecDoc provides a *Knowledge Importer from Heuristic Use Case Analysis*. The actual heuristic analysis plugin provides an *Issues Overview*, which presents all security issues uncovered by the heuristic analysis for a given use case in UNICASE. The *Importer Wizard* can be used by developers to structure and document decisions based on the heuristic findings and based on the question how they shall be addressed during further development. Third, the tool support enables developers to capture their design decisions, when editing an UML diagram in Eclipse Papyrus. Therefore, a specialized *Documentation Wizard* is provided. In addition, developers can use the *Design Decision Overview* to see all *Decisions* linked to UML entities in an opened UML diagram. Finally, DecDoc provides *Code Annotations for Implementation Decisions*, so that decision knowledge can be captured and documented within source code files. Therefore, the Eclipse code editor is extended by *Core and Augmented Annotations*, which developers can use to either *create* new DKE instances during implementation or *link* existing ones within their code. These tasks are supported by different pop-ups, which can be opened directly within the code editor. Furthermore, an *Annotations Overview* shows a summary of all used annotations for the opened code file. In addition, the connection between UNICASE projects and Java projects containing the source code files may be configured in different *Configuration* and *Preferences* dialogs. The appearance of the annotations within the code can likewise be configured.

Part of DecDoc	**Related Plugins**	**Features**	**Implemented As**

Knowledge Editor	`*.decision`
	`*.decision.edit`
	`*.decision.ui`
		Standard Editor
		Decision Editor
		Decision Overview
		Solution Management
			Editor
			Editor
			Editor, View
			Dialog

Knowledge Importer from Heuristic Use Case Analysis	`+.hera`
	`+.hera.wizard`
		Issues Overview
		Importer Wizard
			View
			Wizard

Capturing Support for Design Decisions	`*.decision.umlcapture`	Documentation Wizard Design Decision Overview	Wizard View

Code Annotations for Implementation Decisions	`*.decision.annotations`
	`*.decision.annotations.edit`
	`*.decision.annotations.ui`
		Core Annotations
		Augmented Annotations
		DKE Creation & Linking
		Annotations Overview
		Configuration & Preferences
			Code Annotation
			Code Annotation
			Pop-up
			View
			Dialog

Legend: * means "org.unicase", + means "de.unihannover.se"

Table 6.1: Features and Plugins of DecDoc

Each feature is realized using a specific implementation within the Eclipse platform. Whereas an *Editor* is an additional page added to the basic UNICASE model element editor, a *View* is a separate Eclipse view, which can be positioned anywhere in the Eclipse IDE. Furthermore, *Dialogs* allow for a single-page interaction with the user for monolithic tasks that cannot be further refined or structured. In contrast, *Wizards* are used where a complex task or procedure requires the user to take several steps for completion. Finally, *Code Annotations* and *Pop-ups* are visually integrated in the Eclipse code editor in order to avoid interruptions of the coding activities of the developer. It should be noted, that all related plugins given in Table 6.1 are necessary to provide all the listed features. The mandatory part for DecDoc is the knowledge editor, as it provides the knowledge model and basic user interface of DecDoc. All further parts of DecDoc, such as the code annotations and the knowledge importer, are not required to run the tool support.

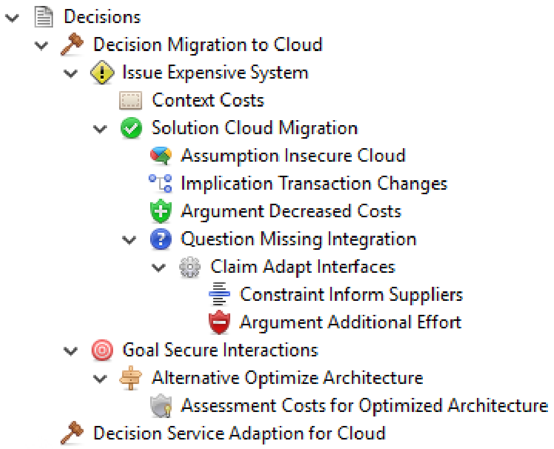
In the following sections, each part of DecDoc with its contained features is presented in more detail. Therefore, the running example of the trading system CoCoME with its related decisions is used, as described in Chapter 5.

6.3 Knowledge Editor for the Decision Documentation Model

The *Knowledge Editor* of DecDoc provides a general editing support for all DKEs. The basic knowledge model of UNICASE was used as a foundation for the DecDoc knowledge model for decision knowledge, which was modeled and implemented as ECore model. The abstract *KnowledgeElement* from the documentation model is replaced by the *UnicaseModelElement* forming the root element of the UNICASE knowledge model. Thus, in DecDoc and its knowledge editor all DKEs inherit basic attributes from the *UnicaseModelElement*, such as *name*, *description*, *creator*, and *creationDate*. Moreover, the *concerns*-relation between DKEs maps to the *annotations*-relation of the *UnicaseModelElement*. Finally, there is a technical restriction in UNICASE, that does not allow for applying associated classes within the knowledge model. Therefore, the *Assessment* element was implemented with a relation to *Context* elements as criteria for the assessment. Because duplicated names for relations are not supported in UNICASE, the *isBoundTo*-relation between *Question* and *Context* elements was renamed to *assessed*. Finally, the *containedIn*-relation for *Decisions* and *DecisionComponents* was modeled as composition. It should be noted that this underlying ECore model of DecDoc may be extended by further model elements and relations, so that requirement B.2 is addressed.

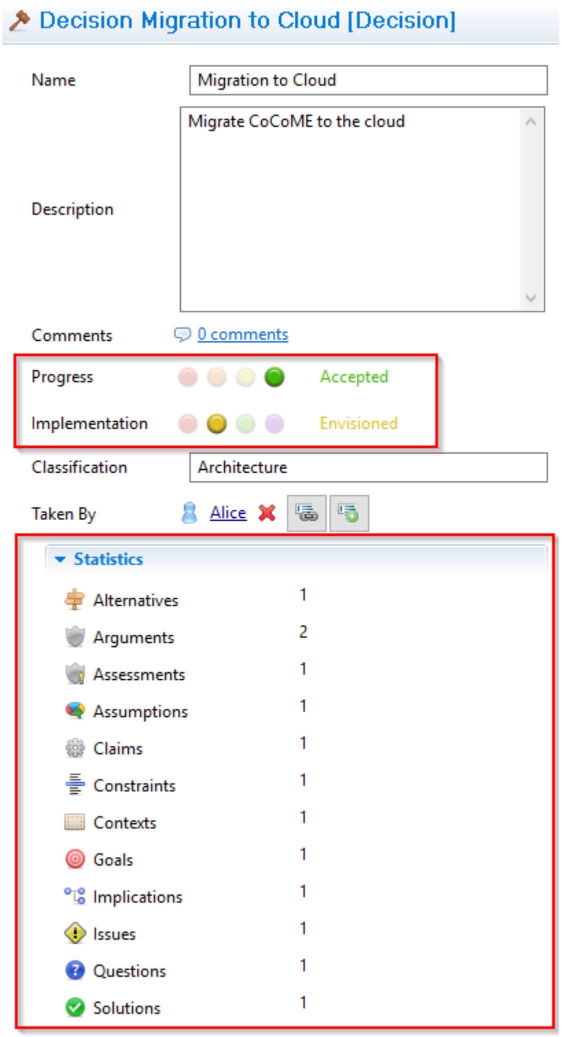
Based on this ECore model of the documentation model, developers can access and edit instances of DKEs with the basic UNICASE *Navigator*, as well as with the *Standard Editor*, which is the *Model Element Editor* of UNICASE (requirement A.3). A typical view with the Navigator on decision knowledge is shown in Figure 6.2 using the data from the running example. Here, the Navigator processes the *containedIn*-relation and depicts it as tree structure.

However, the Standard Editor of UNICASE was extended to address requirement A.3. First, a new presentation of enumeration attributes was implemented, such that developers can set the enumeration value by selecting a graphical icon. For instance, the *progress* and *implementation* attributes of *Decisions* may be set by selecting the appropriate color. Furthermore, a statistics widget was added to the editor, so that the number of contained elements can be easily retrieved for each *Decision* instance. Both extensions are depicted and highlighted in Figure 6.3. Of course, the Standard Editor also shows



- Decisions
  - Decision Migration to Cloud
    - Issue Expensive System
      - Context Costs
      - Solution Cloud Migration
        - Assumption Insecure Cloud
        - Implication Transaction Changes
        - Argument Decreased Costs
        - Question Missing Integration
          - Claim Adapt Interfaces
            - Constraint Inform Suppliers
            - Argument Additional Effort
    - Goal Secure Interactions
      - Alternative Optimize Architecture
        - Assessment Costs for Optimized Architecture
  - Decision Service Adaption for Cloud

Figure 6.2: Navigator View on Running Example Decisions



Decision Migration to Cloud [Decision]

Name	Migration to Cloud
Description	Migrate CoCoME to the cloud
Comments	0 comments
Progress	Accepted
Implementation	Envisioned
Classification	Architecture
Taken By	Alice

Statistics

Alternatives	1
Arguments	2
Assessments	1
Assumptions	1
Claims	1
Constraints	1
Contexts	1
Goals	1
Implications	1
Issues	1
Questions	1
Solutions	1

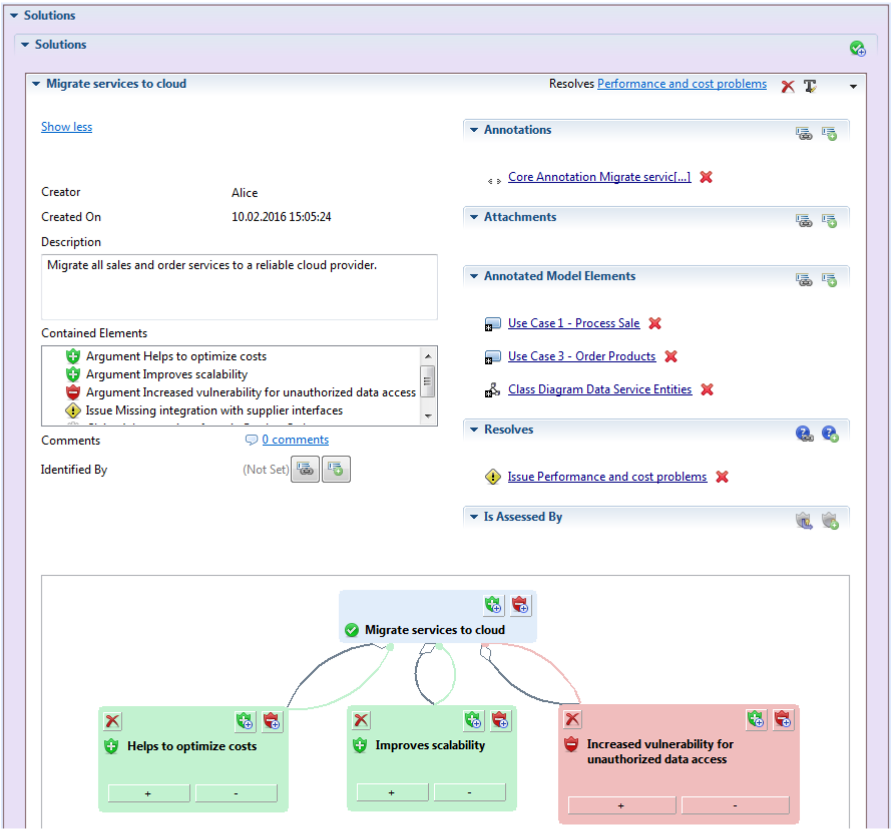
Figure 6.3: Standard View with Highlighted Enumeration Attributes and Statistics

all other attributes and relations of the currently opened model element instance, regardless whether a RDM- or NDM-related element has been opened. In addition, both the Navigator and the Standard Editor are capable of processing extensions of the ECore model of DecDoc without further need for implementation changes. Thus, requirement B.2 is fulfilled.

With the *Decision Editor*, DecDoc provides a specialized editor for *Decision* element instances. This editor is accessible as separate editor page within the *Model Element Editor*. It is depicted in Figure 6.4 with slightly modified data from the running example. Its fundamental layout consists of three panels for element instances of *Question*, *Solution*, and *Context* elements. To enable developers to distinguish them more easily, different colors were used for the panels. Each of these panels shows the respective instances. For *Questions* and *Solutions*, it is also indicated, whether the *resolves*-relation was set. All instances can be expanded in order to present all attributes and relations, as well as a graph representation for all *Arguments* linked to the currently viewed instance. This addresses requirement D.1. All displayed attributes, relations, and arguments within the graph can be edited (requirement A.3). Moreover, the Decision Editor works for both NDM- and RDM-related elements. Thereby, Standard Editor and Decision Editor address requirement A.1. It should be noted, however, that due to performance issues in the early development of DecDoc the Decision Editor only processes and displays the DKE instances, which are directly contained within the selected *Decision*. Thus, instances contained in DKEs are currently not shown, except for *Arguments* related to a displayed instance.

Next, the Knowledge Editor contains a *Decision Overview*, which consists of an extended statistical summary as Eclipse view, and an editor page for the Standard Editor with a graph visualization of all knowledge related to a selected *Decision*. The visualization is shown in Figure 6.5. In contrast to the arguments graph in the Decision Editor, the Decision Overview not only depicts instances of DKEs, but also of other decision instances and linked development artifacts, such as use case elements in UNICASE. In the figure, the decision to migrate CoCoME to the cloud from the running example is depicted with its current references. It is noteworthy that also the *isBoundTo*-relation to the succeeding decision to adapt the CoCoME service is visualized.

Finally, the Knowledge Editor of DecDoc provides a *Solution Management* dialog for a selected *Decision*. This dialog is shown in Figure 6.6 with *Questions* and *Solutions* from the example decision of the CoCoME cloud migration. Developers can use the dialog to quickly check potential *Solutions* for given *Questions*, which consist of all respective DKE instances contained in all decisions of a UNICASE project. This includes instances contained in other instances. *Solution* and *Question* instances, which are already linked by the *resolves*-relation, are highlighted in green color. Moreover, when a *Question* instance is selected, a previously linked solution element is marked with a small link icon to distinguish it from other linked *Solution* instances. Also, *Solutions* contained in the same decision as the selected *Question* instances, are displayed in bold font. This helps developers to navigate easily through



Solutions

Solutions

Migrate services to cloud — Resolves Performance and cost problems

Show less

Creator: Alice

Created On: 10.02.2016 15:05:24

Description: Migrate all sales and order services to a reliable cloud provider.

Contained Elements:
- Argument Helps to optimize costs
- Argument Improves scalability
- Argument Increased vulnerability for unauthorized data access
- Issue Missing integration with supplier interfaces

Comments: 0 comments

Identified By: (Not Set)

Annotations
- Core Annotation Migrate servic[...]

Attachments

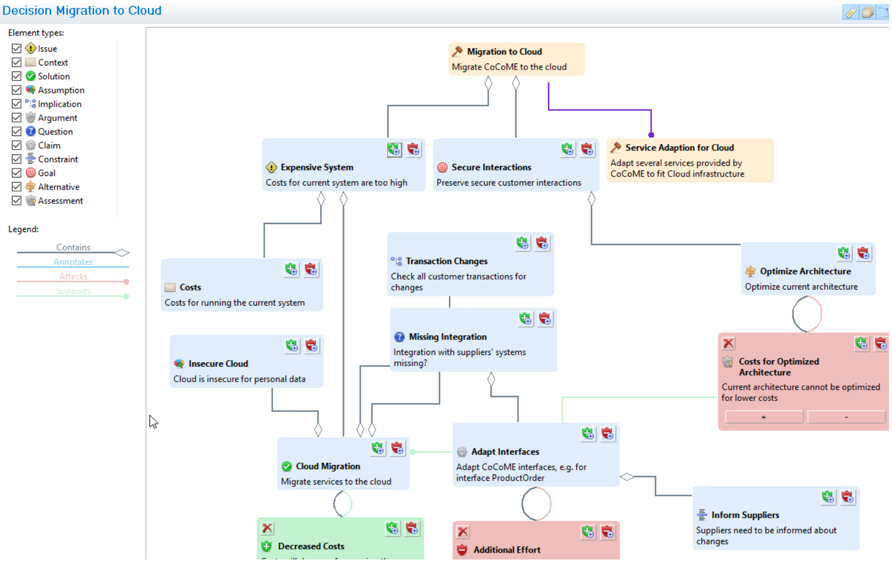
Annotated Model Elements
- Use Case 1 - Process Sale
- Use Case 3 - Order Products
- Class Diagram Data Service Entities

Resolves
- Issue Performance and cost problems

Is Assessed By

Migrate services to cloud
Helps to optimize costs
+
-
Improves scalability
+
-
Increased vulnerability for unauthorized data access
+
-

Figure 6.4: Decision Editor with Argument Graph



Decision Migration to Cloud

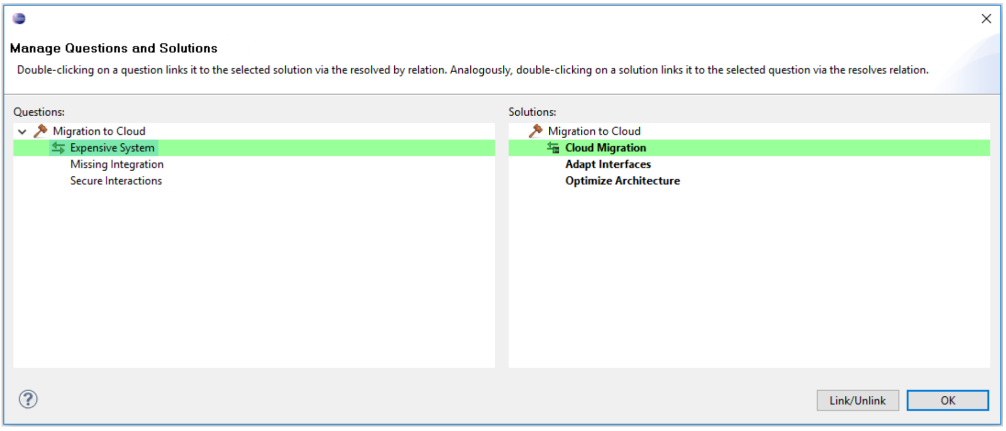
Element types:
- ☑ Issue
- ☑ Context
- ☑ Solution
- ☑ Assumption
- ☑ Implication
- ☑ Argument
- ☑ Question
- ☑ Claim
- ☑ Constraint
- ☑ Goal
- ☑ Alternative
- ☑ Assessment

Legend:

Contains
Annotates
Attacks
Supports
Migration to Cloud
Migrate CoCoME to the cloud
Expensive System
Costs for current system are too high
Secure Interactions
Preserve secure customer interactions
Service Adaption for Cloud
Adapt several services provided by CoCoME to fit Cloud infrastructure
Transaction Changes
Check all customer transactions for changes
Optimize Architecture
Optimize current architecture
Costs
Costs for running the current system
Missing Integration
Integration with suppliers' systems missing?
Costs for Optimized Architecture
Current architecture cannot be optimized for lower costs
+
-
Insecure Cloud
Cloud is insecure for personal data
Cloud Migration
Migrate services to the cloud
Adapt Interfaces
Adapt CoCoME interfaces, e.g. for interface ProductOrder
Inform Suppliers
Suppliers need to be informed about changes
Decreased Costs
Additional Effort

Figure 6.5: Graph Visualization of Decisions and Their Elements

larger amounts of *Questions* and *Solutions*. With the Solution Management dialog, requirement B.3 is fulfilled.



Manage Questions and Solutions

Double-clicking on a question links it to the selected solution via the resolved by relation. Analogously, double-clicking on a solution links it to the selected question via the resolves relation.

Questions:
- Migration to Cloud
  - Expensive System
  - Missing Integration
  - Secure Interactions

Solutions:
- Migration to Cloud
  - **Cloud Migration**
  - **Adapt Interfaces**
  - **Optimize Architecture**

Figure 6.6: Solution Management Dialog with Running Example Data

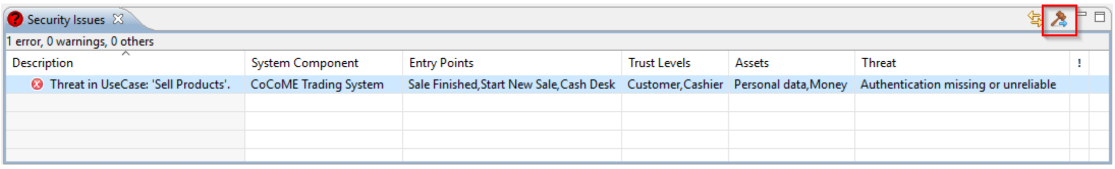
Overall, the Knowledge Editor addresses further requirements. Because the *containedIn*-relation is visualized in the navigator and the Decision Overview, requirement A.2 is fulfilled. In addition, the visualization of links between DKE instances in the argument graph of the Decision Editor and the Decision Overview fulfills requirement D.1.

6.4 Knowledge Importer from Heuristic Use Case Analysis

The *Knowledge Importer from Heuristic Use Case Analysis* enables developers to import decision-related knowledge from results produced by HeRA based on the analysis of UNICASE use cases for potential security issues.

A prerequisite for the import process is that a heuristic analysis is performed by the developers using the HeRA plugin for Eclipse [Hesse, Gaertner, et al. 2014]. As a result, the HeRA plugin creates markers whenever a potential security flaw is uncovered within a use case and its contained steps. These results are shown in the *Issues Overview*. This view shows the knowledge element instance a result was derived from, as well as the content of this result, such as the affected system component and uncovered threats. Figure 6.7 shows the result for an analysis triggered by requirements engineer Bob for the use case "Sell Products" from the CoCoME running example.

The *Importer Wizard* can be activated by clicking the `Import`-button in the issues overview for a selected result. Then, decision knowledge from the result is imported and presented to the developer in multiple steps. According to the concept described in Section 5.4.1, the wizard works semi-automatically for creating and linking instances of DKEs based on the heuristic results (requirement

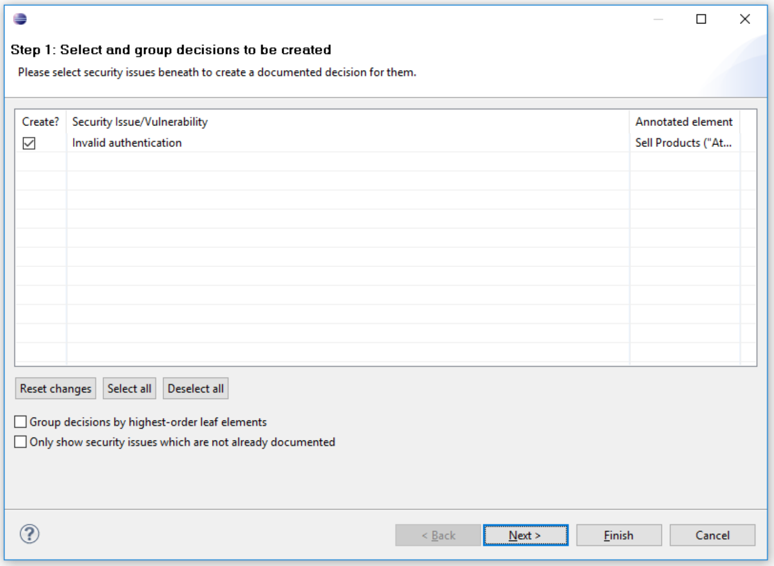


1 error, 0 warnings, 0 others

Description	System Component	Entry Points	Trust Levels	Assets	Threat	!
Threat in UseCase: 'Sell Products'.	CoCoME Trading System	Sale Finished,Start New Sale,Cash Desk	Customer,Cashier	Personal data,Money	Authentication missing or unreliable	

Figure 6.7: Issues Overview with Highlighted `Import`-Button

D.2). However, the automatically generated proposals for these instances and their links might be inaccurate due to language ambiguities, redundant content in descriptions of use case steps, and the quality of the applied heuristics. Thus, developers need to check the imported knowledge manually. First, potential decisions resulting from the heuristic findings are presented to the developers, so that they can decide on decisions to be actually created. Also, the use case instances linked to newly created decisions are shown and may be changed. This is depicted in Figure 6.8 for the situation where Bob has decided to create a new decision based on the HeRA finding of a potential invalid authentication for the “Sell Products” use case.


Step 1: Select and group decisions to be created

Please select security issues beneath to create a documented decision for them.

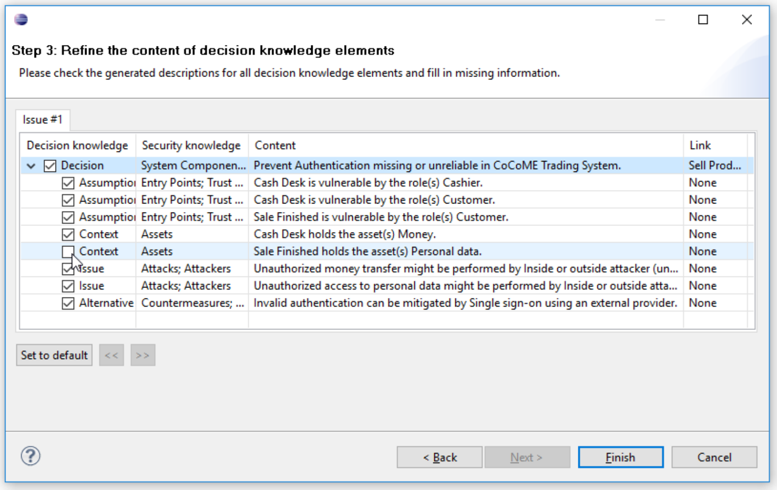
Create?	Security Issue/Vulnerability	Annotated element
☑	Invalid authentication	Sell Products ("At...

□ Group decisions by highest-order leaf elements

□ Only show security issues which are not already documented

Figure 6.8: First Step of the Importer Wizard with Potential Decisions

In the second step, developers may decide which imported knowledge they actually want to document by which type of DKE. Based on their selection, textual descriptions for all created instances of DKEs are proposed to the developers in a third step (cf. default mapping in Table 5.3 in Section 5.4.1). Here, the developers can decide to further refine the descriptions, as well as to adapt the linked element. Also, it is possible to reset current changes of DKE types, content, and links back to the default mapping. Furthermore, creation of instances can be omitted in case their content is not required or inconsistent. This is presented in Figure 6.9, where Bob decides to create only one new *Context* instance.



Step 3: Refine the content of decision knowledge elements

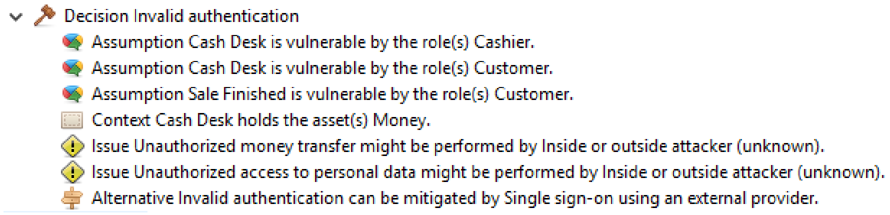
Please check the generated descriptions for all decision knowledge elements and fill in missing information.

Issue #1

Decision knowledge	Security knowledge	Content	Link
☑ Decision	System Componen...	Prevent Authentication missing or unreliable in CoCoME Trading System.	Sell Prod...
☑ Assumptioı	Entry Points; Trust ...	Cash Desk is vulnerable by the role(s) Cashier.	None
☑ Assumptioı	Entry Points; Trust ...	Cash Desk is vulnerable by the role(s) Customer.	None
☑ Assumptioı	Entry Points; Trust ...	Sale Finished is vulnerable by the role(s) Customer.	None
☑ Context	Assets	Cash Desk holds the asset(s) Money.	None
□ Context	Assets	Sale Finished holds the asset(s) Personal data.	None
☑ Issue	Attacks; Attackers	Unauthorized money transfer might be performed by Inside or outside attacker (un...	None
☑ Issue	Attacks; Attackers	Unauthorized access to personal data might be performed by Inside or outside atta...	None
☑ Alternative	Countermeasures; ...	Invalid authentication can be mitigated by Single sign-on using an external provider.	None

Figure 6.9: Third Step of the Importer Wizard with Knowledge Element Instances

When finishing the wizard, a dialog invites the developer to choose the appropriate root element for the newly created decision within the UNICASE project. After all instances have been created, the new decision instance is accessible through the basic UNICASE navigator, as depicted in Figure 6.10, and within the knowledge editor. In summary, by integrating the HeRA analysis results as starting points and knowledge suppliers for decision documentation, DecDoc fulfills requirement C.1.



Decision Invalid authentication
- Assumption Cash Desk is vulnerable by the role(s) Cashier.
- Assumption Cash Desk is vulnerable by the role(s) Customer.
- Assumption Sale Finished is vulnerable by the role(s) Customer.
- Context Cash Desk holds the asset(s) Money.
- Issue Unauthorized money transfer might be performed by Inside or outside attacker (unknown).
- Issue Unauthorized access to personal data might be performed by Inside or outside attacker (unknown).
- Alternative Invalid authentication can be mitigated by Single sign-on using an external provider.

Figure 6.10: Third Step of the Importer Wizard with Knowledge Element Instances

6.5 Capturing Support for Design Decisions

The *Capturing Support for Design Decisions* enables developers to capture their design decisions, as they create or edit UML diagrams in the Eclipse Papyrus UML editor. Therefore, DecDoc provides two features: a *Documentation Wizard* for decision documentation during UML design, and a *Design Decision Overview* to enable developers to keep track of decisions linked to the current UML diagram.

The documentation wizard for decision knowledge can be triggered directly within the Papyrus UML editor by opening the context menu for a selected UML entity. This is shown in Figure 6.11. As a first step, the developers can choose either to create a new decision with a flexible DKE structure or to link

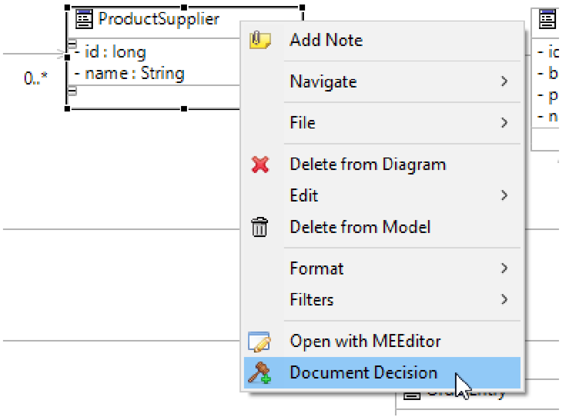
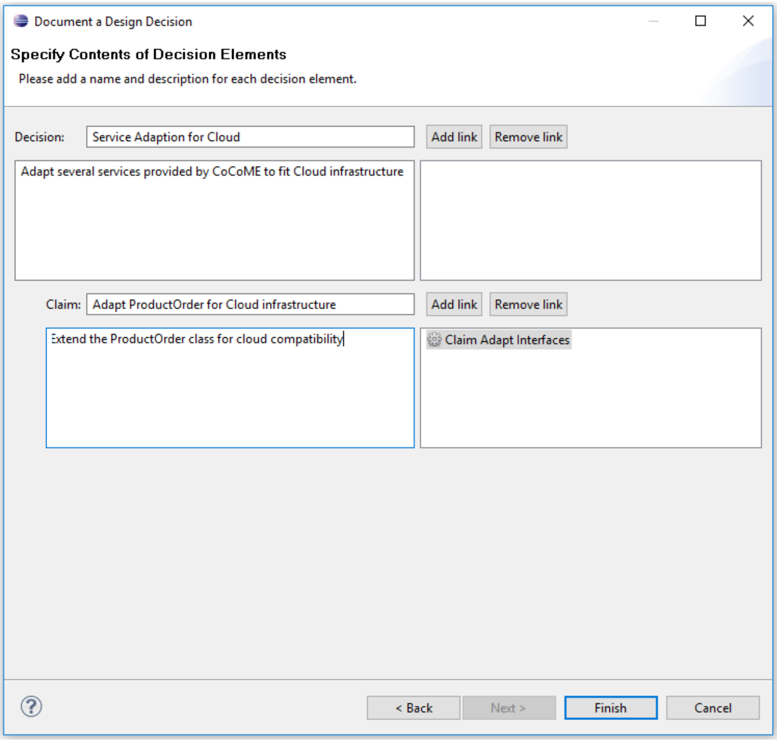


Figure 6.11: Opening the Documentation Wizard in the Papyrus UML Editor

an existing DKE instance with the selected UML model entity. For example, developer Alice may link the *Constraint* to inform the suppliers of CoCoME about the change to a cloud infrastructure with the class `ProductSupplier` from the UML diagram. Instead, Carol may decide to document her new decision to adapt the services for the cloud, while browsing through the related UML entities. Thereby, the second step of the wizard is activated, and newly created instances of DKEs are provided with names and descriptions. In particular, it is also possible to link these new instances to existing


Document a Design Decision

Specify Contents of Decision Elements

Please add a name and description for each decision element.

Decision: Service Adaption for Cloud

Adapt several services provided by CoCoME to fit Cloud infrastructure

Claim: Adapt ProductOrder for Cloud infrastructure

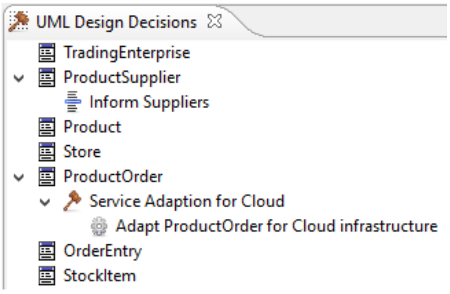
Extend the ProductOrder class for cloud compatibility

Claim Adapt Interfaces

Figure 6.12: Documenting new Decision Knowledge using the Documentation Wizard

ones. This is depicted in Figure 6.12. After finishing the second step, the new instances are created and linked to the selected UML entity according to the developers input.

To get an overview of decision knowledge linked to a particular UML diagram, the design decision overview can be used. This view shows a tree structure with the UML entities of the diagram as root entries. On the deeper levels of the tree, all linked instances of DKEs are presented as leafs. This is shown in Figure 6.13 with data from the running example: Alice and Carol have documented their decision knowledge successfully with the documentation wizard. Now, both the linked existing constraint and the newly documented decision have been related to the `ProductSupplier` and `ProductOrder` classes.



```
UML Design Decisions
  TradingEnterprise
v ProductSupplier
    Inform Suppliers
  Product
  Store
v ProductOrder
  v Service Adaption for Cloud
      Adapt ProductOrder for Cloud infrastructure
  OrderEntry
  StockItem
```

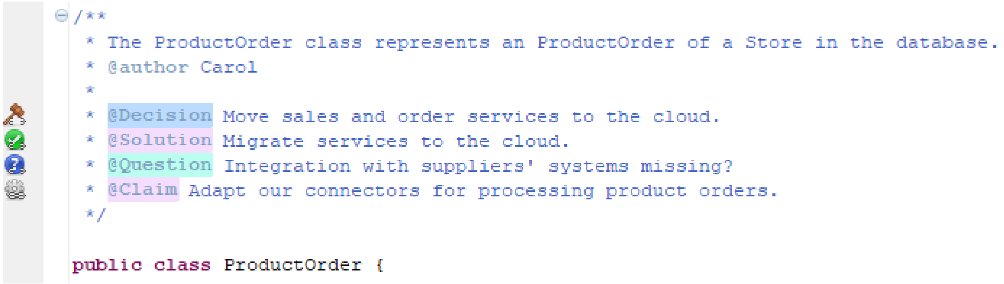
Figure 6.13: Design Decision Overview with Running Example Data

Thereby, DecDoc fulfills the requirement to integrate a decision knowledge capturing support during UML design in Eclipse (requirement C.2). Moreover, the abilities of the wizard to support the creation of links between UML entities as development artifacts and decision knowledge are addressing requirement D.2.

6.6 Code Annotations for Implementation Decisions

With *Code Annotations for Implementation Decisions*, developers can document their decision knowledge directly within the source code files they are currently working on. DecDoc provides the feature of *Core and Augmented Annotations*, which can be used within Java source code. Their appearance and functionality was implemented in the style of the commonly used Javadoc annotations [*Javadoc Documentation by Oracle* 2016]. An example is given in Figure 6.14, which shows developer Carol from the running example documenting her service adaption decision within the code file of the class `ProductOrder`.

Core annotations can be used to link existing knowledge elements with the code, and can also be used to create new instance of DKEs. In both cases, an instance of *Core Annotation* is created, as described in Section 5.4.3. This instance stores information on the relation between DKE instance and code

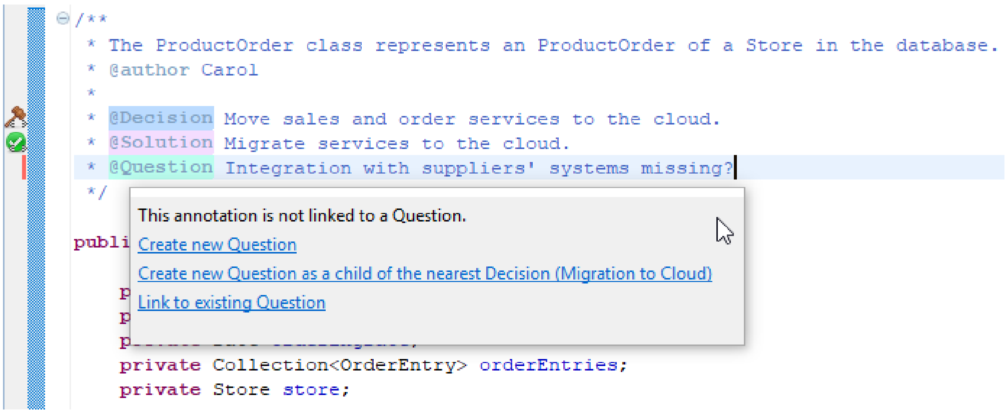


```
/**
 * The ProductOrder class represents an ProductOrder of a Store in the database.
 * @author Carol
 *
 * @Decision Move sales and order services to the cloud.
 * @Solution Migrate services to the cloud.
 * @Question Integration with suppliers' systems missing?
 * @Claim Adapt our connectors for processing product orders.
 */

public class ProductOrder {
```

Figure 6.14: Core Annotations in the `ProductOrder` Class

annotation. In the example depicted in Figure 6.14, the given annotations `@Decision`, `@Solution`, and `@Question` link existing knowledge elements from the cloud migration decision originally taken by Alice. In contrast, Carol uses the `@Claim` annotation to create a new *Claim* instance with the textual description given by the annotation content for her service adaption decision. The choice to either link existing knowledge elements or create new ones is presented to developers as soon as they save the code file with a newly added annotation. Therefore, a *Creation and Linking* pop-up is used to guide the developers efficiently through the process. An example for the core annotation `@Question` is depicted in Figure 6.15.



```
/**
 * The ProductOrder class represents an ProductOrder of a Store in the database.
 * @author Carol
 *
 * @Decision Move sales and order services to the cloud.
 * @Solution Migrate services to the cloud.
 * @Question Integration with suppliers' systems missing?
 */

publi

    p
    p
    p
    private Collection<OrderEntry> orderEntries;
    private Store store;
```

This annotation is not linked to a Question.

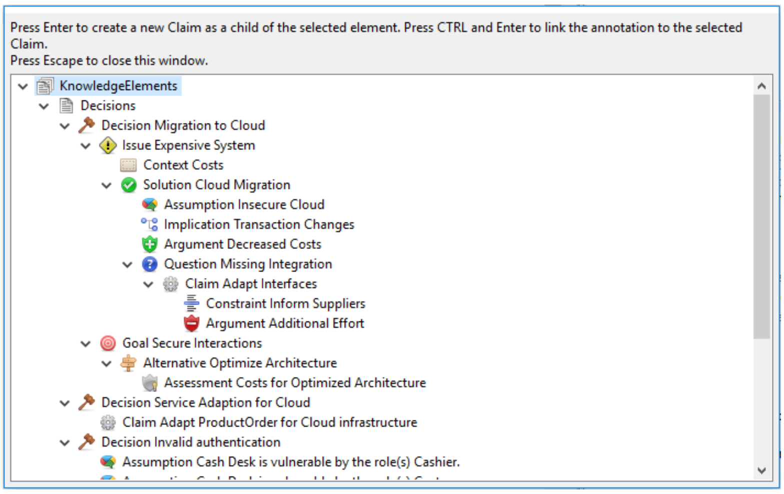
Create new Question

Create new Question as a child of the nearest Decision (Migration to Cloud)

Link to existing Question

Figure 6.15: Creation and Linking Pop-up for Core Annotation `@Question`

Regarding the running example, it was chosen to link the depicted annotation with the existing *Question* instance in the cloud migration decision. The appropriate instance has to be selected out of a tree view on the related UNICASE project with its contained decisions and their elements. Instead, new knowledge element instances can be created using the pop-up. Then, in one alternative, DecDoc proposes to create the new instance as a child instance of the nearest preceding *Decision* instance linked in the code file, which addresses requirement D.2. Otherwise, the appropriate location can be manually selected in the tree view. An example of this tree view is depicted in Figure 6.16 for the creation of the new *Claim* instance by Carol in the running example.



Press Enter to create a new Claim as a child of the selected element. Press CTRL and Enter to link the annotation to the selected Claim.
Press Escape to close this window.

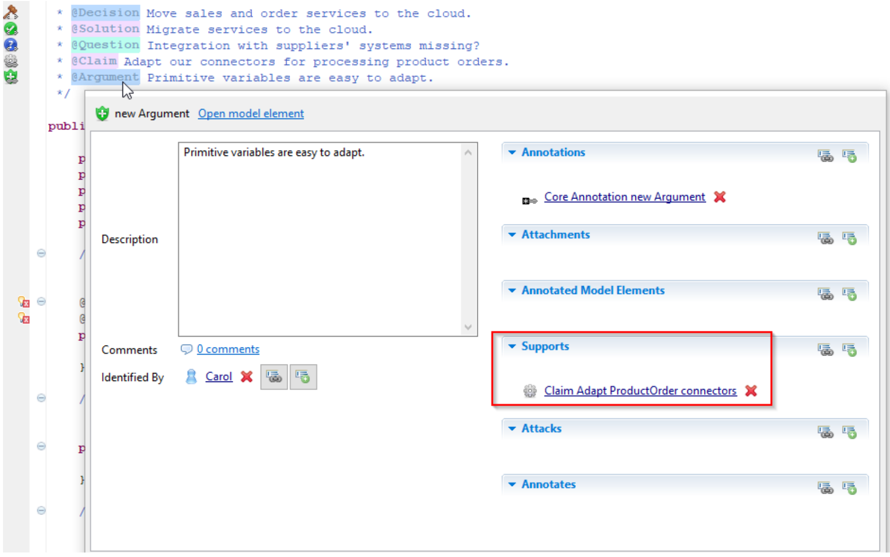
- KnowledgeElements
  - Decisions
    - Decision Migration to Cloud
      - Issue Expensive System
        - Context Costs
        - Solution Cloud Migration
          - Assumption Insecure Cloud
          - Implication Transaction Changes
          - Argument Decreased Costs
          - Question Missing Integration
            - Claim Adapt Interfaces
              - Constraint Inform Suppliers
              - Argument Additional Effort
      - Goal Secure Interactions
        - Alternative Optimize Architecture
          - Assessment Costs for Optimized Architecture
    - Decision Service Adaption for Cloud
      - Claim Adapt ProductOrder for Cloud infrastructure
    - Decision Invalid authentication
      - Assumption Cash Desk is vulnerable by the role(s) Cashier.

Figure 6.16: Tree View for Creating New DKE Instances by Core Annotations

Furthermore, developers may use *Augmented Annotations* to create new instances of DKEs with predefined relations and contents. For instance, *Arguments* can be instantiated by using the `@Pro` and `@Con` annotations. Thereby, either a supporting or attacking argument is created as child of the last annotated decision model element instance in the code file. When the code file with the augmentation is saved, it is automatically transformed into the related core annotation, and the underlying instance of *Argument* is created. A *supports*- or *attacks*-relation is subsequently set according to the type of augmented annotation that was used. Also, the content of the annotation is set as description for the newly created instance. An example is depicted in Figure 6.17, after Carol saved an `@Pro` annotation to document that the adaption for the class `ProductOrder` can be easily performed for the primitive variables.

Another important feature of DecDoc is the *Annotations Overview*, which allows developers to keep track of their annotated decision knowledge within code files. This view shows all annotations linked to a selected DKE instance. It is depicted in Figure 6.18, and shows the annotation for the newly created *Claim* by Carol. Besides information on the annotated code file and the exact location of the annotation within the file, the view also indicates, whether an annotation was used to create the related DKE instance. This is shown in the column "Hard Linked?". For Carols *Claim* it indicates "Yes", as the claim was created based on her annotation. Thus, the instance will be removed in case the annotation is deleted from the code file. Otherwise, the annotation would be linked to a previously existing DKE instance; even if the annotation was removed, the instance would nevertheless persist.

Finally, DecDoc provides two important *Configuration and Preferences* dialogs, where developers can adapt DecDoc according to their needs. First, developers may adjust the appearance of the code annotations for decision knowledge in an Eclipse *Preferences* dialog, as shown in Figure 6.19. Second, they can configure the relation between UNICASE projects containing decision knowledge and



```
 * @Decision Move sales and order services to the cloud.
 * @Solution Migrate services to the cloud.
 * @Question Integration with suppliers' systems missing?
 * @Claim Adapt our connectors for processing product orders.
 * @Argument Primitive variables are easy to adapt.
 */
```

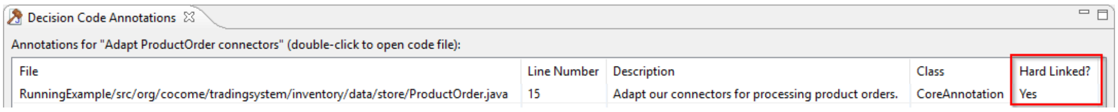
new Argument Open model element

Description: Primitive variables are easy to adapt.

Comments: 0 comments

Identified By: Carol

Figure 6.17: Pro-Argument with Highlighted *Supports*-Relation Resulting from `@Pro`

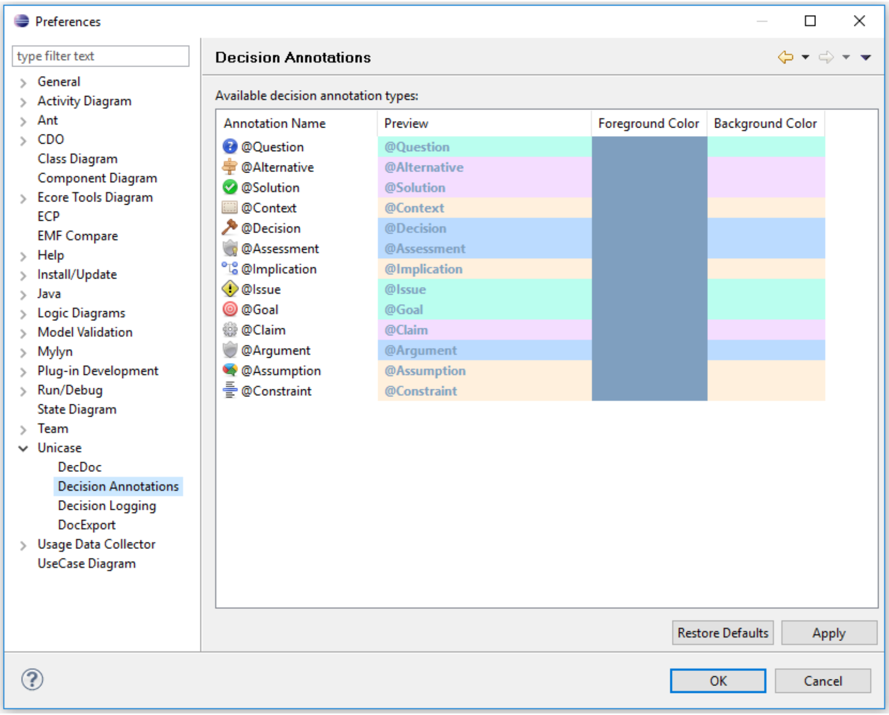


Decision Code Annotations

Annotations for "Adapt ProductOrder connectors" (double-click to open code file):

File	Line Number	Description	Class	Hard Linked?
RunningExample/src/org/cocome/tradingsystem/inventory/data/store/ProductOrder.java	15	Adapt our connectors for processing product orders.	CoreAnnotation	Yes

Figure 6.18: Annotations Overview with Highlighted Link Information



Preferences

Decision Annotations

Available decision annotation types:

Annotation Name	Preview	Foreground Color	Background Color
@Question	@Question	■	■
@Alternative	@Alternative	■	■
@Solution	@Solution	■	■
@Context	@Context	■	■
@Decision	@Decision	■	■
@Assessment	@Assessment	■	■
@Implication	@Implication	■	■
@Issue	@Issue	■	■
@Goal	@Goal	■	■
@Claim	@Claim	■	■
@Argument	@Argument	■	■
@Assumption	@Assumption	■	■
@Constraint	@Constraint	■	■

Figure 6.19: Preferences Dialog for Code Annotations

Java projects containing source code files. For detail, one UNICASE project can be related to one Java project, which is shared in an SVN repository. In this way DecDoc ensures that the versions of knowledge contents in the UNICASE project is synchronized with the revision of the related code files in the Java project. Furthermore, this setting controls in which UNICASE project the instances of *Core Annotation* elements are created to link DKE instances and code annotations. Taking all these features together, DecDoc fulfills requirement C.3, as it integrates code annotations to document decision knowledge in source code files.

Part IV

Evaluation

7 Evaluation of Documentation Model

In this chapter, an empirical study is presented in order to evaluate the feasibility of documenting real-world decision knowledge using the decision documentation model presented in Chapter 5. Therefore, design session transcripts of professional software designers are analyzed for any contained decision knowledge, which is then documented using the documentation model. First, the investigated transcripts and related studies are introduced as *study foundations*. Second, the *research process* with the study setup and the applied data analysis is presented. Third, *results* and findings of the study are described in relation to the requirements of the documentation model. Finally, potential *threats to validity* and their mitigation during the study are discussed. A brief report on the study setup and results was previously published in [Hesse and Paech 2016].

7.1 Study Foundations

In the following paragraphs, the investigated design sessions are introduced as the study subject together with the study goal. Furthermore, related studies are discussed, which either investigate the same data set or are related to this study due to a similar research goal and process.

Study Subject and Goal

The study subject are transcripts originating from two different design sessions, which were originally distributed as material for the international workshop "Studying Professional Software Design" in 2010 [Hoek, Petre, and Baker 2010]. In these sessions, teams of professional software designers were given the design task to create a high-level system design for a traffic simulation system. This system was intended to be used by students to simulate traffic flows influenced by the road layout, traffic lights, and the number of cars. All designer teams received a textual description of this design task and had a time limit of one hour and fifty minutes to create the system. The designers were asked to use a whiteboard for any sketches or notes during their work. Each design session was performed with a team of two professional designers. The entire session was recorded on video, and all discussions were transcribed by the workshop organizers. In this study, the transcripts of two sessions with designers from Adobe (referred to as *AD*) and Amberpoint (referred to as *MB*) are investigated. The transcript of a third design session with designers from Intuit is not investigated, because the session was significantly shorter than the others, and deviated in setup and conditions for the designers.

The task description for the designers included a set of briefly sketched requirements for the traffic simulator. An overview of these requirements is given in Table 7.1. They describe several aspects of the *system* model, such as the need to provide representations of intersections (R-I) with a coordinated system behavior for lights (R-II.a, R-II.b), traffic sensors (R-II.c), or the traffic simulation itself (R-III). In addition, these requirements also cover the *interaction* of users with the system. For instance, the students as users of the system shall be able to control the traffic flow or traffic density (R-IV). Furthermore, non-functional requirements are given with regard to the user interaction and the quality of the design. In detail, the system shall be easy to use (R-V) and motivate the users (R-VI), while the system design is required to be elegant (R-VII) and clear (R-VIII).

No.	**Content of Requirement**
Functional Requirements	
R-I	Enable students to create a visual map with at least six intersections and roads of varying length as simulation area.
R-II	Enable students to describe the behavior and timing of traffic lights; the system shall allow for left-hand green arrow lights.
R-II.a	Combination of traffic lights, which result in crashes, are not allowed.
R-II.b	Every intersection on the map is a four-way intersection and has traffic lights.
R-II.c	Enable students to choose for each intersection to have sensors, which trigger the traffic lights.
R-III	Enable students to simulate traffic flows on the map in real-time; the system shall depict the traffic flows and traffic light states.
R-IV	Enable students to change the density of traffic entering the simulation.
Non-functional Requirements	
R-V	The system shall be easy to use.
R-VI	The system shall motivate the students to explore the simulation.
R-VII	The system design shall be elegant.
R-VIII	The system design shall be clear.

Table 7.1: Summary of Design Task Requirements

To evaluate whether the documentation model is feasible to document real-world decision knowledge in practice, a source of accessible decision knowledge from industry professionals is required. At

Category	Design Aspect	Team
System Concept	MVC	AD
System Concept	User Interface	MB
Road System		
High-level organization	Intersections; Network	AD
Intersections	Signals and sensors in approaches	MB
Intersections	Have roads (with lights and cars)	AD
Roads	Lanes, with signal per lane	AD
Roads	Capacity	AD
Roads	Latency	MB
Connection of roads to intersections	Intersections have queues (roads)	AD
Connection of roads to intersections	Lights and sensors in approaches	MB
Traffic Signals		
Place in hierarchy	Belong to roads	AD
Place in hierarchy	Belong to approaches	MB
Safety: Independent lights with safety checks	Controller checks dynamically	AD
Safety: Independent lights with safety checks	UI checks at definition time	MB
Relations among intersections	Independent	AD
Relations among intersections	Synchronized	MB
Setting timing	System sets timing	AD, MB
Setting timing	Students set timing	MB
Traffic Model		
-	Master traffic object, discrete cars	MB
Discrete cars	Cars with state, route, destination	MB
Discrete cars	Random choices at intersections	AD, MB
Simulator		
Set of objects	executing in parallel threads	MB
Set of objects	traversed by a controller at each clock tick	AD
Model of Time	Uniform time ticks	AD
User Interface: Display		
Layout of visual map	intersections implied by road crossings	AD, MB
Relation of layout distances to road length	layout determines road length	MB
Relation of layout distances to road length	layout determines length, constrained to grid	AD
Define the map	click-drag-drop visual editing	AD, MB
Setting light timing	double-click on intersection	AD, MB
Defining traffic model	set traffic loads only at edges	AD, MB
Viewing results	see individual cars, lights	MB
Viewing results	see view of density on roads	MB
Saving and restoring	supported	MB
Saving and restoring	not supported	AD

Table 7.2: Summary of the Design Space according to [Shaw 2012]

best, these professionals would apply the documentation model directly during decision making and documentation. However, design decisions are often critical in the competitive environment of industrial software development, because they reflect business knowledge and operational strategies of the respective development organization. In consequence, it was difficult to find industrial designers who were free and willing to apply a documentation model for decisions in practice, and share the documentation outcome. In this regard, the design decision transcripts from the UCI design workshop are a valuable source of raw decision knowledge from professionals, which is accessible for further investigation. For this reason, the documentation model was applied on the transcripts, although the underlying decisions were made beforehand. Furthermore, based on the session transcripts, related studies (cf. [Jackson 2010] and [Shaw 2012] in the next paragraph) explored and structured the underlying design space of the design task. A summary of the design space with its different aspects is given in Table 7.2. Findings on the decision knowledge from these previous studies can be used and reflected within the study for the evaluation of the documentation model. For instance, design decisions identified by previous studies can be compared to those decisions, which were uncovered in this study. This helps to improve the quality of results, and to mitigate potential threats to validity.

The overall goal of this study is to investigate the feasibility of the documentation model to structure and document decision knowledge resulting from real-world decisions. Using the Goal Question Metric (GQM) approach [Basili, Caldiera, and Rombach 1994], this goal is described with the structured template of GQM in Table 7.3.

GQM Template	**Goal Description**
Determine	the usage of the documentation model for documenting decisions
with respect to	feasibility
for the purpose of	evaluating the documentation model
in the context of	design decisions driven by given requirements
from the viewpoint of	researchers.

Table 7.3: Description of Study Goal with GQM

Regarding this study goal, the transcripts of the UCI design workshop are well suited to serve as study subject. First, they provide real-world raw data on design decisions from professional software designers. The transcripts represent the original design discussions held by the designers, and the data was not altered or extended by others. Therefore, creating a structured decision documentation with the documentation model using the data provides unbiased insights on the capabilities of the model. Second, the transcripts are well recognized as a study object within the research community on design decisions. Thus, the validity and integrity of the investigated data was checked in many other related studies. Third, the setup of the UCI design workshop and its related studies provide additional information on the requirements, design space, and context of the transcripts, which can be used as additional input to this study. Thereby, it is possible to link the documented decisions with requirements and design artifacts from the design space. However, it should be noted, that in this study no evaluation of the model is performed regarding links to source code, as no source code or other implementation artifacts were created during the design workshop.

Related Studies

Several studies were performed based on the introduced design session transcripts. An overview of these studies is given in special issues of *Design Studies* in 2010 and *IEEE Software* in 2012 as well as in the book of Petre and Hoek [Petre and Hoek 2013] as an extended collection of contributions from the original workshop.

In the special issue of *Design Studies*, Ball, Onarheim, and Christensen have published a study to investigate the solution development of designers in relation to the given requirements. They analyze the design requirements and the mental simulation of solutions based on the thoughts expressed by the designers [Ball, Onarheim, and Christensen 2010]. Similarly, Baker and Hoek focus on the generation of ideas in multiple cycles of discussing different solution-related subjects. In particular, they identify reoccuring solution ideas in these cycles [Baker and Hoek 2010]. In contrast, the study of Tang, Aleti, et al. examines the decision making process of the designers. The authors find that design decisions may either address the problem or solution space of the given design task, so that both spaces co-evolve over time [Tang, Aleti, et al. 2010]. Also, the study of Christiaans and Almendra focuses on investigating the design decision making. The authors describe the design strategies with the related cognitive processes applied by the designers [Christiaans and Almendra 2010]. All these studies apply a research process, which is similar to the study presented in this chapter: the studies analyzed the transcripts by coding relevant text parts according to given coding schemes. However, only the study of Ball, Onarheim, and Christensen explicitly considers relations between requirements and decision knowledge. Here, the requirements for the designers are grouped according to their level of complexity and examined for relations to different design strategies. All other studies focus primarily on the design outcome in relation to the discussions and decision making processes of the designers. None of the studies explicitly addresses the structured extraction and documentation of decision knowledge given in the transcripts. Thus, the study presented in this chapters not only evaluates the feasibility of the documentation model regarding its requirements. It also contributes an investigation of relations between fine-grained decision knowledge, requirements, and design.

In 2010, Jackson has presented a study to analyze the specific structures of the design space addressed by the designers in their sessions. In detail, the study outlines different layers of complexity for the designed traffic simulation system resulting from simulation entities, such as road layouts, signal units, and vehicles, and their interactions [Jackson 2010]. This analysis is later extended and refined by Shaw in the special issue of *IEEE Software* to model and classify the actual design decisions made by the designers within the given design space. Therefore, the author describes different potential designs of the traffic simulation and highlights the designs resulting from the decisions made by the designers [Shaw 2012]. Both studies do not systematically codify the fine-grained decision knowledge documented within the transcripts and, therefore, differ from the research process of the study presented in this chapter. However, both studies provide representations of the design space and its contained design decisions, which are valuable information for the evaluation of the documentation model.

Further contributions published in the special issue of *IEEE Software* focus on decision making processes of the designers. Dilmaghani and Dibble investigate, how the designers use the whiteboard, for instance to draw system sketches or visualize aspects of the design problem and potential solutions. From this analysis, they conclude 10 basic rules to enhance early-stage designs [Dilmaghani and Dibble 2012]. Similarly, Rooksby and Ikeya examine, how designers may remain focused during the early-stage design processes, so that their collaborative work becomes coordinated and productive. They outline the importance of a shared focus, an open attitude towards new ideas, and the identification of agreement and disagreement between designers [Rooksby and Ikeya 2012]. Vliet and Tang present an extension to their study formerly published in *Design Studies* with a detailed analysis of the interplay between the solution-driven and problem-driven approach of solving design problems [Vliet and Tang 2012]. Finally, Nakakoji et al. present an approach to document decision knowledge by recording and linking different visual sources, such as pictures from the whiteboard with a video from the design session [Nakakoji et al. 2012]. Whereas this approach aims to improve the documentation of decisions, the decisions itself remain implicit within the recorded sources.

Further related studies address links between decision knowledge and either requirements or design artifacts. An overview of these related empirical studies with their respective documentation approaches can be found in the scientific state of the art for decision documentation presented in Chapter 4. However, these studies differ from the study presented in this chapter, because they do not investigate decision knowledge with fine-grained knowledge structures.

7.2 Research Process

In the following subsections, the research process of this study is described. An overview is depicted in Figure 7.1. First, a preparation phase was performed to define the research questions for the study. Then, a coding table was created based on these questions to code and document the decision

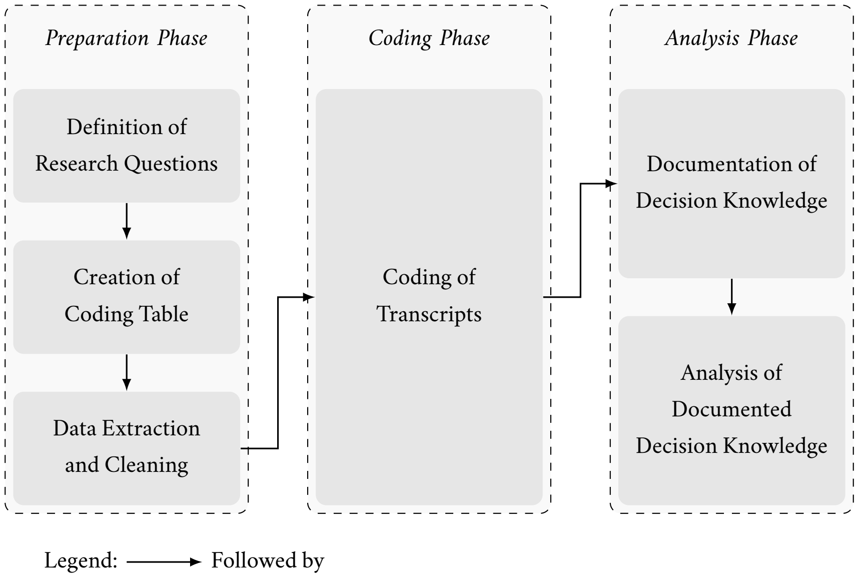

Preparation Phase
Definition of Research Questions
Creation of Coding Table
Data Extraction and Cleaning
Coding Phase
Coding of Transcripts
Analysis Phase
Documentation of Decision Knowledge
Analysis of Documented Decision Knowledge
Legend:
Followed by

Figure 7.1: Overview of the Study Research Process

knowledge within the transcripts. The data was extracted from the transcripts to prepare them for coding. Second, a coding phase was carried out. Coders were trained to apply the coding table on the extracted data, before the complete coding of the transcripts was performed. Third, the coded decision knowledge was documented and examined using the documentation model and DecDoc during the analysis phase. Finally, the research questions were investigated based on this documentation.

7.2.1 Preparation Phase

Definition of Research Questions

Based on the overall goal, different research questions were derived in order to evaluate the feasibility of the documentation model for documenting decision knowledge in practice with regard to the requirements described for the model in Section 5.2. These research questions (abbreviated as RQ) are:

RQ1: Is it feasible to document RDM and NDM decisions in practice using the documentation model?

RQ2: Is it feasible to document decisions in an iterative manner in practice using the documentation model?

RQ3: Is it feasible to capture decision knowledge during development in practice using the documentation model?

RQ4: Is it feasible to document links within decision knowledge in practice using the documentation model?

Each of these research questions maps to one of the four requirements for the documentation model as described in Section 5.2. RQ1 investigates the feasibility of the model to fulfill requirement A, the documentation of RDM and NDM decisions. In particular, this requires an investigation of the actual usage of knowledge elements for capturing knowledge resulting from RDM and NDM in practice. Thus, decision knowledge resulting from RDM and NDM contained within the transcripts needs to be identified, and structured using the model elements. Thereby, the integrated documentation of NDM and RDM knowledge within one decision can also be examined, as it is likely that both decision making strategies will occur mixed and intertwined within the transcripts.

Next, RQ2 investigates requirement B, the iterative decision documentation. Consistent iterative documentation structures within the model can be examined by representing the flow of design discussions and their contained decisions described in the transcripts using the model structures, such as aggregations of model elements. Throughout these discussions, design decisions are enriched with further knowledge and refined over time. Thus, the capabilities of the model to represent these iterations with newly added instances of knowledge elements can be examined. Also, the fine-grained knowledge elements for capturing decision problem and context knowledge will be instantiated using the transcripts. However, no new knowledge elements will be derived based on the abstract elements within documentation model, as this conflicts with the investigation method of coding the transcripts according to defined coding tables.

Third, RQ3 investigates requirement C, which describes capturing mechanisms for decision knowledge during development. It is mainly investigated how decision knowledge can be captured during design, as the investigated studies contain the contents of design sessions. Furthermore, within the study the relations between design decisions and given requirements are examined and discussed, such that knowledge capturing during requirements engineering is partly covered. However, the selection of transcripts as raw data for the current study does not allow for investigating how decision knowledge can be captured during implementation using the documentation model, because implementation artifacts, such as code files or binaries, would be necessary for such an investigation. Such artifacts were not created in the original setup of the UCI design workshop.

Finally, RQ4 investigates requirement D, the support of links within decision knowledge. In detail, relations to reference and refine the content of knowledge elements will be investigated using the different knowledge element instances originating from the coding of the transcripts. As these instances emerge from the flow of design discussions over time, their refinement can be evaluated by applying relations from the documentation model. Furthermore, relations between decisions are investigated. However, links between knowledge elements and development artifacts are examined for RQ3 by relating the given requirements to instances of decision knowledge elements. Thus, they will not be considered within RQ4.

Creation of Coding Table

Leaf model elements and major relations defined by the documentation model were used to derive a coding table, so that instances of the different knowledge elements and their relations could be identified within the transcripts. Codes resulting from model elements are given in Table 7.4. As decision knowledge within the transcripts unfolds over time due to the progress of each design session, all codes aim to identify and document decision knowledge incrementally. Thereby, each of these codes addresses RQ2. A general code for *Context* was added to capture context knowledge, which could not be identified as one of the more detailed sub-type elements of *Context*. In contrast, no general codes were added for *Solution* and *Problem*, as their sub-type elements allow for distinguishing decision knowledge either as resulting from RDM or NDM. In detail, knowledge resulting from RDM is coded as *Alternative* and *Issue*, whereas knowledge resulting from NDM is represented by *Claim* and *Goal*. By indicating decision knowledge originating either from RDM or NDM, these four codes also address RQ1. Regarding the DKE *Argument*, codes were set according to relations to other instances of knowledge elements, as arguments supported, challenged or assessed previous statements within the transcripts. During the coding, supporting arguments were indicated as *pro-Argument*

Code	**Description**	**Related RQ**
Issue	A concrete question explicitly stated within the transcripts; results from RDM	RQ1, RQ2
Goal	A more general and broad goal expressed by the designers in relation to scenario they have in mind; results from NDM	RQ1, RQ2
Alternative	Proposed solution, which can be assessed according to defined criteria; results from RDM	RQ1, RQ2
Claim	Proposed solution, which is based on personal experience and informal knowledge; results from NDM	RQ1, RQ2
Context	Broad and less specific information regarding a decision or decision element	RQ2
Assumption	Information, which is uncertain or approximated by the designers	RQ2
Constraint	Information representing a limitation or restriction	RQ2
Implication	Information containing a consequence	RQ2
pro-Argument	Supporting information regarding a decision or another decision element	RQ2
con-Argument	Information challenging a decision or another decision element	RQ2
Assessment	Information given to assess a decision or another decision element	RQ2

Table 7.4: Codes for Identifying Instances of Documentation Model Elements

along with the `supports`-relation, and attacking arguments were coded as *con-Argument* with an `attacks`-relation. In addition, *assessment* codes were used together with `concerns`-relations to each assessed knowledge element instance. Instances of decisions were not explicitly coded, but created for each design aspect within the design space description given in Table 7.2.

Furthermore, several codes were used to highlight relations between decisions, and relations between decision elements. These codes are described in Table 7.5. As RQ4 is concerned with the capability of the model to create links between model elements, all codes address this research question. First, instances of DKEs are typically contained within a decision instance or another DKE instance, which is expressed by the code *containedIn*. Also, DKE instances may be related to other instances, which is indicated by *concerns*. In particular, this code is used to indicate relations of DKE instances to requirements addressed by the designers in their decisions. Thereby, RQ3 is likewise addressed. Next, arguments may *support* or *attack* other instances. Furthermore, several relations between decision instances were investigated for each transcript. To this end, codes were used based on the model relations for decisions: *relatedTo* for general relations, *dependsOn* to express the dependency of one decision on the solution of another, *boundTo* to indicate that a decision requires the outcome of another decision to be implemented, and *conflictsWith* to express that two decisions cannot be realized together.

Code	**Description**	**Related RQ**
I.containedIn(D|DKE)	The current instance I is part of the decision instance D or the decision knowledge element instance DKE	RQ1, RQ2, RQ4
I.concerns(D|DKE|R)	I is related to D or DKE or requirement R, for instance if designers express uncertainty about the given requirements or describe any extensions to them	RQ3, RQ4
I.supports(D|DKE)	I is an argument in favor of D or DKE	RQ4
I.attacks(D|DKE)	I is an argument challenging D or DKE	RQ4
D1.relatedTo(D2)	Decision instance D1 is generally related to decision instance D2	RQ4
D1.dependsOn(D2)	D1 depends on the successful identification of a solution for D2	RQ4
D1.boundTo(D2)	D1 is bound to the implementation of D2	RQ4
D1.conflictsWith(D2)	D1 cannot be implemented together with D2	RQ4

Table 7.5: Codes for Identifying Relations between Documentation Model Element Instances

However, four relations of the documentation model were not coded. The *comments*-relation between *Arguments* and other DKEs was not used, because arguments from the design session transcripts should be clearly identified either as *supporting* or *attacking* arguments. In addition, there was no conclusive data to apply the relation *isAssessedIn*, as this requires *Context* instances to be determined as criteria for decision solutions. This was not possible because the decision making processes were not finished by the designers due to the time restriction of the UCI workshop. Similarly, there was no foundation for identifying the *resolves*-relation for *Solutions* and the *isBoundTo*-relation for *Questions*.

Data Extraction and Cleaning

In order to prepare the data from the transcripts for coding, all discussion statements were extracted from the textual PDF files provided by the workshop organizers into Excel sheets. The discussion statements were ordered according to their time stamp indicating their occurrence within the discussions. Discussion statements within transcripts, which were marked as *inaudible* by the workshop organizers, were marked to be excluded from coding.

7.2.2 Coding Phase

Coding of Transcripts

The author of this thesis coded the extracted data of both transcripts completely. The actual coding procedure is illustrated by an example statement of transcript and its assigned codes in Table 7.6. In order to improve the quality of the coding, the second author of [Hesse and Paech 2016] also coded the first 10% of both transcripts. Then, both codings were compared, all deviations were discussed, and the coding table as well as the criteria for setting a code were further refined. For each DKE code assigned to a statement within the transcripts, a sequential numerical ID was assigned in order to clearly identify all coded knowledge element instances and their relations. Because the numerical values of the ID were assigned sequentially, they constitute an ordinal ranking for time, i.e., the later a statement was made, the higher is the respective ID.

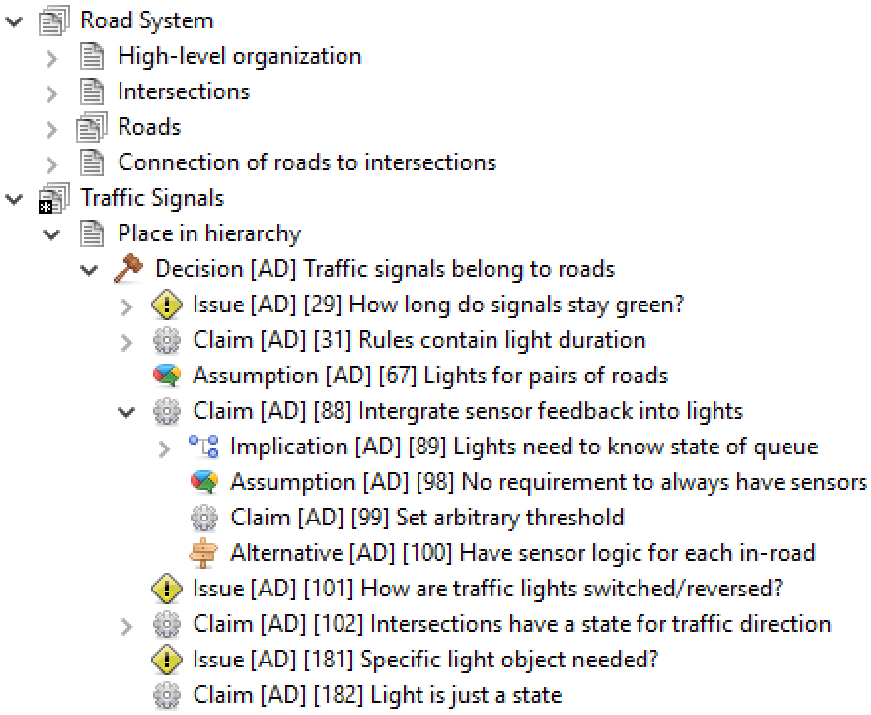
Adobe session transcript at [54min:33.6s]	"[...] if we want to get more sophisticated we could have sensor logic taking sensor for each in-road so that if say these two are empty then these two just stay green forever? Until these sensors go down."
Codes for RQ1	Alternative, ID: 100
Codes for RQ2, RQ4	100.containedIn(88), 100.concerns(73)
Codes for RQ3	100.concerns(R-II.c)

Table 7.6: Example for Codes Applied on Transcript Statement

7.2.3 Analysis Phase

Documentation of Decision Knowledge

Based on the coded data from the transcripts, all decision knowledge elements were documented using the documentation model and DecDoc accordingly. An example of this documentation process with DecDoc is shown in Figure 7.2. In detail, the requirements given in Table 7.1 were created as general `KnowledgeElements`. Furthermore, the design space described in Table 7.2 was created in DecDoc as a high-level structure with 20 decisions belonging to the Adobe transcript and 21 belonging to the Amberpoint transcript. *Categories* were represented as a tree of basic `Sections` in DecDoc to group the decision knowledge, whereas *Design Aspects* were modeled as `Decision` elements. Within these decisions, all coded decision knowledge was created as instance of the appropriate sub-type of `DecisionComponent`, such that all identified decision knowledge elements were either contained in such a decision or in another decision knowledge element. These instances were subsequently linked according to the coded data. It should be noted that the coded text was set as content of the `description` attribute for all identified DKE instances, and the ID was set as `name`. Finally, relations between `Decisions` were set based on their contained knowledge elements.



- Road System
  - High-level organization
  - Intersections
  - Roads
  - Connection of roads to intersections
- Traffic Signals
  - Place in hierarchy
    - Decision [AD] Traffic signals belong to roads
      - Issue [AD] [29] How long do signals stay green?
      - Claim [AD] [31] Rules contain light duration
      - Assumption [AD] [67] Lights for pairs of roads
      - Claim [AD] [88] Intergrate sensor feedback into lights
        - Implication [AD] [89] Lights need to know state of queue
        - Assumption [AD] [98] No requirement to always have sensors
        - Claim [AD] [99] Set arbitrary threshold
        - Alternative [AD] [100] Have sensor logic for each in-road
      - Issue [AD] [101] How are traffic lights switched/reversed?
      - Claim [AD] [102] Intersections have a state for traffic direction
      - Issue [AD] [181] Specific light object needed?
      - Claim [AD] [182] Light is just a state

Figure 7.2: Example for Documented Decision Knowledge from Adobe Transcript in DecDoc

Analysis of Documented Decision Knowledge

In total, 182 instances of decision knowledge elements were identified within the Adobe transcript, and 198 instances were found in the Amberpoint transcript. The accurate numbers per DKE are presented in Table 7.7. It should be noted that the numbers for *Issues*, *Claims*, and *Implications* were higher within the Adobe transcript, whereas more *Arguments* were found within the Amberpoint transcript. Also, only one *Assessment* of different alternatives was uncovered within the Adobe transcript. For the Adobe session, one additional decision was identified during coding in comparison to the decisions identified in [Shaw 2012]. However, for three decisions originating from the design space no DKE instances could be found. For Amberpoint, no additional decisions were found, but one decision from the design space did not contain any DKE instances.

Transcript	Issue	Goal	Alt.	Claim	Cont.	Asp.	Cstr.	Impl.	Arg+	Arg-	Assm.	∑
Adobe	36	3	21	39	17	11	8	33	10	3	1	182
Amberpoint	35	2	22	30	28	18	14	23	15	11	0	198

Table 7.7: Total Number of Documentation Model Element Codes

Regarding the documentation model relations, 332 relations were identified within the Adobe transcript and 359 within the Amberpoint transcript. The detailed numbers per relation are shown in Table 7.8. As expected, the number for *containedIn* matches the total number of DKE instances for both transcripts, because every DKE instance was either part of a decision or of another DKE instance. Regarding the *concerns*-relation, 55 of these relations pointed to requirements for the Adobe transcript, and 65 for the Amberpoint transcript. Interestingly, all *Arguments* within the Adobe transcript had only one *supports*- or *attacks*-relation to another instance. In contrast, two opposing arguments were found within the Amberpoint transcript, which had two *attacks*-relations to different instances. Relations between decision instances could only be identified for those decisions containing DKE instances. Several *boundTo*-relations were identified to the top-level decisions from the design space for both transcripts. Whereas most decisions depended on the outcome of another decision mostly concerned with the same *Design Aspect* from the design space for the Adobe team, mainly general relations between decisions via *relatedTo* were found within the Amberpoint transcript. Finally, one pair of conflicting decisions was identified within the Amberpoint decisions.

Code	Adobe Transcript	Amberpoint Transcript
containedIn	182	198
concerns	113	109
supports	10	15
attacks	3	13
relatedTo	6	13
dependsOn	10	8
boundTo	8	2
conflictsWith	0	1
∑	332	359

Table 7.8: Total Number of Documentation Model Relation Codes

As highlighted by Shaw, both teams of designers followed different approaches to solve the given design problem [Shaw 2012]. The Adobe team was dedicated to the functionality of the system, and, therefore, mostly considered the technical architecture according to the Model-View-Controller-approach. In contrast, the Amberpoint designers focused on the user interface, as well as the interaction design between the system and the students. These differences are illustrated by Table 7.9, which depicts the decisions with most DKE instances for each team. Together with different personal interests and experiences of the involved designers this may explain why the total amounts of instances per DKE differ between the teams, although both teams received the same problem description and worked under equal conditions.

Adobe Decisions	**#DKE**	**Amberpoint Decisions**	**#DKE**
Set of objects – traversed by a controller at each clock tick	25	Discrete cars – Cars with state, route, destination	26
Intersections – Have roads (with lights and cars)	22	Intersections – Signals and sensors in approaches	23
High-level organization – Network	17	Connection of roads to intersections – Lights and sensors in approaches	20
Place in hierarchy – Traffic signals belong to roads	16	Traffic Model – Master traffic object, discrete cars	20
Layout of visual map – Intersections implied by road crossings	12	System Concept – User Interface	16

Table 7.9: Decisions with Highest Amounts of DKE Instances [Hesse and Paech 2016]

7.3 Results and Discussion

7.3.1 Results for RQ1: Feasibility of Documenting RDM and NDM Decisions using the Model

In order to answer the first research question, the codes for *Issue*, *Goal*, *Alternative*, and *Claim* need to be examined, as they are related to decision knowledge originating either from RDM or NDM. With regard to the amounts of DKE instances identified within transcripts, knowledge originating from both strategies has been identified and documented, as summarized in Table 7.10. More than half of all identified DKE instances for the Adobe transcript and nearly half of all identified DKE instances for the Amberpoint transcript are related to RDM or NDM, which can be considered significant numbers in relation to the overall numbers of DKE instances for the two transcripts. However, it should be noted that the number of two identified *Goals* within each transcript is small in comparison to 36 and 35 *Issues*, which were identified respectively. In contrast, the difference in numbers for identified *Alternatives* and *Claims* is less pronounced.

Transcript	Issue	Goal	Alternatives	Claim	Percentage of Total Amount
Adobe	36	3	21	39	54.4%
Amberpoint	35	2	22	30	44.9%

Table 7.10: Amounts of RDM- and NDM-related Codes

For the Adobe transcript, two decisions were found containing no RDM elements, whereas the Amberpoint transcript contained 4 decisions without RDM elements, and one decision without NDM elements. Also, for one decision within the Amberpoint transcript, neither RDM- nor NDM-related elements were identified. However, these decisions were rather small, as they had a total size between one and eight DKE instances. Examination of the structure of the remaining decisions revealed the mixed occurrence of RDM and NDM knowledge within the same decision. A typical example from the Adobe transcript is given in Figure 7.3. Here, the *Claim* with ID 56 emerged as a solution proposal to an *Issue*, after the designers had already discussed a couple of *Alternatives*. A second example from the Amberpoint transcript is depicted in Figure 7.4. It shows a decision instance containing a mix of *Claims*, *Issues*, and one *Alternative*. Whereas in the first example the last identified DKE instance was an NDM-related element, it is an RDM-related *Alternative* in the second example.

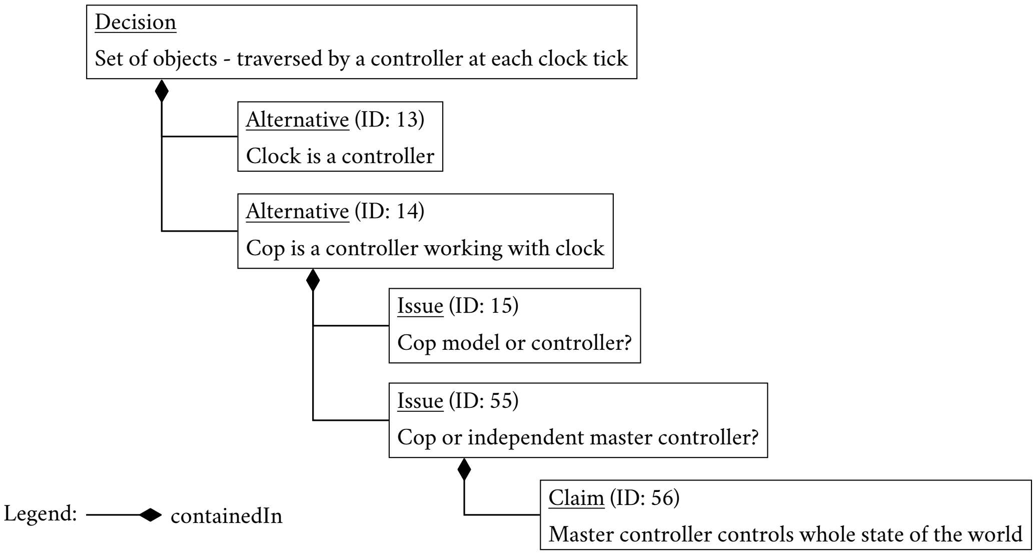

Decision
Set of objects - traversed by a controller at each clock tick
Alternative (ID: 13)
Clock is a controller
Alternative (ID: 14)
Cop is a controller working with clock
Issue (ID: 15)
Cop model or controller?
Issue (ID: 55)
Cop or independent master controller?
Claim (ID: 56)
Master controller controls whole state of the world
Legend:
containedIn

Figure 7.3: Adobe Decision Excerpt with Mix of RDM and NDM Element Instances

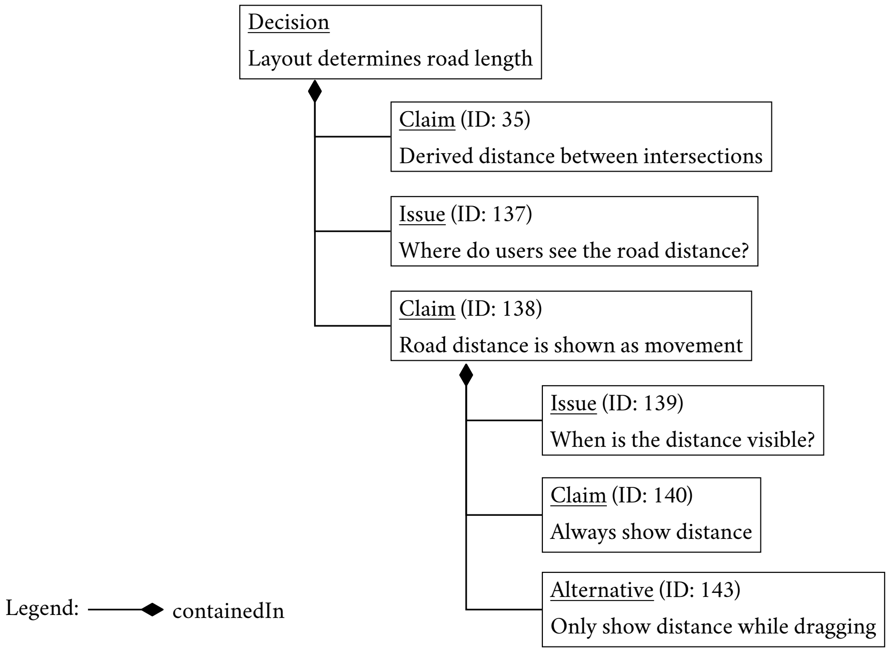

Decision
Layout determines road length
Claim (ID: 35)
Derived distance between intersections
Issue (ID: 137)
Where do users see the road distance?
Claim (ID: 138)
Road distance is shown as movement
Issue (ID: 139)
When is the distance visible?
Claim (ID: 140)
Always show distance
Alternative (ID: 143)
Only show distance while dragging
Legend:
containedIn

Figure 7.4: Amberpoint Decision Excerpt with Mix of RDM and NDM Element Instances

7.3.2 Discussion of Results for RQ1

The results for RQ1 show that it is feasible to capture and document decision knowledge originating from RDM and NDM using the documentation model. Herein, the decision knowledge may be based purely on RDM, purely on NDM, or on a mixture of both strategies. However, it should be noted that the codes for documenting RDM- and NDM-related decision knowledge used in this study are only a first approach to capture and structure this knowledge. Of course, many further and more refined codes could be derived based on the two decision making strategies. One promising candidate could be *Assessment*, as it is related to the process of determining a solution using RDM. Interestingly, only one *Assessment* was uncovered in the study. On the one hand, this could be related to personal preferences of the specific designers who have participated in the investigated design sessions. In this case, results for this DKE are assumed to vary if decision knowledge from further design sessions would be coded and documented. On the other hand, further assessments could have been contained in the investigated transcripts, which were not captured by the code for *Assessment* and its related DKE. In this case, a further refinement of the *Assessment* code with regard to properties of solutions may be helpful. Also, the low number of identified *Goals* is noteworthy. This, however, may have been caused by the requirements given to the designers along with the problem description. Thereby, the designers might not have felt the need to search and discuss their own goals, but referred to the given ones with more concrete *Issues*.

Furthermore, it is interesting that the designers in both teams did mix RDM and NDM in various decisions. This is in line with the findings of Zannier, Chiasson, and Maurer who describe the intertwined usage of both strategies in practice [Zannier, Chiasson, and Maurer 2007]. However, it is not yet fully understood whether this mixed strategy use has any positive or negative impacts on the quality of decision outcomes. In the presented study, no validation or quality assessment of the design decisions regarding the design task and its requirements was performed. In the related studies, the quality of the overall design process was examined (cf.[Tang, Aleti, et al. 2010] and [Shaw 2012]), but no detailed reviews were executed to determine the quality of the identified design decisions. Thus, it cannot be assessed if the mixed strategy use did impact the quality of the decisions made by the designers. In particular, this implies that it is also not possible to draw conclusions if a mixed strategy use should be documented this way. Nevertheless, the presented documentation model is capable of documenting decision knowledge according to different structures that emerge over time during design discussions. This would be even more beneficial in case that the mixed strategy use is found to have a positive impact on decision outcome in future studies. In the opposite case, support for incrementally adding new DKE instances freely over time should likely be limited or extended by rule sets and transformation support, which enforce and help to automate the documentation according to the preferred strategy.

7.3.3 Results for RQ2: Feasibility of Documenting Decisions Iteratively using the Model

The iterative documentation of decision knowledge can be assessed by examining the structures for decision instances, which emerged while documenting the coded knowledge from the transcripts. For the Adobe transcript, 93 DKE instances were added directly to decisions, and 89 DKE instances were contained in other DKE instances. Similar numbers were found for the Amberpoint transcript, for which 97 DKE instances were added to decisions directly, and 101 DKE instances were part of other DKE instances. Based on this distribution, two major patterns for refining existing documented knowledge were observed. First, a given decision instance was enriched with additional knowledge to clarify the current solution proposal or to add further context. DKE instances were then added directly to the respective decision. Second, given DKE instance were refined during the flow of the design discussions. This resulted in multiple deeper levels created using the *containedIn*-relation. Examples for these two patterns are given based on the Amberpoint transcript in Figure 7.5. Box (a) shows a decision refinement, where the *Assumption* with ID 27 was added in a further iteration to refine a formerly emerged *Constraint* by stating that the timing of traffic lights would impact the traffic speed simulated by the system. In contrast, box (b) depicts the specific refinement of an *Alternative*, so that an *Implication* and an *Assumption* are added as part of this existing alternative. This was the result of a discussion loop, which highlighted specific consequences and uncertainties regarding cyclic default timings for lights at intersections. As a result, it is feasible to cover both the enrichment of given decisions and the extension of particular knowledge elements within a decision using the decision documentation model.

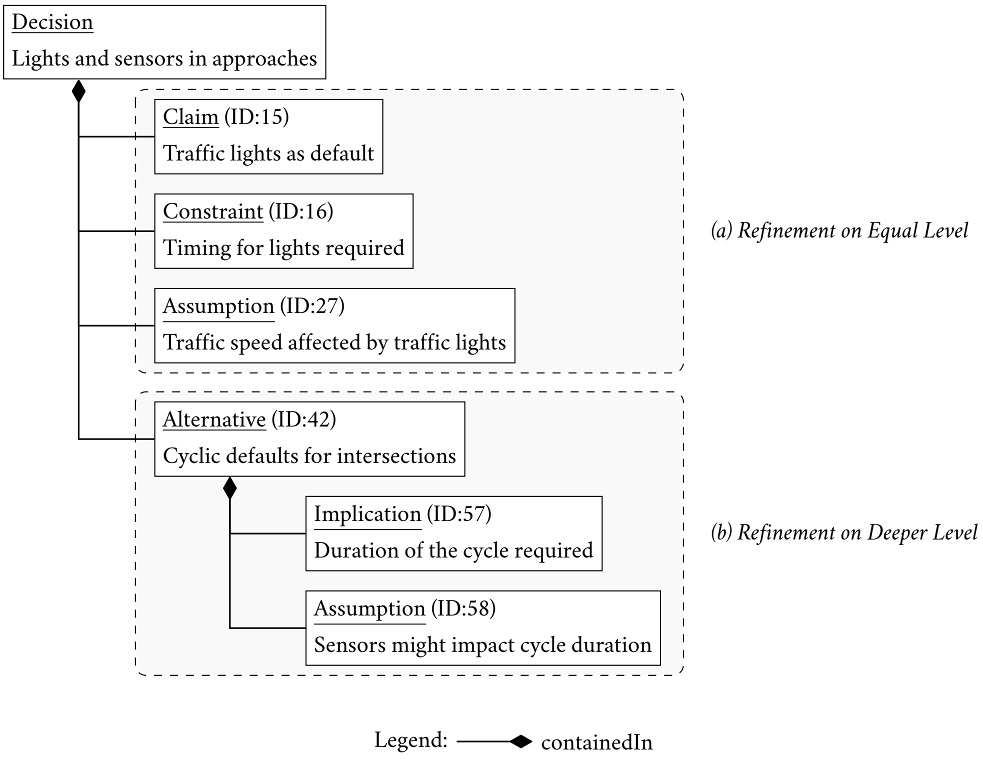

Decision
Lights and sensors in approaches
Claim (ID:15)
Traffic lights as default
Constraint (ID:16)
Timing for lights required
Assumption (ID:27)
Traffic speed affected by traffic lights
(a) Refinement on Equal Level
Alternative (ID:42)
Cyclic defaults for intersections
Implication (ID:57)
Duration of the cycle required
Assumption (ID:58)
Sensors might impact cycle duration
(b) Refinement on Deeper Level
Legend:
containedIn

Figure 7.5: Amberpoint Decision Excerpt with Iterative Refinements by Adding DKE Instances

In addition, it was investigated how the documentation model supports the refinement of documented knowledge by providing fine-grained knowledge elements. The usage of *Issues* and *Goals* for knowledge on the problem space of decision was presented in Section 7.3.1. In this section, the usage of *Context* and its sub-elements *Assumption*, *Constraint*, and *Implication* is assessed. An overview of total numbers for these elements uncovered within the transcripts is given in Table 7.11. In general, slightly more context knowledge emerged during the Amberpoint design session than during the Adobe session. The distribution of DKE instances according to the type of context knowledge differs

Transcript	Context	Assumption	Constraint	Implication	Percentage of All Codes
Adobe	17	11	8	33	37.9%
Amberpoint	28	18	14	23	43.2%

Table 7.11: Numbers of Context Codes

between both teams. Whereas the Adobe mostly stated *Implications*, the Amberpoint team focused on more general *Context* elements. Also, the Amberpoint designers made more explicit *Assumptions* and *Constraints* than the Adobe team. This shows that the different fine-grained knowledge elements could actually be applied for documenting the knowledge from the transcripts.

An example for applying these context elements during documentation is given in Figure 7.6. Here, the Amberpoint designers stated different context aspects for the topic of cycle times within the system during one iteration, as shown by the consecutive IDs. Interestingly, they started by stating general thoughts regarding the timing and potential states of the traffic lights. Then, this knowledge was enriched by more specific context knowledge, such as an *Assumption* on the actual time format for the timing and an *Implication* concerning the symmetry of timing for the opposite directions of an intersection.

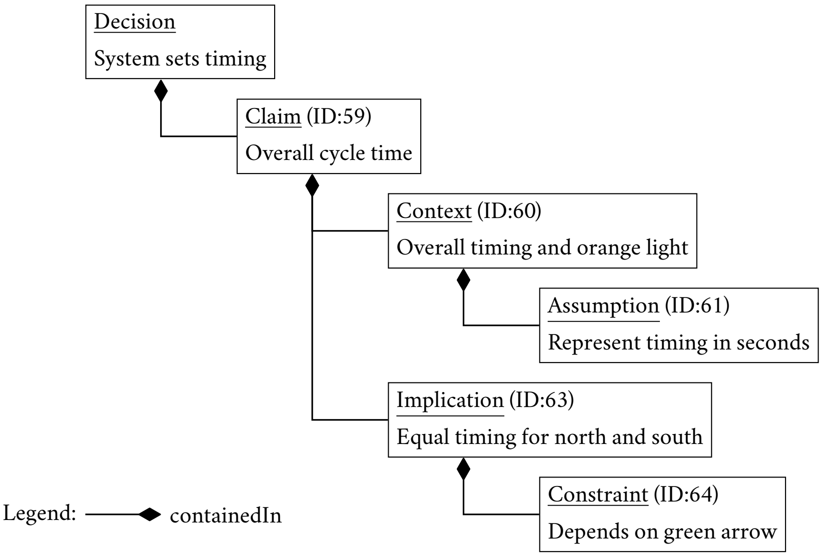


Figure 7.6: Amberpoint Decision Excerpt with Different *Context* Instances

7.3.4 Discussion of Results for RQ2

The results for RQ2 show how the documentation model supports different ways of documenting refined decision knowledge in an iterative manner. Furthermore, the results show that the fine-grained context elements are not yet over-specified, as all sub-types were applied frequently within both transcripts. Nevertheless, it should be noted that knowledge coding and documentation was performed based on written transcripts of finished design sessions, so that all knowledge was accessible during documentation. Thus, DKE instances could be adequately assigned to other instances using the *concerns*-relation, as their place in hierarchy and content could be looked up and evaluated without time pressure. However, it remains an open question whether this iterative refinement process can be performed with similar accuracy during running design sessions without a transcript or for design sessions being held in multiple meetings over time. On the one hand, the refinement abilities provided by the documentation model could be useful to guide these design sessions by making decision knowledge structures explicit and, thereby, highlighting open questions, missing information, and conflicting discussion contributions. On the other hand, this introduces additional challenges for the refinement process and the resulting documentation, as shifts in the interpretation of documentation and inconsistencies between created DKE instances need to be addressed.

Interestingly, no DKE instances could be assigned to three Adobe design decisions and one Amberpoint design decision, although the related design aspects have been explicitly assigned to the teams in the design space described by *Shaw2012*. The complete coding of the session transcripts and the iterative documentation support of the documentation model make it less likely that existing DKE instances actually contained in these decisions have been missed. In consequence, the documentation provided by this study can complement the high-level view provided by the design space with detailed structures of the underlying decisions.

7.3.5 Results for RQ3: Feasibility of Capturing Decision Knowledge during Development using the Model

Many references between DKE instances and requirements from the task descriptions have been uncovered and documented using the *concerns*-relation. An detailed distribution of these references for the Adobe transcript is given in Table 7.12. The numbers per DKE highlight that especially

Requirement	**Issue**	**Goal**	**Alt.**	**Claim**	**Cont.**	**Asp.**	**Cstr.**	**Impl.**	**Arg+**	**Arg-**	**Assm.**	$\sum$
R-I	3	1	2	2	2			2				12
R-II	2		3	1		3	1	2	1		1	14
R-II.a							2					2
R-II.b								1	1	1		3
R-II.c			1	2	1	1	1					6
R-III	5	1	1	4	1				1			13
R-IV	1		1	1			1	1				5
R-V												0
$\sum$	11	2	8	10	4	4	5	6	3	1	1	55

Table 7.12: *concerns*-Relations between DKEs and Requirements for the Adobe Transcript

Issues and *Claims* were driven by requirements during the design discussion, as these elements show the highest numbers of references to requirements. In contrast, the Adobe designers were mostly concerned with requirements I, II, and III, which address the map creation, the handling of traffic lights, and the actual traffic simulation. This is in line with the teams' focus on functionality according to its Model-View-Controller-approach.

The aforementioned three requirements were also discussed by the Amberpoint team, as shown in Table 7.13. In addition, the Amberpoint designers also addressed the non-functional requirement V, which concerns the ease of use of the traffic simulation system. This fits the user interface focus of this team. Most references to requirements were found for *Issues, Alternatives,* and Constraints.

Requirement	**Issue**	**Goal**	**Alt.**	**Claim**	**Cont.**	**Asp.**	**Cstr.**	**Impl.**	**Arg+**	**Arg-**	**Assm.**	$\sum$
R-I	2	1	2	2	2	3			2	1		15
R-II	4			1	1	1	4	2		1		14
R-II.a	1						2			1		4
R-II.b												0
R-II.c	2		2			1	1					6
R-III	1		5	2	3	2	1		1			15
R-IV			1	3	1							5
R-V				1			2		2	1		6
$\sum$	10	1	10	9	7	7	10	2	5	4	0	65

Table 7.13: *concerns*-Relations between DKEs and Requirements for the Amberpoint Transcript

The references to requirements also influenced the knowledge structures that emerged during the design discussions. An example is depicted in Figure 7.7. Here, the Amberpoint team addressed multiple requirements within the same decision. Thus, several context elements were uncovered, which are related to requirement II and II.c. According to the content of DKE instances, several solution proposals are discussed in order to satisfy the constraint of timing dependencies between different directions for the traffic lights. This addresses the required behavior and timing control for traffic lights, as described by requirement II, and the control of traffic lights by sensors, as described in requirement II.c. The first *Claim* with ID 73 is concerned with an overlapping time to address different timings for different directions. The second *Claim* with ID 109 proposes to set the speed per road, so that the designers assume a relation between the light sensors and the car speed in the *Assumption* with the ID 148. This assumption also references requirement II.c, as it describes an uncertainty of designers about the actual meaning of the requirement regarding their proposed implementation. One cause for this knowledge structure might be the need to make trade-offs between different solution proposals, as no formal assessments of alternatives were found within this transcript. Then, relations to requirements may help the designers to check whether a claim is in line with all requirements they wanted to address with their decision.

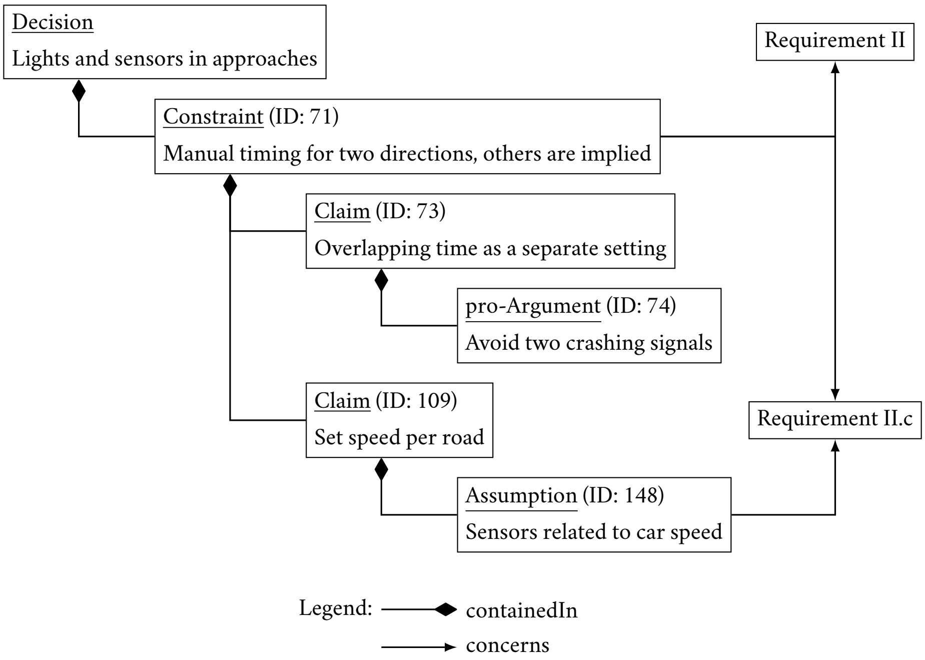

Decision
Lights and sensors in approaches
Requirement II
Constraint (ID: 71)
Manual timing for two directions, others are implied
Claim (ID: 73)
Overlapping time as a separate setting
pro-Argument (ID: 74)
Avoid two crashing signals
Claim (ID: 109)
Set speed per road
Requirement II.c
Assumption (ID: 148)
Sensors related to car speed
Legend:
containedIn
concerns

Figure 7.7: Amberpoint Decision Excerpt with Relations between DKE Instances and Requirements

7.3.6 Discussion of Results for RQ3

Overall, the results for RQ3 show that it is feasible to capture relations between decisions and requirements with the documentation model. In particular, the presented example highlights that decision knowledge and requirements impact each other. On the one hand, both teams derived context knowledge, such as *Constraints* and *Implications*, based on the given requirements. Thereby, there decisions and the emerging knowledge was guided. On the other hand, *Assumptions* made during the decision process pointed out the need for clarifying or even extending the requirements specification. In consequence, the study results outline the mutual impact of design decisions and requirements, which became visible through the application of the decision documentation model. A more detailed investigation of this impact is presented in [Hesse and Paech 2016].

Interestingly, both teams did not address any non-functional requirements in their design discussions, except for requirement V in the Amberpoint session. In addition, the distribution of references to requirements per DKE deviate between the two teams, whereas the overall numbers of references are similar. First, this confirms the well-known fact that designers should consider non-functional requirements in their decisions more explicitly. Second, relations to requirements seem to depend more on the preferences and priorities of the design team, than on the actual content of the given requirements. These two behaviors are likely to decrease the quality of design decisions, and, therefore, should be avoided by designers. This makes it even more important to make relations between decision knowledge and requirements explicit to uncover and mitigate such behavior. Also, a higher number of relations from design decisions to a particular requirement would indicate that this requirement is an important driver for the design of the system. This would be in line with the findings described in [Chen, Ali Babar, and Nuseibeh 2013], and strengthen the comprehensiveness and value of the documented decision knowledge.

7.3.7 Results for RQ4: Feasibility of Linking Decision Knowledge using the Model

Regarding the refinement of documented decision knowledge, it was investigated how different DKE instances as well as different decisions were linked according to the coded knowledge. For the DKE instances, 58 *concerns*-relations pointing to other DKE instances were set for the Adobe transcript, and 44 for the Amberpoint transcript. In addition, several *supports*- und *attacks*-relations have been set for *Arguments*. In the Amberpoint transcript, two *contra-Arguments* were uncovered, which had two *attacks*-relations to other DKE instances. One of these arguments is depicted in

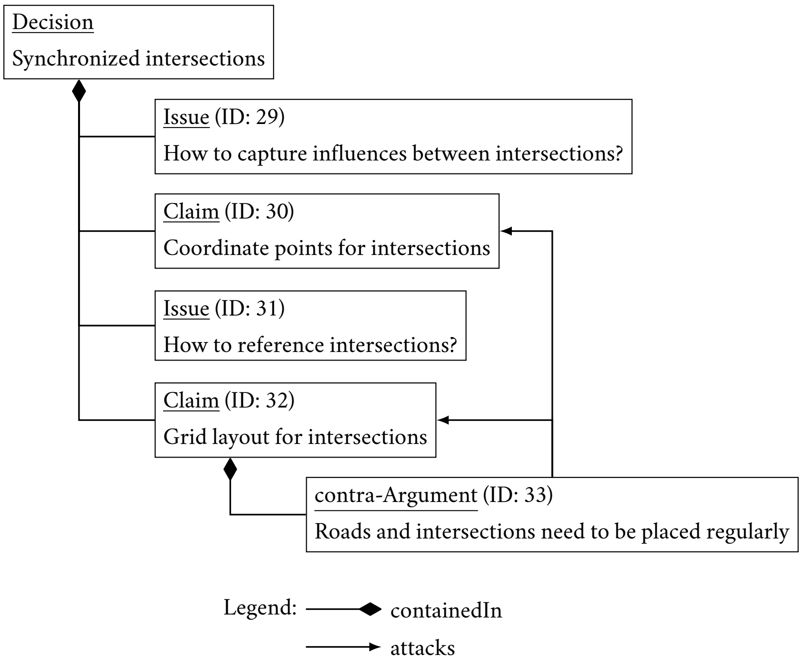

Decision
Synchronized intersections
Issue (ID: 29)
How to capture influences between intersections?
Claim (ID: 30)
Coordinate points for intersections
Issue (ID: 31)
How to reference intersections?
Claim (ID: 32)
Grid layout for intersections
contra-Argument (ID: 33)
Roads and intersections need to be placed regularly
Legend:
containedIn
attacks

Figure 7.8: Amberpoint Decision Excerpt

Figure 7.8. This example highlights how relations helped to refine documented decision knowledge. The *contra-Argument* with ID 33 occurred in the discussion loop concerned with a *Claim* to have a grid layout for intersections, i.e., the argument was added as a part of this claim. However, the argument also addressed coordinates for intersections in a previous claim, so it was linked to both claims using the *attacks*-relation.

Furthermore, a total number of 24 relations between decisions was found for both, the Adobe and the Amberpoint transcript. An example for these relations is depicted in Figure 7.9 based on the Adobe transcript. The graph excerpt shows how the major decision regarding the Model-View-Controller-approach as system concept impacted other decisions. For instance, the decisions for having a network and intersections as entities of the model and visually forming intersections by road crossings, are bound to the implementation of the related model, view, and controller components. In contrast, other decisions only depend on the outcome of previous decisions, but do not require their actual implementation. For instance, the content of the decisions to create roads and discrete cars depends on the decision to have explicit intersections. Similarly, the attachment of a capacity and lanes with signals to roads depends on the decision to have roads. However, these dependent decisions could be changed in case the related decision is omitted or rejected.

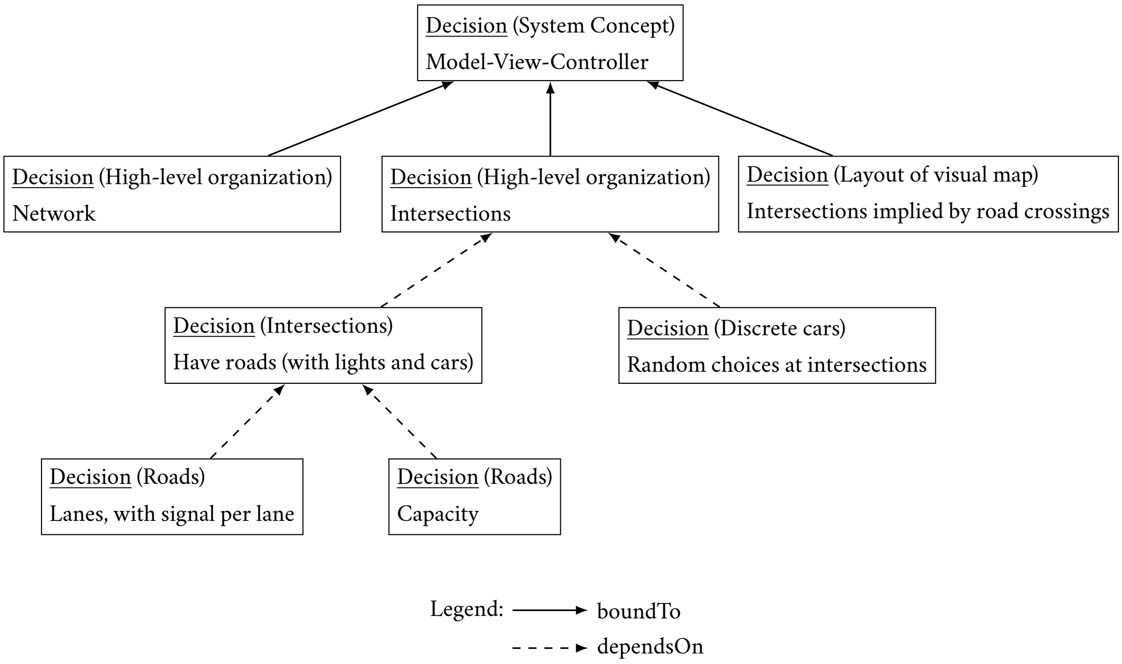

Decision (System Concept)
Model-View-Controller
Decision (High-level organization)
Network
Decision (High-level organization)
Intersections
Decision (Layout of visual map)
Intersections implied by road crossings
Decision (Intersections)
Have roads (with lights and cars)
Decision (Discrete cars)
Random choices at intersections
Decision (Roads)
Lanes, with signal per lane
Decision (Roads)
Capacity
Legend:
boundTo
dependsOn

Figure 7.9: Excerpt from Adobe Relations Graph for Decisions

Interestingly, no conflicting decisions were found in the Adobe transcript, but one pair of conflicting decisions was identified within the Amberpoint transcript. In detail, two decisions on setting the timing within the system are in conflict, as they claim to let both, the system and the students set the timing for the traffic lights. However, this conflict was resolved later on in a third decision on lights and sensors in intersection approaches, when the designers proposed to introduce default timings for the lights, which can be overwritten by the students.

7.3.8 Discussion of Results for RQ4

As the results for RQ4 point out, it is feasible to link both, decision instances and DKE instances, using the decision documentation model. In particular, the *concerns*-, *supports*-, and *attacks*-relations complement the *containedIn*-relation, which has the character of a composition. Thus, a DKE instance can either be part of a decision or of a DKE instance. Then, the other relation types can be used to document further references to DKE instances, as shown for the *Argument* from the Amberpoint transcript.

Nevertheless, it should be noted that these references also cause additional effort for maintaining existing links in case that further iterative refinements extend or adapt the documented knowledge. This is highlighted by the example of the conflicting decisions in the Amberpoint transcript. Here, the content of a third decision impacted the relation between two others. In the particular case, the *conflictsWith*-relation has to be removed and a dependency is created from the third decision to the two others. In consequence, the documentation process could be even better supported by further extending the documentation model, so that relations between decisions can have associated decisions similar to an *Argument* associated to the *resolves*-relation.

Finally, it should be noted that only the most relevant relations from the documentation model have been investigated. Thus, specific relations, such as *comments*, *isBoundTo*, and *isAssessedIn*, were not examined. However, due to the relatively small numbers of *attacks*- and *supports*-relations it appears unlikely that these decisions would have occurred in significant amounts. One reason could be that the designers were rather exploring the design space and discussing design problems and solution than preparing particular decisions for an actual implementation. This reason was caused by the setup for the design sessions with a strict time limit and a focus on design discussions without possibilities to actually implement the designed system.

7.4 Threats to Validity

According to Runeson et al. [Runeson et al. 2012], four different categories of threats to validity have to be considered for the presented study. These categories are described and discussed in the following paragraphs.

Internal Validity
Threats to internal validity concern the correlation between the investigated factors and other factors [Runeson et al. 2012]. The decision knowledge stated by the designers in their discussions might have been influenced by missing further instructions regarding documentation or design reasoning. Thus, the designers might have worked less structured and did not articulate all decision-related thoughts. However, these conditions are realistic and apply for most design discussions in practice. In addition, if the designers would have been required to use specific methods or structured processes for design reasoning or decision documentation, the investigated decision and results of this study would depend on these methods and processes. Next, the quality and guidance of the investigated design sessions might have been impacted by the rather short design prompt. In consequence, the actual decision knowledge expressed by the designers might have been limited by uncertainties about the actual decision problem and context. But again, this corresponds to decision situations in practice. This threat was addressed by checking the acquired decision knowledge with the results of additional studies, such as the studies of Tang, Aleti, et al. [Tang, Aleti, et al. 2010] and Shaw [Shaw 2012].

Construct Validity
Threats to construct validity concern any gaps between intended and actual observations of the researchers [Runeson et al. 2012]. The coding table for decision knowledge might have identified something else than decision knowledge elements (cf. the related threat in Section 3.4). This threat was mitigated by retrieving the codes based on the definitions for each knowledge element, as described in Chapter 5. Also, all codes were tested on the given data and refined in order to capture the elements given in the documentation model in an optimal way. In addition, there is an excellent fit of the presented results with the decisions identified by Shaw [Shaw 2012]. The coding in the presented study covered 18 out of 20 decisions for the Adobe transcript, and 20 out of 21 decisions for the Amberpoint transcript identified in the study of [Shaw 2012]. Finally, the experiences and insights from performing the Firefox study presented in Chapter 3 were used to improve the study setup and coding procedure.

Reliability Validity
Threats to reliability validity concern the degree to which data and analyses of a study are dependent on specific researchers [Runeson et al. 2012]. Only one coder coded all data from both transcripts, which might make the coding less reliable. This threat was addressed by checking and aligning the codes with samples acquired from the second coder. Small parts of the design discussions were inaudible in the videos and, therefore, were marked and left out in the transcripts. In consequence, some relevant decision knowledge might have been not accessible, because it was not contained

in the transcripts. This was addressed by checking the surrounding text in the transcripts for all inaudible parts to uncover hints on any missed content.

External Validity

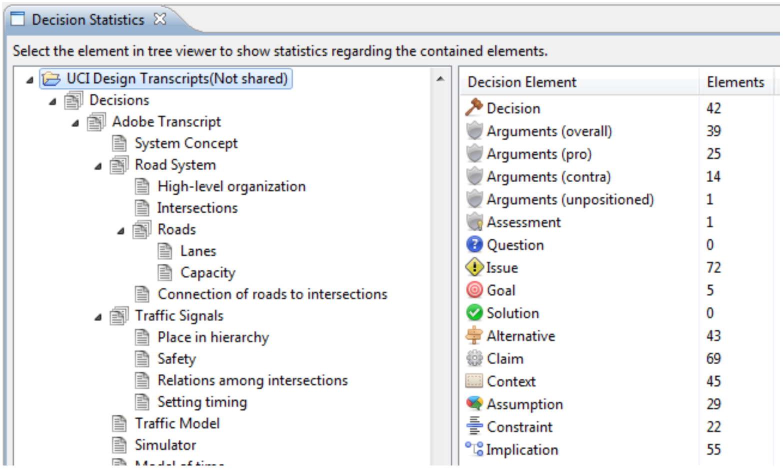
Threats to external validity concern the degree to which the results of our study can be generalized [Runeson et al. 2012]. Only transcripts of two design sessions were investigated in this study. In consequence, the presented findings depend on the designers of two investigated teams, and generalization or comparison to other teams and other design setups may be difficult. The third transcript of the UCI design workshop could have been included in the analysis, but this would have caused more threats to internal validity due to the deviations in this session's setup. Also, the designers performing the investigated sessions were professionals from industry, holding key roles in their respective companies. Thus, the investigated transcripts and the acquired results are likely to represent typical design sessions and the related decision knowledge.

8 Evaluation of Tool Support

In this chapter, the tool support DecDoc is evaluated regarding its feasibility and practicability for documenting decision knowledge. The *feasibility of documenting complex decision knowledge using DecDoc* is illustrated by a demonstration of the tool support with decision knowledge investigated in the previous chapter. These results were also briefly sketched in [Hesse, Kuehlwein, and Roehm 2016]. Because not all requirements for DecDoc can be covered with this demonstration, the *feasibility of documenting decision during implementation using DecDoc* is evaluated by a first case study with students. The students applied DecDoc functionalities on decision knowledge resulting from a task of a practical course on software engineering. This study was formerly published in [Hesse, Kuehlwein, Paech, et al. 2015].

8.1 Feasibility of Documenting Complex Decision Knowledge using DecDoc

To demonstrate the feasibility of documenting decision knowledge using DecDoc for realistic amounts of data exceeding the presented running example, DecDoc was used to document the decision knowledge emerging from the design session transcripts of the UCI design workshop (cf. Section7.2.3). Thereby, decision knowledge was uncovered without the possibility to perform an analysis on additional development artifacts, such as use cases, UML diagrams or source code files. Due to the absence of these artifacts, they could not be linked to any emerging decision knowledge. Thus, only the features of the *Knowledge Editor* of DecDoc can be demonstrated with the transcripts as the main source of decision knowledge. In consequence, this demonstration concerns requirements A.1 to A.3, B.3, and D.2, which mainly address the support for an incremental documentation process of decision knowledge. First, the statistical *Decision Overview* gives an impression of the actual amounts of documented knowledge from the transcripts. This is depicted in Figure 8.1. In total, 380 DKE instances were created for both transcripts. It should be noted that due to the coding and analysis procedure described in Section 7.2.1, no instances of *Question* and *Solution* elements were created, i.e., the number of occurrences equals zero within the statistics. Also, one argument was linked to other instances using both the *attacks*- and *supports*-relation, so that it was counted as *pro-Argument* and as *con-Argument*. This explains why the sum of all distinct argument counts (pro, contra, unpositioned) is one count higher than the overall sum of arguments, which correctly represents the number of all argument instances.



Decision Statistics

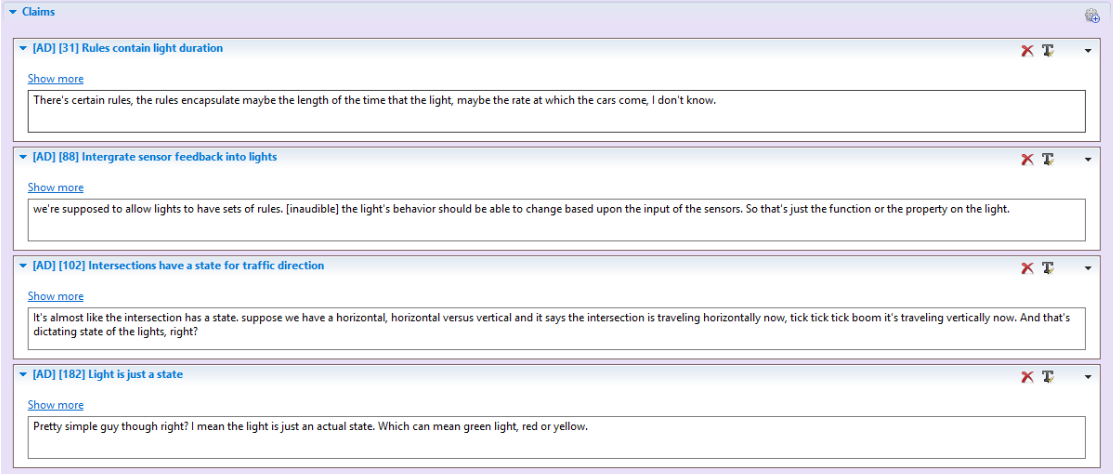
Select the element in tree viewer to show statistics regarding the contained elements.

- UCI Design Transcripts(Not shared)
  - Decisions
    - Adobe Transcript
      - System Concept
      - Road System
        - High-level organization
        - Intersections
        - Roads
          - Lanes
          - Capacity
        - Connection of roads to intersections
      - Traffic Signals
        - Place in hierarchy
        - Safety
        - Relations among intersections
        - Setting timing
      - Traffic Model
      - Simulator

Decision Element	Elements
Decision	42
Arguments (overall)	39
Arguments (pro)	25
Arguments (contra)	14
Arguments (unpositioned)	1
Assessment	1
Question	0
Issue	72
Goal	5
Solution	0
Alternative	43
Claim	69
Context	45
Assumption	29
Constraint	22
Implication	55

Figure 8.1: Statistics Overview Showing all Decision Knowledge from the Transcripts

For documenting decision knowledge regarding one decision, the *Decision Editor* was used. An example from the Adobe transcript comprising the decision that traffic signals belong to the road within their simulation model is depicted in Figure 8.2. The picture shows the list of claims for



Claims

[AD] [31] Rules contain light duration

Show more

There's certain rules, the rules encapsulate maybe the length of the time that the light, maybe the rate at which the cars come, I don't know.

[AD] [88] Intergrate sensor feedback into lights

Show more

we're supposed to allow lights to have sets of rules. [inaudible] the light's behavior should be able to change based upon the input of the sensors. So that's just the function or the property on the light.

[AD] [102] Intersections have a state for traffic direction

Show more

It's almost like the intersection has a state. suppose we have a horizontal, horizontal versus vertical and it says the intersection is traveling horizontally now, tick tick tick boom it's traveling vertically now. And that's dictating state of the lights, right?

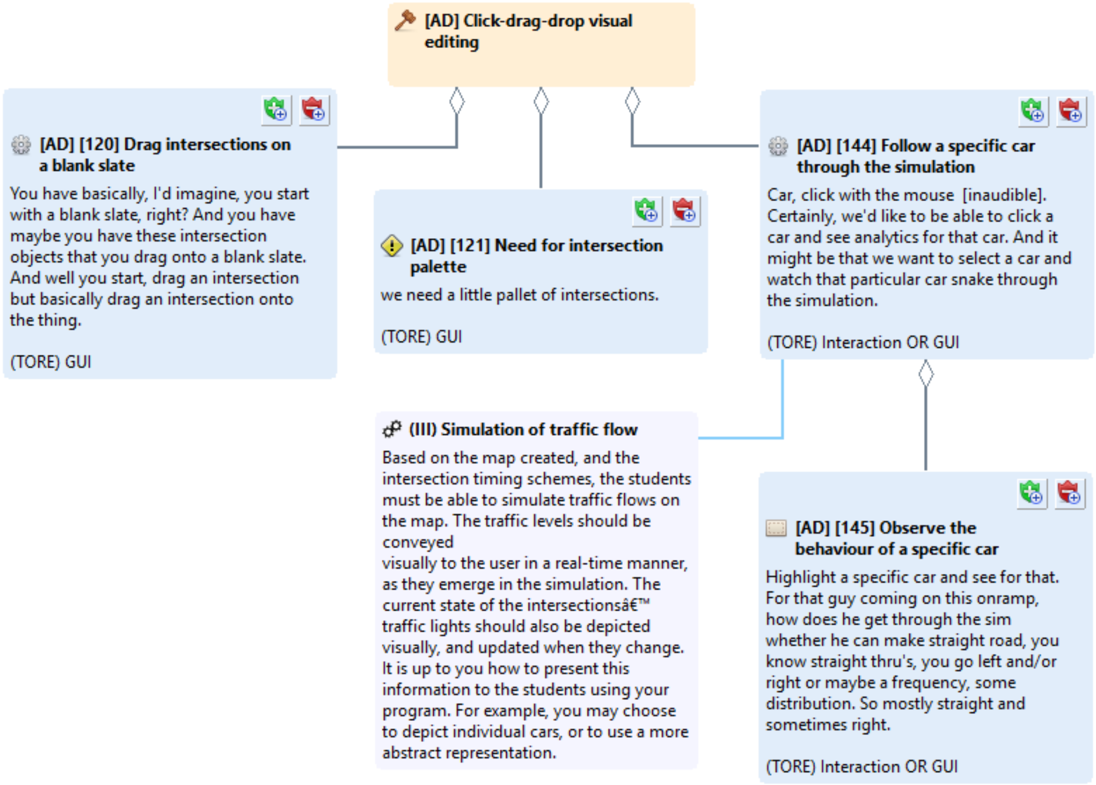
[AD] [182] Light is just a state

Show more

Pretty simple guy though right? I mean the light is just an actual state. Which can mean green light, red or yellow.

Figure 8.2: Decision Editor with Claim Descriptions

this decision with their descriptions. Thereby, the time-efficient documentation of textual contents within DKE instances using the Decision Editor is highlighted. It also outlines that the editor enables developers to swiftly access and edit documented decision knowledge, even if many instances of the same DKE exist within a decision. To explore the relations between DKE instances, the graphical *Decision Overview* is used. An example is given in Figure 8.3 for the Adobe decision concerned with a drag-and-drop visual editing support within the user interface of the traffic simulator. This graph visualization represents DKE instances as nodes and their relations as edges. In consequence, it is required to deal with larger numbers of nodes and edges, and it has to cope with the size of the textual description resulting from realistic decision knowledge. Furthermore, the graph also includes relations between further development artifacts and the actual decision knowledge. In the example, this is highlighted by the relation between the functional requirement III regarding the simulation of the traffic flow and claim 144 on the ability to follow a specific car through the simulation.



[AD] Click-drag-drop visual editing

[AD] [120] Drag intersections on a blank slate

You have basically, I'd imagine, you start with a blank slate, right? And you have maybe you have these intersection objects that you drag onto a blank slate. And well you start, drag an intersection but basically drag an intersection onto the thing.

(TORE) GUI

[AD] [121] Need for intersection palette

we need a little pallet of intersections.

(TORE) GUI

[AD] [144] Follow a specific car through the simulation

Car, click with the mouse [inaudible]. Certainly, we'd like to be able to click a car and see analytics for that car. And it might be that we want to select a car and watch that particular car snake through the simulation.

(TORE) Interaction OR GUI

(III) Simulation of traffic flow

Based on the map created, and the intersection timing schemes, the students must be able to simulate traffic flows on the map. The traffic levels should be conveyed
visually to the user in a real-time manner, as they emerge in the simulation. The current state of the intersectionsâ€™ traffic lights should also be depicted visually, and updated when they change. It is up to you how to present this information to the students using your program. For example, you may choose to depict individual cars, or to use a more abstract representation.

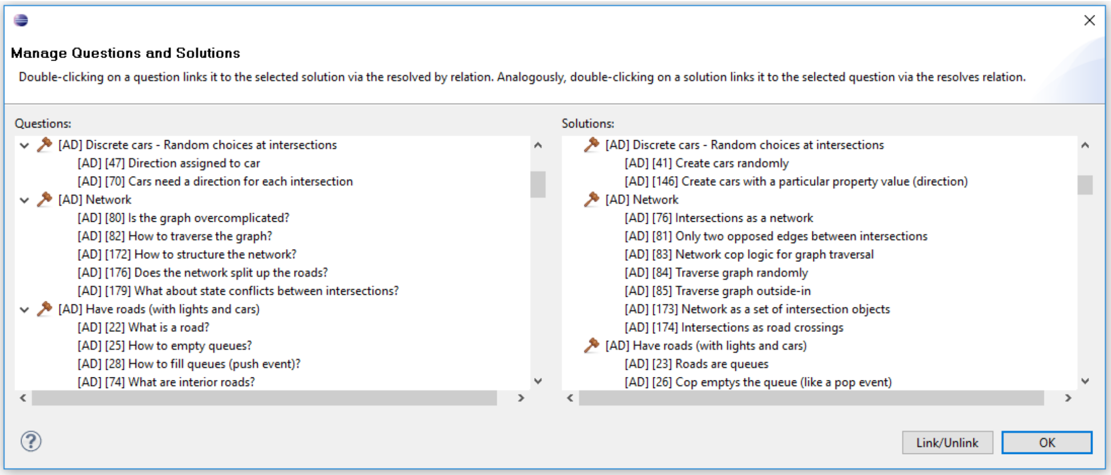
[AD] [145] Observe the behaviour of a specific car

Highlight a specific car and see for that. For that guy coming on this onramp, how does he get through the sim whether he can make straight road, you know straight thru's, you go left and/or right or maybe a frequency, some distribution. So mostly straight and sometimes right.

(TORE) Interaction OR GUI

Figure 8.3: Graphical Decision Overview Showing an Adobe Decision

Finally, questions and solutions within the design session transcripts were explored using the *Solution Management* dialog. An excerpt of the questions and solutions within documented decision knowledge is given in Figure 8.4. In particular, the dialog allows for an easy comparison of the ratio between questions and solutions per decision. For instance, the figure shows that the decision to create a network of roads contains 5 questions, but seven potential solutions. It should be noted that the dialog efficiently reduces the complexity of the emerged decision knowledge structures by presenting all *Question* and *Solution* instances of one decision, regardless of the number of *contains*-relations for the respective instances. Thus, developers are not getting confused by deeper hierarchy trees created by *contains*-relations between instances, which emerge due to an iterative decision-making and the collaborative documentation process in DecDoc.



Manage Questions and Solutions

Double-clicking on a question links it to the selected solution via the resolved by relation. Analogously, double-clicking on a solution links it to the selected question via the resolves relation.

Questions:

- [AD] Discrete cars - Random choices at intersections
  - [AD] [47] Direction assigned to car
  - [AD] [70] Cars need a direction for each intersection
- [AD] Network
  - [AD] [80] Is the graph overcomplicated?
  - [AD] [82] How to traverse the graph?
  - [AD] [172] How to structure the network?
  - [AD] [176] Does the network split up the roads?
  - [AD] [179] What about state conflicts between intersections?
- [AD] Have roads (with lights and cars)
  - [AD] [22] What is a road?
  - [AD] [25] How to empty queues?
  - [AD] [28] How to fill queues (push event)?
  - [AD] [74] What are interior roads?

Solutions:

- [AD] Discrete cars - Random choices at intersections
  - [AD] [41] Create cars randomly
  - [AD] [146] Create cars with a particular property value (direction)
- [AD] Network
  - [AD] [76] Intersections as a network
  - [AD] [81] Only two opposed edges between intersections
  - [AD] [83] Network cop logic for graph traversal
  - [AD] [84] Traverse graph randomly
  - [AD] [85] Traverse graph outside-in
  - [AD] [173] Network as a set of intersection objects
  - [AD] [174] Intersections as road crossings
- [AD] Have roads (with lights and cars)
  - [AD] [23] Roads are queues
  - [AD] [26] Cop emptys the queue (like a pop event)

Figure 8.4: Solution Management for Decision Knowledge from Adobe Transcript

In summary, the usage of DecDoc for documenting the decision knowledge from the UCI design session transcripts showed the feasibility of documenting and structuring decision knowledge using the *Knowledge Editor*. Thereby, the suitability of the Knowledge Editor with its *Decision Editor* to fulfill the requirements A.1 to A.3 is demonstrated. Furthermore, using the *Decision Overview*, it was feasible to visualize relations for decisions and DKE instances in general (requirement D.1), and the *containedIn*-relation in particular (requirement A.2). The *Solution Management* helped to cope with complex knowledge structures, so that DecDoc covers requirement B.3.

However, some requirements could not be examined using the data from the UCI design workshop. Most importantly, the requirements C.1 and C.2 could not be evaluated, as the material of the UCI workshop does not provide information on either security-related requirements or design diagrams. Thus, the import of decision knowledge using the HeRA plugin and the documentation of decision in UML diagrams could not be applied. Furthermore, this also impacts the evaluation of requirement D.2, which describes the necessity to provide the semi-automatic creation of links between decision knowledge and other artifacts. As these artifacts were not available and the HeRA importer wizard was not applied, the related features for proposing links within the wizard were not tested. Nevertheless, requirement C.3 regarding the usage of code annotations will be evaluated in the case study presented in the next section. In this case study, also requirement B.1 regarding the documentation support for decision knowledge for collaborative work involving multiple developers, and requirement B.2 regarding support for extensions of the documentation model within DecDoc will be examined.

8.2 Feasibility of Documenting Implementation Decisions using DecDoc

The requirements B.1, B.2, and C.3 for DecDoc were investigated regarding the feasibility of documenting knowledge for implementation decisions, because these requirements mainly concern the collaborative usage of DecDoc by multiple developers and its usage during development activities. Therefore, a preliminary case study was performed with students from Heidelberg University.

8.2.1 Study Foundations

The overall goal of this study was to evaluate the feasibility of documenting decision knowledge during implementation using the tool support DecDoc with a focus on code annotations. Therefore, DecDoc was presented to and applied by undergraduate students in computer science within a practical course. Seven students participating in the course were grouped into two development teams. Team members were assigned with respect to their skill levels and prior programming experience to achieve comparable and equally skilled teams. Both teams worked on a software project, which required them to plan, implement and document an Eclipse plugin. The teams received an identical description of the project with an initial set of requirements provided in the form of scenarios, which described the intended usage of the plugin to be developed. Both teams performed three sprints with a duration of one week from mid February until the beginning of March 2015. At the end of each sprint, both teams presented their current development progress.

The setting of this practical course was chosen for the study, as it provides suitable conditions to evaluate the aforementioned requirements. First, teams with multiple developers were studied, so that the collaborative focus of requirements B.1 and C.3 could be investigated. Second, the students created code during the course, which was another precondition to be able to examine requirement C.3. To prepare the study, a tutorial for DecDoc was held for all students, so that they had a similar level of competency regarding the usage of DecDoc. Also, the teams received a textual manual on how to use the code annotations. Whereas the usage of UNICASE for documenting the related project knowledge was mandatory for the students, they were not required to use DecDoc in order to pass the course. Thus, annotation usage during implementation could be observed realistically.

8.2.2 Research Process

The overall goal of the study is covered by the research question: Is it feasible to document implementation decisions using DecDoc with its code annotations? To answer this question, the Technology

Acceptance Model (TAM) [Davis, Bagozzi, and Warshaw 1989] was applied to create questionnaires for the students in order to explore the actual use of the DecDoc annotations. In detail, TAM consists of three variables. First, the *ease of use* describes the degree to which a person expects the approach to be effortless. Second, *usefulness* covers the subjective probability for a person to increase its job or work performance. Third, the *intention to use* describes the willingness of a person to use the approach in the future. These variables were assessed by the following statements designed to cover a variable (numbered from S1 to S4):

S1: It was easy to create decision elements with code annotations. *(Ease of use)*

S2: It was easy to locate decision elements within the Eclipse Code Editor. *(Ease of use)*

S3: Code annotations have been useful for the documentation of decisions. *(Usefulness)*

S4: In the future I would use code annotations again to document decisions. *(Intention to use)*

Statements were used instead of open questions, so that the comparability of the responses could be ensured. Thereby, the creation and usage of code annotations within DecDoc was distinguished for further investigation by introducing statement S1 and S2, which both address the ease of use. In contrast, S3 and S4 cover the entire code annotations for usefulness and the intention of use. The answers for each statement were collected using a six point Likert scale [Likert 1932], as this is an established approach in survey research. A statement is considered to be *accepted*, if the majority of students marked a four or higher on the scale. For each sprint, one anonymous questionnaire was formed using these statements. The questionnaires were handed out to the students and answered by them after each sprint presentation. Only the third and last questionnaire contained statement 4, as this statement refers to the overall experience with the code annotations of DecDoc during all sprints. Thus, for this statement the 'not used'-answer was not given. It should be noted, that the total number of seven students is not sufficient to achieve statistical evidence. To address this shortcoming, in each questionnaire the students were also asked for rationale and comments in general and for each statement. Thereby, individual feedback could be collected to better understand the formal answers for the statements.

8.2.3 Results and Discussion

The results of the questionnaires are depicted in Table 8.1. In general, more students used the code annotations over time, what explains the declining amount of 'Not used, no answer'-results for the last sprint. However, the high amount of 'Not used, no answer'-results particularly in the first sprint provides only a limited support for the other statements. But according to the overall number of rejecting answers and the definition for accepting a statement given in Section 8.2.2, no statement had to be rejected. Thus, the results give an indication that it is feasible to document decision knowledge during implementation using DecDoc and its code annotations (requirement C.3). This also concerns the ability of DecDoc to provide generic functionality for accessing and editing knowledge elements provided by extensions within DecDoc, because the code annotations feature provides several new knowledge elements to represent the annotations. Thus, requirement B.2 is covered indirectly by this study.

Sprint no.	Statement no.	Strongly disagree	Disagree	Rather disagree	Rather agree	Agree	Strongly agree	Not used, no answer	∑ Disagree, Agree	Accepted
I	S1	0	0	0	0	1	2	4	0 / 3	yes
	S2	0	0	0	2	1	1	3	0 / 4	yes
	S3	0	1	0	3	1	1	1	1 / 5	yes
II	S1	0	0	0	1	2	1	3	0 / 4	yes
	S2	0	0	0	2	1	1	3	0 / 4	yes
	S3	0	1	0	0	1	3	2	1 / 4	yes
III	S1	0	0	1	0	3	2	1	1 / 5	yes
	S2	0	1	0	0	3	0	3	1 / 3	yes
	S3	0	0	1	0	2	2	2	1 / 4	yes
	S4	0	1	1	0	4	1	-	2 / 5	yes

Table 8.1: Questionnaire Results from all Sprints

In addition, several students described in their individual feedback that DecDoc was very useful for them in order to remember and reflect decisions made during implementation. As the code of the project and the decision documentation emerged over time during three sprints, both the code and the decision knowledge documented through the annotations were versioned and stored in the code repository and the shared knowledge repository of DecDoc. Thus, the study also indicates that requirement B.1 is fulfilled, as the team members were able to access and use this shared decision knowledge. Nevertheless, regarding the usefulness of the annotations, one rejecting answer could be found in the first two questionnaires and multiple rejecting answers were given in the third questionnaire. According to the individual feedback on the questionnaires, it is likely that these results are due to some errors in the integration of annotations and the code versioning system. These errors lead to incorrect locations of already versioned decision annotations within the code. These errors partly are related to the employed Eclipse version 3.7 and were not entirely fixed during the course. Furthermore, some students proposed enhancements in functionality after trying

DecDoc. For instance, it was proposed to add keywords to annotations in order to create references to other decisions when typing the annotation. This finding outlines the importance of supporting the semi-automatic creation of links for DecDoc (requirement D.2).

8.2.4 Threats to Validity

Four different categories of threats to validity have to be considered for this study according to Runeson et al. [Runeson et al. 2012]. These categories are described and discussed in the following paragraphs.

Internal Validity
Threats to internal validity concern the correlation between the investigated factors and other factors [Runeson et al. 2012]. First, the knowledge of the students regarding software engineering was varying. Also, they were not experienced in designing and implementing software. These aspects were addressed by providing a tutorial for the presented documentation tool support to all student teams. Furthermore, the teams were grouped according the subjective experience levels stated by the students. However, it was not possible to compensate the missing experience completely. Second, only links between decision knowledge and code files as external knowledge were covered in the questionnaire by questions concerning the usage of annotations. Thus, links to other kinds of external knowledge, such as use cases or design diagrams, were not investigated. In consequence, the assessment of the approach by the students might be incomplete regarding the linking features of DecDoc. This threat was partly mitigated by exploring the feasibility to link decisions with other related artifacts using DecDoc for documenting the relations between decisions and requirements from the UCI design session transcripts, as described in Section 8.1. For instance, it was demonstrated that the *Decision Overview* graph visualizes links between decision knowledge and related requirements from task description of the UCI design sessions (cf. Figure 8.3).

Construct Validity
Threats to construct validity concern gaps between intended and actual observations of the researchers [Runeson et al. 2012]. Regarding the presented case study, the questionnaire could have measured something different than TAM, because it was not tested and refined before the study. However, this threat was addressed by using questions in the questionnaire, which are typical for TAM.

Reliability Validity
Threats to reliability validity concern the degree to which data and analyses of a study are dependent on specific researchers [Runeson et al. 2012]. Regarding the study presented in this chapter, the reliability validity might be affected, as the students of the practical course knew that the study

authors were both involved in supervising the practical course and investigating DecDoc. Thus, the students could have provided altered documentation results and answers to the questionnaire in order to improve their grades in the course. To mitigate this threat, the investigators were not involved in grading the students.

External Validity

Threats to external validity concern the degree to which the results of the study can be generalized [Runeson et al. 2012]. First, the development project realized by the students during the practical course was small in size regarding the available development time, the amount of requirements to be addressed, and the team size. In consequence, the usefulness of DecDoc for the investigated project might be assessed differently in larger development projects. Also, industry projects are performed with different project settings, so that the results found for student projects are not applicable to them. Nevertheless, the Eclipse IDE is a tool commonly used in industry, and UNICASE was also applied in industry settings [Helming et al. 2009]. Thus, the results of the presented study give a first indication for the usefulness of DecDoc in practice.

Part V

Summary

9 Conclusion and Future Work

In this chapter, the achievements presented in this thesis are summarized and reflected with regard to future work. First, a conclusion of the contributions of the presented work is given. Next, open issues and challenges are outlined as *limitations of this work*. Finally, potential *future work* is discussed based on these limitations.

9.1 Conclusion

The two major goals of this thesis were to investigate decision documentation for a mixed application of different decision making strategies, and decision documentation during different development activities (cf. Section 1.1). To reach these goals, the thesis provides three fundamental contributions.

First, this thesis provided two different studies on the state of practice for decision making and the scientific state of the art for decision documentation. In detail, the actual usage of different decision making strategies by developers was studied by investigating the comments to 260 issue reports collected from the Firefox project. Most importantly, it was found that the developers made the overwhelming majority of decision in a naturalistic way. This finding outlines the importance of documenting knowledge resulting from naturalistic decision making. Moreover, a previous finding of Zannier, Chiasson, and Maurer was confirmed, i.e., that developers tend to mix rational and naturalistic decision making, rather than applying only a single strategy for one decision [Zannier, Chiasson, and Maurer 2007]. In contrast, an analysis of the scientific state of the art for decision documentation showed that most approaches focus on documenting decision knowledge originating from a rational decision making process. In addition, no comprehensive approach was found that provides support for importing and linking decision knowledge from requirements engineering, design and implementation activities. Thus, the need for a new documentation approach for decision knowledge was highlighted, which combines the support for mixed decision making strategies and is capable of using decision knowledge from multiple development activities.

Second, the thesis introduced a new documentation model for decision knowledge based on the requirements derived from the two studies. This documentation model supports the iterative and collaborative documentation of decision knowledge during related development activities. In particular, the model provides a variety of different knowledge elements to document decision knowledge. These elements offer different levels of granularity, such as the coarse-grained *Solution*-element, as well as the fine-grained *Claim*-element. Furthermore, the elements can be combined in a flexible manner, as the model does not prescribe or enforce strict documentation structures. Instead, the knowledge elements can be nested and linked, as necessary. In particular, knowledge elements originating either from rational or naturalistic decision making can be mixed. Thereby, the iterative decision documentation is supported. Also, multiple developers can use the model for collaborative documentation of decision knowledge. To this end, the model provides a capturing support for decision knowledge emerging during requirements engineering, design, and implementation. In more detail, the model is integrated with the UNICASE knowledge model, so that decision knowledge can be linked to use cases, UML design diagrams and their contained entities, as well as to code statements. Moreover, an evaluation of the model was provided in the form of a case study on documented decision knowledge in design session transcripts from the UCI design workshop. This evaluation showed the feasibility of documenting decision knowledge originating from mixed decision making strategies in an iterative manner using the documentation model.

Third, this thesis contributed the tool support DecDoc to facilitate an effective and convenient use of the model in practice. Therefore, UNICASE was extended with several plugins, which provide editing and capturing support for the documentation model. More specifically, DecDoc provides both, a general editor for all decision knowledge elements from the model, as well as a specialized support for selected model elements. For instance, an editor for matching *Problem* and *Solution* elements is provided. Moreover, DecDoc offers capturing support for decision knowledge originating from security-related requirements through integrating the HeRA plugin. Also, the UNICASE UML editor was extended, such that decision knowledge can be captured and documented when editing UML diagrams. Most importantly, DecDoc provides specialized code annotations for decisions knowledge, which can be used during implementation by the developers. Thereby, decision knowledge can be captured similar to `JavaDoc` documentation directly within the code. Finally, the feasibility of documenting decisions with DecDoc was demonstrated with complex examples from the transcripts of the UCI design workshop. In addition, a case study with students using DecDoc was performed in a practical course on software engineering.

9.2 Limitations of this Work

Regarding the contributions, several limitations exist, which are discussed in the following paragraphs.

The findings of the two studies in Chapter 3 and 4 indicate that developers use both rational and naturalistic decision making for their decisions and document the knowledge according to the applied strategy. Thus, it is important to highlight, that it is not yet clear, whether this mix of the two strategies actually provides the best results for the decisions to be made. Therefore, the actual outcome of a decision needs to be measured and evaluated, for instance in terms of economic value or benefits in software quality and maintenance. Regarding the data investigated in this thesis, such measurements were neither in the focus of the thesis nor applicable for the data. For the Firefox issue reports, the decision outcome could not be determined exactly, whereas the outcome of the decisions described in the UCI design session transcripts remains unclear, because those decisions were never actually implemented in software. Also, the studies do not provide an insight, whether it is advantageous to document decisions according to one strategy in order to make the documentation comprehensive and accessible for future exploration, even if the decision was made by applying a different strategy. The variety of scientific approaches, which support RDM rather than NDM, might indicate an existing preference for documenting decision knowledge according to RDM. However, there might be a difference between the preferences of developers to document their decision knowledge, as it emerged, and the future users of this knowledge, such as researchers and other developers. This was reflected by supporting both RDM and NDM in the documentation model. Nevertheless, this estimated difference was not addressed by this thesis and requires further investigation.

It may be seen as a limitation of the documentation model that it allows documented decision knowledge to be inconsistent. Whereas this characteristic of the model is caused by its flexible structure and thereby addresses the iterative documentation support, it may complicate the further analysis and exploitation of the documented knowledge due to missing or corrupted knowledge structures. In addition, a mixed documentation according to different decision making strategies and different abstraction levels of knowledge elements might decrease the comprehensibility of the documentation. Several measures address this limitation of the model. First, developers need to identify and discuss any issues of ambiguity and potential mistakes in the documentation, which was created using the decision documentation model. This is likely to happen, as developers will collaborate in any case during their development activities, for instance, when they discuss design options or merge code. Then, the developers may add the results of their discussions as new knowledge elements or adapt the relations of existing ones to optimize the documentation. Second, the manual documentation effort by developers needs to be lowered in order to avoid mistakes and cover all important knowledge sources. Whereas the documentation model and DecDoc already offer several possibilities for this knowledge import, further knowledge sources still remain unused. For instance, analyses of requirements and design specifications in Word documents (cf. [Jansen, Avgeriou, and Ven 2009]) can provide further semi-automatic input for decision knowledge from external artifacts. However, this requires specialized approaches for both extracting the related knowledge from the artifacts and storing the related knowledge in a shared knowledge repository. Thus, this is beyond the scope of this thesis.

The tool support DecDoc is limited by its implementation as an Eclipse plugin, which binds DecDoc to a specific version of Eclipse with all potential and actual errors and workarounds for this version. Also, DecDoc is currently only integrated with SVN code repository system. Nevertheless, other popular IDEs, such as IntelliJ IDEA or Microsoft Visual Studio, and code repositories, such as Git, exist and are commonly used in industry. However, providing DecDoc for these IDEs would also require providing UNICASE and the EMFStore within these environments. An integration of DecDoc with Git would require a detailed analysis of the Git branching concept with a similar concept for the EMFStore. As these prerequisites are currently not fulfilled, this thesis focused on providing DecDoc within the Eclipse environment for SVN. Next, DecDoc is currently focused on capturing and structuring decision knowledge for documentation purposes. Whereas the dashboard and the graph visualizations of decision knowledge elements in general and *Arguments* in particular already address the exploitation and reflection of decision knowledge, these features are not yet fully comprehensive. For instance, a more sophisticated support for reviews of decisions could be provided in the dashboard by analyzing the *contains*-relation between instances of decision knowledge elements. However, further studies on the documentation model and DecDoc are required to determine the necessity of and guidelines for these features.

9.3 Future Work

Based on the results and limitations of this thesis, the following points of future research should be outlined:

Value of Decision Documentation
As the documentation of decision knowledge typically leads to an increased effort for developers, it appears to be important to determine how the value of this documentation can be maximized, so that this effort is indeed invested. Therefore, it is necessary to investigate which decision making strategies and decision knowledge elements are required to support specific usages. For instance, it was already shown by Davide Falessi, Cantone, and Becker that documented RDM decision knowledge elements helped developers to improve their understanding and time-efficiency for decision making processes of future decisions. For decisions made in a naturalistic manner, the upside of documentation could be even higher, because the decision knowledge for these decisions was found to be over-represented

compared with decision knowledge resulting from RDM. If documentation for these decisions is now supported, the related decision knowledge remains visible and helps developers to understand the requirements, design and implementation of the system. Thus, lowered maintenance costs and an improved integrity of design could be the benefits of documenting these decisions.

Focus on Important Decisions

In practice, it is not realistic for developers to document all decisions explicitly. Beside the required effort for documentation, this is caused by individual differences between developers regarding their preferences in creative and reflective thinking when making decisions [Razavian et al. 2015]. Thus, DecDoc and many related tools focus on design decisions with importance for the success of the development project, such as architectural design decisions (cf. [Jansen and Bosch 2005; Capilla, Nava, Pérez, et al. 2006; Zimmermann, Koehler, et al. 2009]). Nevertheless, it remains an open question how this importance of particular decisions can be determined early in the decision making process. A prominent example is given by the study of Chen, Ali Babar, and Nuseibeh, which shows the difficulty of identifying decisions on architecturally significant requirements [Chen, Ali Babar, and Nuseibeh 2013]. Here, more research is required to determine classifications of important decisions and rule sets to identify them.

Consistency between Documentation and Implementation of Decisions

Documentation tools, such as DecDoc, could be extended to support consistency checks between documented decision knowledge and the actual implementation of these decisions within the code. Thereby, the awareness of developers for these decisions would be increased, so that the decision can either be implemented deliberately or challenged explicitly. However, such consistency checks are difficult to realize based on documentation. Typically, a formal verification for both the decision documentation and the implementation artifacts is required to provide such consistency checks (cf. [Zimmermann, Gschwind, et al. 2007; Cleland-Huang et al. 2013]). Thus, investigating and realizing such consistency checks requires dedicated future research projects.

Maintenance of Documentation

The evolution of documented decision knowledge still remains an open challenge. Whereas DecDoc alleviates the recognition and adaption of outdated or incorrect documentation due to the collaborative work of developers on this documentation, the principal problem remains: Decision knowledge becomes outdated or even wrong, as the system under development and its design decisions evolve over time. Current approaches show that this phenomenon cannot be addressed completely in an automated way (cf. [Capilla, Nava, and Duenas 2007; Capilla, Zimmermann, et al. 2011]). Instead, developers are required to explicitly maintain their documentation and update it, if necessary. Future research should identify measures and rules to trigger this maintenance process semi-automatically, because developers will not be able to check huge amounts of documented knowledge for necessary changes manually.

A

Literature Review Hits

In this appendix, the complete list of all included search hits for the literature review (cf. Section 4) is provided. The type of hit *A* indicates a search hit generated by automatic search, while *M* indicates hits resulting from manual search. An *ES* indicates hits containing an existing study comparing either existing approaches or tools for documenting decision knowledge. The hits presented in Table A.1 were included within the literature review after the second exclusion round.

Type of Hit	Sources	Amount
A	[Rockwell et al. 2009; Konemann 2009; Wang and Burge 2010; Smith, Bohner, and McCrickard 2005; Davide Falessi, Cantone, and Becker 2006; Davide Falessi, Becker, and Cantone 2006; David Falessi, Cantone, and Kruchten 2008; Davide Falessi, Capilla, and Cantone 2008; Aurum, Wohlin, and Porter 2006; Canfora, Casazza, and De Lucia 2000; López et al. 2012; Jansen, Avgeriou, and Ven 2009; Jansen, Bosch, and Avgeriou 2008; Zimmermann, Gschwind, et al. 2007; Capilla, Zimmermann, et al. 2011; Manteuffel, Tofan, Avgeriou, et al. 2016]	16
A, ES	[Alexeeva, Perez-Palacin, and Mirandola 2016; Ali Babar, Boer, et al. 2007; Tang, Avgeriou, et al. 2010; Weinreich and Groher 2016; Ding et al. 2014; Capilla, Jansen, et al. 2016; Li, Liang, and Avgeriou 2013; Tofan et al. 2014]	8
M	[Ali Babar, Gorton, and Kitchenham 2006; Ali Babar and Gorton 2007; Capilla, Nava, and Duenas 2007; Jansen and Bosch 2005; Tyree and Akerman 2005; Burge and Brown 2004; Burge and Brown 2008; Kruchten, Lago, and Vliet 2006; Tang, Jin, and Han 2007; Buchgeher and Weinreich 2011; Gaubatz, Lytra, and Zdun 2015; Zimmermann, Koehler, et al. 2009; Cleland-Huang et al. 2013; Nowak and Pautasso 2013; Manteuffel, Tofan, Koziolek, et al. 2014]	15

Table A.1: Overview of Search Hits included within the Literature Review

List of References

ACM Digital Library (2017). URL: `http://dl.acm.org/` (visited on 05/2017) (cited on p. 63).

Alexeeva, Zoya, Diego Perez-Palacin, and Raffaela Mirandola (2016). "Design Decision Documentation: A Literature Overview". In: *Software Architecture: 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 – December 2, 2016, Proceedings*. Ed. by Bedir Tekinerdogan, Uwe Zdun, and Muhammad Ali Babar. Springer International Publishing, pp. 84–101 (cited on pp. 59, 177).

Ali Babar, Muhammad, Remco C. de Boer, Torgeir Dingsoyr, and Rik Farenhorst (2007). "Architectural Knowlege Management Strategies: Approaches in Research and Industry". In: *Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK/ADI'07)*. IEEE, pp. 35–41 (cited on pp. 5, 7, 8, 27, 60, 177).

Ali Babar, Muhammad and Ian Gorton (2007). "A Tool for Managing Software Architecture Knowledge". In: *Proceedings of the Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK/ADI'07: ICSE Workshops 2007)*. IEEE, pp. 11–17 (cited on pp. 20, 72, 177).

Ali Babar, Muhammad, Ian Gorton, and Barbara Kitchenham (2006). "A Framework for Supporting Architecture Knowledge and Rationale Management". In: *Rationale Management in Software Engineering*. Springer (cited on pp. 67, 69, 70, 72, 75, 177).

Apache Subversion (2016). URL: `https://subversion.apache.org/` (visited on 11/2016) (cited on p. 20).

Arnott, David and Graham Pervan (2008). "Eight key issues for the decision support systems discipline". In: *Decision Support Systems* 44.3, pp. 657–672 (cited on pp. 4, 28).

Atlassian JIRA (2017). URL: `https://de.atlassian.com/software/jira` (visited on 07/2017) (cited on p. 19).

Aurum, Aybüke, Claes Wohlin, and Andrew Porter (2006). “Aligning Software Project Decisions: A case study”. In: *International Journal of Software Engineering and Knowledge Engineering* 16.06, pp. 795–818 (cited on pp. 5, 8, 27, 177).

Baker, Alex and André van der Hoek (2010). “Ideas, subjects, and cycles as lenses for understanding the software design process”. In: *Design Studies* 31.6, pp. 590–613 (cited on p. 135).

Ball, Linden J., Balder Onarheim, and Bo T. Christensen (2010). “Design requirements, epistemic uncertainty and solution development strategies in software design”. In: *Design Studies* 31.6, pp. 567–589 (cited on p. 135).

Basili, Victor R., Gianluigi Caldiera, and Dieter H. Rombach (1994). “Goal Question Metric Paradigm”. In: *Encyclopedia of Software Engineering*. Ed. by John J. Marciniak. New York: Wiley-Interscience, pp. 528–532 (cited on pp. 34, 58, 134).

Bruegge, Bernd, Oliver Creighton, Jonas Helming, and Maximilian Koegel (2008). “Unicase - An Ecosystem for Unified Software Engineering Research Tools”. In: *International Conference on Global Software Engineering*. IEEE, pp. 1–6 (cited on p. 29).

Buchgeher, Georg and Rainer Weinreich (2011). “Automatic tracing of decisions to architecture and implementation”. In: *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*. IEEE, pp. 46–55 (cited on pp. 67, 69, 72–74, 76, 177).

Burge, Janet E. and David C. Brown (2004). “An Integrated Approach for Software Design Checking Using Design Rationale”. In: *Proceedings of the First International Conference of Design Computing and Cognition*. Berlin, Heidelberg: Springer, pp. 557–576 (cited on pp. 67–70, 72, 73, 76, 177).

— (2008). “Software Engineering Using RATionale”. In: *Journal of Systems and Software* 81.3, pp. 395–413 (cited on pp. 5, 8, 27, 28, 177).

Canfora, Gerardo, Gerardo Casazza, and Andrea De Lucia (2000). “A Design Rationale based Environment for Cooperative Maintenance”. In: *International Journal of Software Engineering and Knowledge Engineering* 10.5, pp. 627–645 (cited on pp. 3, 5, 8, 27, 67, 69, 72, 73, 177).

Capilla, Rafael and Muhammad Ali Babar (2008). “On the role of architectural design decisions in software product line engineering”. In: *European Conference on Software Architecture*. Springer, pp. 241–255 (cited on p. 20).

Capilla, Rafael, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar (2016). “10 years of software architecture knowledge management: Practice and future”. In: *Journal of Systems and Software* 116, pp. 191–205 (cited on pp. 60, 177).

Capilla, Rafael, Francisco Nava, and Juan C. Duenas (2007). “Modeling and Documenting the Evolution of Architectural Design Decisions”. In: *Proceedings of the Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK/ADI’07: ICSE Workshops 2007)*. IEEE, p. 9 (cited on pp. 8, 67–69, 72, 175, 177).

Capilla, Rafael, Francisco Nava, Sandra Pérez, and Juan C. Dueñas (2006). “A web-based tool for managing architectural design decisions”. In: *ACM SIGSOFT Software Engineering Notes* 31.5, pp. 1–8 (cited on pp. 8, 28, 175).

Capilla, Rafael, Olaf Zimmermann, Uwe Zdun, Paris Avgeriou, and Jochen M. Kuester (2011). "An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle". In: *5th European Conference, ECSA 2011, Essen, Germany*. Springer, pp. 303–318 (cited on pp. 67, 69, 72, 175, 177).

Carroll, John M. and Mary Beth Rosson (1992). "Getting around the task-artifact cycle: how to make claims and design by scenario". In: *ACM Transactions on Information Systems (TOIS)* 10.2, pp. 181–212 (cited on p. 27).

Cayrol, Claudette and Marie-Christine Lagasquie-Schiex (2009). "Bipolar abstract argumentation systems". In: *Argumentation in Artificial Intelligence*. Ed. by Iyad Rahwan. New York: Springer US, pp. 65–84 (cited on p. 94).

Chan, Siew H. and Qian Song (2010). "Motivational Framework: Insights into Decision Support System Use and Decision Performance". In: *Decision Support Systems*. Ed. by Chiang S. Jao. 1st ed. Vukovar: InTech, pp. 1–24 (cited on p. 28).

Chen, Lianping, Muhammad Ali Babar, and Bashar Nuseibeh (2013). "Characterizing Architecturally Significant Requirements". In: *IEEE Software* 30.2, pp. 38–45 (cited on pp. 3, 19, 153, 175).

Christiaans, Henri and Rita Assoreira Almendra (2010). "Accessing decision-making in software design". In: *Design Studies* 31.6, pp. 641–662 (cited on p. 135).

Cicchetti, Domenic V. and Sara A. Sparrow (1981). "Developing criteria for establishing interrater reliability of specific items: Applications to assessment of adaptive behavior". In: *American Journal of Mental Deficiency* 86.2, pp. 127–137 (cited on p. 42).

Cleland-Huang, Jane, Mehdi Mirakhorli, Adam Czauderna, and Mateusz Wieloch (2013). "Decision-centric traceability of architectural concerns". In: *7th Int. Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13)*. IEEE, pp. 5–11 (cited on pp. 67–69, 72–76, 175, 177).

Cohen, Jacob (1988). *Statistical Power Analysis for the Behavioral Sciences*. 2nd. Hillsdale: Erlbaum Associates (cited on p. 44).

Conklin, Jeff and Michael L. Begeman (1988). "gIBIS: A hypertext tool for exploratory policy discussion". In: *ACM Transactions on Information Systems (TOIS)* 6.4, pp. 303–331 (cited on p. 26).

Davis, Fred D., Richard P. Bagozzi, and Paul R. Warshaw (1989). "User Acceptance of Computer Technology: A Comparison of Two Theoretical Models". In: *Management Science* 35.8, pp. 982–1002 (cited on p. 164).

DecDoc Update Site (2017). URL: `http://svn.ifi.uni-heidelberg.de/unicase/0.5.2/ures/decdoc-features/` (visited on 12/2017) (cited on p. 111).

DecDoc User Manual (2017). URL: `https://svn.ifi.uni-heidelberg.de/unicase/0.5.2/ures/manual_decision_editor_v01.pdf` (visited on 12/2017) (cited on p. 111).

Dilmaghani, Ania and James Dibble (2012). "Strategies for Early-Stage Collaborative Design". In: *Software* 29.1, pp. 39–45 (cited on p. 136).

Ding, Wei, Peng Liang, Antony Tang, and Hans van Vliet (2014). "Knowledge-based approaches in software documentation: A systematic literature review". In: *Information and Software Technology* 56.6, pp. 545–567 (cited on pp. 59, 177).

Eclipse Project (2016). URL: `http://www.eclipse.org/` (visited on 11/2016) (cited on pp. 20, 112).

Elsevier ScienceDirect (2017). URL: `http://www.sciencedirect.com/` (visited on 05/2017) (cited on p. 63).

EMF Client Platform (2016). URL: `http://www.eclipse.org/ecp/` (visited on 11/2016) (cited on p. 29).

EMF Store (2016). URL: `http://eclipse.org/emfstore/` (visited on 11/2016) (cited on pp. 29, 112).

Falessi, David, Giovanni Cantone, and Philippe Kruchten (2008). "Value-Based Design Decision Rationale Documentation: Principles and Empirical Feasibility Study". In: *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*. IEEE, pp. 189–198 (cited on pp. 65, 177).

Falessi, Davide, Martin Becker, and Giovanni Cantone (2006). "Design Decision Rationale: Experiences and Steps Ahead Towards Systematic Use". In: *SIGSOFT Software Engineering Notes* 31.5 (cited on pp. 65, 67–69, 72–74, 177).

Falessi, Davide, Giovanni Cantone, and Martin Becker (2006). "Documenting Design Decision Rationale to Improve Individual and Team Design Decision Making: An Experimental Evaluation". In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering - ISESE '06*. ACM, pp. 134–143 (cited on pp. 3, 5, 20, 65, 174, 177).

Falessi, Davide, Giovanni Cantone, Rick Kazman, and Philippe Kruchten (2011). "Decision-making techniques for software architecture design". In: *ACM Computing Surveys* 43.4, pp. 1–28 (cited on pp. 7, 20, 27, 48).

Falessi, Davide, Rafael Capilla, and Giovanni Cantone (2008). "A Value-Based Approach for Documenting Design Decisions Rationale: A Replicated Experiment". In: *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge (SHARK'08)*. ACM Press, p. 63 (cited on pp. 65, 105, 177).

Firefox project (2015). *Issue tracking system Bugzilla*. (visited on 08/2015). URL: `https://bugzilla.mozilla.org/page.cgi?id=productdash%5C%5Cboard.html%5C&product=Firefox%5C&bug%5C_status=open%5C&tab=components` (cited on p. 41).

Gamer, Matthias, Jim Lemon, Ian Fellows, and Puspendra Singh (2012). *irr: Various Coefficients of Interrater Reliability and Agreement. R package version 0.84.* (visited on 08/2015). URL: `http://CRAN.R-project.org/package=irr` (cited on p. 42).

Gärtner, Stefan, Thomas Ruhroth, Jens Bürger, Kurt Schneider, and Jan Jürjens (2014). "Maintaining Requirements for Long-Living Software Systems by Incorporating Security Knowledge". In: *Proc. of the 22th International Requirements Engineering Conference*. IEEE (cited on p. 100).

Gaubatz, Patrick, Ioanna Lytra, and Uwe Zdun (2015). “Automatic enforcement of constraints in real-time collaborative architectural decision making”. In: *Journal of Systems and Software* 103, pp. 128–149 (cited on pp. 67–70, 72, 73, 75, 76, 177).

Geer, David (2005). “Eclipse becomes the dominant Java IDE”. In: *IEEE Computer* 38.7, pp. 16–18 (cited on p. 20).

Git (2016). URL: `https://git-scm.com/` (visited on 11/2016) (cited on p. 20).

Gore, Julie, Adrian Banks, Lynne Millward, and Olivia Kyriakidou (2006). “Naturalistic Decision Making and Organizations: Reviewing Pragmatic Science”. In: *Organization Studies* 27.7, pp. 925–942 (cited on p. 25).

Hansman, Simon and Ray Hunt (2005). “A taxonomy of network and computer attacks”. In: *Computers & Security* 24.1, pp. 31–43 (cited on p. 100).

Helming, Jonas, Joern David, Maximilian Koegel, and Helmut Naughton (2009). “Integrating System Modeling with Project Management - A Case Study”. In: *33rd Annual IEEE International Computer Software and Applications Conference*. IEEE, pp. 571–578 (cited on p. 167).

Herold, Sebastian, Holger Klus, Yannick Welsch, Constanze Deiters, et al. (2008). “CoCoME - The Common Component Modeling Example”. In: *The Common Component Modeling Example*. Ed. by Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil. Springer, pp. 16–53 (cited on p. 81).

Hesse, Tom-Michael, Stefan Gaertner, Tobias Roehm, Barbara Paech, Kurt Schneider, and Bernd Bruegge (2014). “Semiautomatic security requirements engineering and evolution using decision documentation, heuristics, and user monitoring”. In: *Proceedings of the 1st International Workshop on Evolving Security and Privacy Requirements Engineering (ESPRE)*. IEEE, pp. 1–6 (cited on pp. 14, 19, 99–102, 113, 119).

Hesse, Tom-Michael, Christian Kuecherer, and Barbara Paech (2015). “Experiences with Supporting the Distributed Responsibility for Requirements through Decision Documentation”. In: *Softwaretechnik-Trends* 35.1, pp. 14–15 (cited on p. 14).

Hesse, Tom-Michael, Arthur Kuehlwein, Barbara Paech, Tobias Roehm, and Bernd Bruegge (2015). “Documenting Implementation Decisions with Code Annotations”. In: *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*. KSI Research, pp. 152–157 (cited on pp. 14, 106–108, 159).

Hesse, Tom-Michael, Arthur Kuehlwein, and Tobias Roehm (2016). “DecDoc: A Tool for Documenting Design Decisions Collaboratively and Incrementally”. In: *2016 1st International Workshop on Decision Making in Software ARCHitecture (MARCH)*. IEEE, pp. 30–37 (cited on pp. 15, 81, 82, 111, 159).

Hesse, Tom-Michael, Veronika Lerche, Marcus Seiler, Konstantin Knoess, and Barbara Paech (2016). “Documented decision-making strategies and decision knowledge in open source projects: An empirical study on Firefox issue reports”. In: *Information and Software Technology* 79, pp. 36–51 (cited on pp. 13, 18, 21–23, 33, 35, 37, 38, 40, 49).

Hesse, Tom-Michael and Barbara Paech (2013). "Supporting the Collaborative Development of Requirements and Architecture Documentation". In: *Proceedings of the 3rd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks'13)*. IEEE, pp. 22–26 (cited on p. 14).

— (2016). "Documenting Relations Between Requirements and Design Decisions: A Case Study on Design Session Transcripts". In: *Requirements Engineering: Foundation for Software Quality: 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings*. Ed. by Maya Daneva and Oscar Pastor. Cham: Springer International Publishing, pp. 188–204 (cited on pp. 14, 77, 131, 141, 144, 152).

Hoek, André van der, Marian Petre, and Alex Baker (2010). *Workshop "Studying Professional Software Design" at University of California, Irvine*. visited on 12/2017. URL: `http://www.ics.uci.edu/design-workshop/` (cited on p. 131).

IBM Rational DOORS (2017). URL: `https://www.ibm.com/software/products/de/ratidoor` (visited on 07/2017) (cited on p. 19).

IEEExplore (2017). URL: `http://ieeexplore.ieee.org/` (visited on 05/2017) (cited on p. 63).

Jackson, Michael (2010). "Representing structure in a software system design". In: *Design Studies* 31.6, pp. 545–566 (cited on pp. 134–136).

Jansen, Anton, Paris Avgeriou, and Jan Salvador van der Ven (2009). "Enriching software architecture documentation". In: *Journal of Systems and Software* 82.8, pp. 1232–1248 (cited on pp. 29, 67, 69, 70, 72, 73, 76, 174, 177).

Jansen, Anton and Jan Bosch (2005). "Software Architecture as a Set of Architectural Design Decisions". In: *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE, pp. 109–120 (cited on pp. 3–5, 8, 20, 59, 67–69, 72, 105, 175, 177).

Jansen, Anton, Jan Bosch, and Paris Avgeriou (2008). "Documenting after the fact: Recovering architectural design decisions". In: *Journal of Systems and Software* 81.4, pp. 536–557 (cited on pp. 67–70, 72, 73, 76, 177).

Javadoc Documentation by Oracle (2016). URL: `http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html` (visited on 11/2016) (cited on pp. 20, 123).

Jiang, Li and Armin Eberlein (2003). "Decision support for requirements engineering process development". In: *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*. Vol. 2. IEEE, pp. 1359–1362 (cited on p. 19).

Jonassen, David (2012). "Designing for decision making". In: *Educational Technology Research and Development* 60.2, pp. 341–359 (cited on pp. 21, 23).

Kahneman, Daniel (2011). *Thinking, fast and slow*. Farrar, Straus and Giroux (cited on p. 22).

Khomh, Foutse, Tejinder Dhaliwal, Ying Zou, and Bram Adams (2012). "Do Faster Releases Improve Software Quality? An Empirical Case Study of Mozilla Firefox". In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, pp. 179–188 (cited on pp. 39, 41).

Kitchenham, Barbara and Stuart Charters (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. Keele University, Durham University, Joint Report (cited on pp. 61, 78).

Klein, Gary (2008). “Naturalistic Decision Making”. In: *Human Factors* 50.3, pp. 456–460 (cited on pp. 7, 23, 24, 39, 40).

Klein, Gary, Roberta Calderwood, and Anne Clinton-Cirocco (2010). “Rapid Decision Making on the Fire Ground: The Original Study Plus a Postscript”. In: *Journal of Cognitive Engineering and Decision Making* 4.3, pp. 186–209 (cited on pp. 23, 37).

Klein, Gary and David Klinger (1991). “Naturalistic Decision Making”. In: *Gateway* 11.3, pp. 16–19 (cited on pp. 18, 24).

Knoess, Konstantin (2014). “Decision making in software engineering: An issue tracker analysis”. MA thesis. Heidelberg University (cited on p. 37).

Ko, Andrew J. and Parmit K. Chilana (2011). “Design, discussion, and dissent in open bug reports”. In: *Proceedings of the 2011 iConference*. ACM, pp. 106–113 (cited on pp. 8, 20, 29, 34–36, 54).

Ko, Andrew J., Robert DeLine, and Gina Venolia (2007). “Information Needs in Collocated Software Development Teams”. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE’07)*. IEEE, pp. 344–353 (cited on pp. 5, 50).

Konemann, Patrick (2009). “Integrating decision management with UML modeling concepts and tools”. In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. IEEE, pp. 297–300 (cited on pp. 67–69, 72, 73, 75, 76, 177).

Kruchten, Philippe, Patricia Lago, and Hans van Vliet (2006). “Building Up and Reasoning About Architectural Knowledge”. In: *Quality of Software Architectures*. Ed. by Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner. Vol. 4214. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 43–58 (cited on pp. 26, 67–69, 72, 177).

Kuehlwein, Arthur (2014). *Documentation of decisions during the implementation phase through code annotations*. BA thesis. Heidelberg University (cited on p. 106).

Kunz, Werner and Horst WJ Rittel (1970). *Issues as elements of information systems*. Vol. 131. Institute of Urban and Regional Development, University of California Berkeley, California (cited on p. 26).

Lee, Jintae (1989). “Decision representation language (DRL) and its support environment”. In: (cited on p. 27).

— (1991). “Extending the Potts and Bruns model for recording design rationale”. In: *Proceedings of the 13th International Conference on Software Engineering*. IEEE, pp. 114–125 (cited on pp. 11, 27).

Li, Zengyang, Peng Liang, and Paris Avgeriou (2013). “Application of knowledge-based approaches in software architecture: A systematic mapping study”. In: *Information and Software Technology* 55.5, pp. 777–794 (cited on pp. 58, 59, 177).

Likert, Rensis (1932). “A Technique for the Measurement of Attitudes”. In: *Archives of Psychology* 22.140, pp. 1–55 (cited on p. 164).

Lipshitz, Raanan, Gary Klein, Judith Orasanu, and Eduardo Salas (2001). "Taking Stock of Naturalistic Decision Making". In: *Journal of Behavioral Decision Making* 14.5, pp. 331–352 (cited on pp. 4, 21, 23, 25, 40).

López, Claudia, Víctor Codocedo, Hernán Astudillo, and Luiz Marcio Cysneiros (2012). "Bridging the gap between software architecture rationale formalisms and actual architecture documents: An ontology-driven approach". In: *Science of Computer Programming* 77.1, pp. 66–80 (cited on pp. 67–69, 72, 73, 177).

Lougher, Robert and Tom Rodden (1993). "Supporting Long-term Collaboration in Software Maintenance". In: *Proceedings of the Conference on Organizational Computing Systems - COCS '93*. ACM, pp. 228–238 (cited on pp. 3, 5, 8, 20, 27).

MacLean, Allan, Richard M Young, Victoria M E Bellotti, and Thomas P Moran (1991). "Questions, Options, and Criteria: Elements of Design Space Analysis". In: *Human-Computer Interaction* 6.3-4, pp. 201–250 (cited on pp. 11, 26, 27).

Manteuffel, Christian, Dan Tofan, Paris Avgeriou, Heiko Koziolek, and Thomas Goldschmidt (2016). "Decision architect – A decision documentation tool for industry". In: *Journal of Systems and Software* 112, pp. 181–198 (cited on pp. 67–69, 72, 73, 75, 76, 177).

Manteuffel, Christian, Dan Tofan, Heiko Koziolek, Thomas Goldschmidt, and Paris Avgeriou (2014). "Industrial Implementation of a Documentation Framework for Architectural Decisions". In: *Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA'14)*. IEEE, pp. 225–234 (cited on pp. 5, 72, 73, 177).

Maule, John A. (2010). "Can Computers Help Overcome Limitations in Human Decision Making?" In: *International Journal of Human-Computer Interaction* 26.2-3, pp. 108–119 (cited on pp. 7, 21, 23, 28).

Mayring, Philipp (2010). "Qualitative Inhaltsanalyse". German. In: *Handbuch Qualitative Forschung in der Psychologie*. Ed. by Günter Mey and Katja Mruck. Wiesbaden: VS Verlag für Sozialwissenschaften, pp. 601–613 (cited on p. 41).

Mentis, Helena M., Paula M. Bach, Blaine Hoffman, Mary Beth Rosson, and John M. Carroll (2009). "Development of Decision Rationale in Complex Group Decision Making". In: *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*. ACM, pp. 1341–1350 (cited on pp. 35, 36).

Nakakoji, Kumiyo, Yasuhiro Yamamoto, Nobuto Matsubara, and Yoshinari Shirai (2012). "Toward Unweaving Streams of Thought for Reflection in Professional Software Design". In: *IEEE Software* 29.1, pp. 34–38 (cited on p. 136).

Ngo, The and Guenther Ruhe (2005). "Decision Support in Requirements Engineering". In: *Engineering and Managing Software Requirements*. Ed. by A. Aybüke and C. Wohlin. Berlin, Heidelberg: Springer, pp. 267–286 (cited on pp. 5, 8, 17, 25, 26).

Nowak, Marcin and Cesare Pautasso (2013). "Team situational awareness and architectural decision making with the software architecture warehouse". In: *Software Architecture: 7th European Conference (ECSA'13)*. Springer, pp. 146–161 (cited on pp. 67–69, 72, 73, 177).

Nuseibeh, Bashar (2001). "Weaving Together Requirements and Architectures". In: *IEEE Computer* 34.3, pp. 115–119 (cited on p. 8).

Orasanu, Judith and Terry Connolly (1993). "The reinvention of decision making". In: *Decision making in action: Models and methods*. Ed. by Gary Klein, Judith Oransanu, R Calderwood, and C E Zsambok. Westport: Ablex Publishing, pp. 3–20 (cited on pp. 23, 24, 39).

Paech, Barbara, Alexander Delater, and Tom-Michael Hesse (2014). "Supporting Project Management Through Integrated Management of System and Project Knowledge". In: *Software Project Management in a Changing World*. Ed. by Guenther Ruhe and Claes Wohlin. Berlin, Heidelberg: Springer, pp. 157–192 (cited on pp. 13, 17, 18, 21, 23, 27, 28, 57, 62, 63, 65, 67, 69, 72, 76).

Papyrus (2016). URL: `https://eclipse.org/papyrus/` (visited on 11/2016) (cited on p. 113).

Petre, Marian and André van der Hoek (2013). *Software Designers in Action: A Human-Centric Look at Design Work*. CRC Press (cited on pp. 36, 135).

R Core Team (2014). *R: A language and environment for statistical computing*. (visited on 08/2015). URL: `http://www.R-project.org/` (cited on p. 42).

Razavian, Maryam, Antony Tang, Rafael Capilla, and Patricia Lago (2016). "In two minds: how reflections influence software design thinking". In: *Journal of Software: Evolution and Process* 28.6, pp. 394–426 (cited on p. 22).

Razavian, Maryam, Antony Tang, Rafael Capilla, Patricia Lago, et al. (2015). *In Two Minds: How Reflections Influence Software Design Thinking*. Tech. rep. VU University Amsterdam (cited on p. 175).

Rekha, Smrithi V and Henry Muccini (2014). "A study on group decision-making in software architecture". In: *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*. IEEE, pp. 185–194 (cited on p. 8).

Rockwell, Justin A., Ian R. Grosse, Sundar Krishnamurty, and Jack C. Wileden (2009). "A Decision Support Ontology for collaborative decision making in engineering design". In: *2009 International Symposium on Collaborative Technologies and Systems*. IEEE, pp. 1–9 (cited on pp. 67–70, 72, 73, 177).

Rooksby, John and Nozomi Ikeya (2012). "Collaboration in Formative Design: Working Together at a Whiteboard". In: *IEEE Software* 29.1, pp. 56–60 (cited on p. 136).

Ruhe, Guenther (2003). "Software Engineering Decision Support – A New Paradigm for Learning Software Organizations". English. In: *Advances in Learning Software Organizations*. Ed. by Scott Henninger and Frank Maurer. Vol. 2640. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 104–113 (cited on p. 3).

Runeson, Per, Martin Höst, Austen Rainer, and Björn Regnell (2012). *Case Study Research in Software Engineering. Guidelines and Examples*. 1st. Hoboken: Wiley, p. 237 (cited on pp. 37, 53, 54, 77, 78, 155–157, 166, 167).

Saaty, Thomas L (2008). “Decision making with the analytic hierarchy process”. In: *International Journal of Services Sciences* 1.1, pp. 83–98 (cited on pp. 21, 22).

Shaw, Mary (2012). “The role of design spaces”. In: *IEEE Software* 29.1, pp. 46–50 (cited on pp. 133, 134, 136, 143, 144, 147, 156).

Shrout, Patrick E and Joseph L Fleiss (1979). “Intraclass correlations: Uses in assessing rater reliability”. In: *Psychological Bulletin* 86.2, pp. 420–428 (cited on p. 42).

Smith, Jamie L, Shawn A Bohner, and D. Scott McCrickard (2005). “Project management for the 21st century: supporting collaborative design through risk analysis”. In: *Proceedings of the 43rd annual Southeast regional conference (ACM-SE 43)*. Vol. 2. ACM Press, pp. 300–305 (cited on pp. 67, 69, 70, 72, 73, 76, 177).

Sommerville, Ian (2010). *Software Engineering*. 9th. USA: Addison-Wesley Publishing Company (cited on pp. 18–20).

Souza, Rodrigo, Christina Chavez, and Roberto A. Bittencourt (2014). “Do Rapid Releases Affect Bug Reopening? A Case Study of Firefox”. In: *Brazilian Symposium on Software Engineering (SBES)*. IEEE, pp. 31–40 (cited on p. 54).

SpringerLink (2017). URL: `https://link.springer.com/` (visited on 05/2017) (cited on p. 63).

Tang, Antony, Aldeida Aleti, Janet Burge, and Hans van Vliet (2010). “What makes software design effective?” In: *Design Studies* 31.6, pp. 614–640 (cited on pp. 7–9, 22, 34–36, 38, 51, 70, 135, 147, 156).

Tang, Antony, Muhammad Ali Babar, Ian Gorton, and Jun Han (2006). “A Survey of Architecture Design Rationale”. In: *Journal of Systems and Software* 79.12, pp. 1792–1804 (cited on pp. 5, 8).

Tang, Antony, Paris Avgeriou, Anton Jansen, Rafael Capilla, and Muhammad Ali Babar (2010). “A comparative study of architecture knowledge management tools”. In: *Journal of Systems and Software* 83.3, pp. 352–370 (cited on pp. 5, 27, 60, 177).

Tang, Antony, Yan Jin, and Jun Han (2007). “A rationale-based architecture model for design traceability and reasoning”. In: *Journal of Systems and Software* 80.6, pp. 918–934 (cited on pp. 8, 26, 67–69, 72, 73, 76, 177).

Tang, Antony and Hans van Vliet (2015). “Software Designers Satisfice”. In: *Software Architecture: 9th European Conference, ECSA 2015, Dubrovnik/Cavtat, Croatia, September 7-11, 2015. Proceedings*. Ed. by Danny Weyns, Raffaela Mirandola, and Ivica Crnkovic. Cham: Springer International Publishing, pp. 105–120 (cited on pp. 7, 36).

The International Journal of Software Engineering and Knowledge Engineering (2017). URL: `http://www.worldscientific.com/worldscinet/ijseke` (visited on 05/2017) (cited on p. 63).

Tofan, Dan, Matthias Galster, Paris Avgeriou, and Wes Schuitema (2014). “Past and future of software architectural decisions – A systematic mapping study”. In: *Information and Software Technology* 56.8, pp. 850–872 (cited on pp. 58, 59, 177).

Tversky, Amos and Daniel Kahneman (1974). “Judgment under uncertainty: Heuristics and biases”. In: *Science* 185.4157, pp. 1124–1131 (cited on pp. 22, 23).

Tyree, Jeff and Art Akerman (2005). “Architecture Decisions: Demystifying Architecture”. In: *IEEE Software* 22.2, pp. 19–27 (cited on pp. 4, 8, 27, 66, 67, 72–75, 177).

UNICASE Project (2016). URL: `http://unicase.org/` (visited on 11/2016) (cited on pp. 29, 112).

Vliet, Hans van and Antony Tang (2012). “Design Strategy and Software Design Effectiveness”. In: *IEEE Software* 29.1, pp. 51–55 (cited on p. 136).

Wang, Wei and Janet E. Burge (2010). “Using rationale to support pattern-based architectural design”. In: *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK’10)*. ACM Press, pp. 1–8 (cited on pp. 72, 177).

Weinreich, Rainer and Iris Groher (2016). “Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review”. In: *Information and Software Technology* 80, pp. 265–286 (cited on pp. 59, 177).

Wieringa, Roel (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer (cited on pp. 9–11).

Zaman, Shahed, Bram Adams, and Ahmed E. Hassan (2011). “Security Versus Performance Bugs: A Case Study on Firefox”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*. ACM, pp. 93–102 (cited on p. 54).

Zannier, Carmen, Mike Chiasson, and Frank Maurer (2007). “A model of design decision making based on empirical results of interviews with software designers”. In: *Information and Software Technology* 49.6, pp. 637–653 (cited on pp. 4, 6, 7, 17, 18, 21–23, 26, 28, 34–40, 51, 68, 70, 147, 171).

Zannier, Carmen and Frank Maurer (2006). “Foundations of Agile Decision Making from Agile Mentors and Developers”. English. In: *Extreme Programming and Agile Processes in Software Engineering*. Ed. by Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi. Vol. 4044. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 11–20 (cited on pp. 6, 36).

Zimmermann, Olaf, Thomas Gschwind, Jochen Küster, Frank Leymann, and Nelly Schuster (2007). “Reusable Architectural Decision Models for Enterprise Application Development”. English. In: *Software Architectures, Components, and Applications*. Ed. by Sven Overhage, Clemens A. Szyperski, Ralf Reussner, and Judith A. Stafford. Vol. 4880. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 15–32 (cited on pp. 67, 69, 70, 72, 73, 76, 175, 177).

Zimmermann, Olaf, Jana Koehler, Frank Leymann, Ronny Polley, and Nelly Schuster (2009). “Managing architectural decision models with dependency relations, integrity constraints, and production rules”. In: *Journal of Systems and Software* 82.8, pp. 1249–1267 (cited on pp. 75, 175, 177).

Errata in this Thesis

Chapter 2.2.

Such tools may be compilers, debuggers, editors for textual and graphical artifacts, or interactive development environments (IDEs)~~ [Sommerville 2010]~~.

This thesis aims to support decision documentation during the development activities *requirements engineering*, *design*, and *implementation* in general. These activities are fundamental to both, classic development process models, such as the waterfall model, as well as incremental process models, like agile methods ~~SCRUM [Sommerville 2010]~~.

For instance, such models can be created in use case diagrams using the *Unified Modeling Language* (abbreviated as UML)~~ [Sommerville 2010]~~.

First, developers create a coarse-grained *architectural design* to identify the overall system structures with principal components and their respective relationships~~ [Sommerville 2010]~~. Second, a more fine-grained design details the interfaces, components and data structures for each principal component~~ [Sommerville 2010]~~.

Developers are concerned with the *implementation* of the software when they create code to realize the system design~~ [Sommerville 2010]~~. This activity highly depends on the personal experience and preferences of developers, so there is no common or general process for implementation ~~[Sommerville2010]~~. For instance, developers may start implementing those components first which are best understood. Other developers might start with coding ~~unfamiliar objects~~ the difficult components~~,~~ because they can better estimate the effort necessary for the simple~~known~~ components~~ [Sommerville 2010]~~.

Chapter 2.4.

Management of Decision Knowledge

Documenting decision knowledge is an important part of decision knowledge management. Decision knowledge management was first described as software engineering activity in [Bruegge,Dutoit 2009]. Besides of documentation it comprises communication, modeling and conflict resolution. Typically, software tools are required to enable and support approaches for decision knowledge management in general, and for documentation of decisions in particular [Arnott and Pervan 2008].

Bernd Bruegge, Allen Dutoit: *Object-Oriented Software Engineering: Using UML, Patterns and Java (3rd edition)*, Prentice Hall, 2009