Electronic version of an article published in **Opdahl, A. L.; Pohl, K.; Rossi, M. (Hrsg): Proceedings of the Sixth International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ'00), Reihe: Essener Informatik Beiträge Band 5, pp. 99-112**

Supporting Evolution: Using Rationale in Use Case Driven Software Development

Allen H. Dutoit[1] and Barbara Paech[2]

[1]Technische Universität München, Institut für Informatik, D-80290 Munich, Germany
dutoit@in.tum.de

[2]Fraunhofer Institute for Experimental Software Engineering, D-67661 Kaiserslautern, Germany
paech@iese.fhg.de

Abstract. The requirements specification–as outcome of the requirements engineering process–falls short of capturing other useful information generated during this process, such as the justification for selected requirements, trade-offs made by stakeholders, and alternative requirements that were discarded. In the context of evolving systems and distributed development, this information is essential. Rationale methods focus on capturing and structuring this missing information. In this paper, we propose an integrated process for capturing requirements and their rationale, discuss its tool support, and describe planned experiments to evaluate this process. Although the idea of integrating rationale methods with requirements engineering is not new, few research projects have focused on the use of rationale during later phases to support the evolution of the system under development.

1 Introduction

The *requirements engineering* process aims at eliciting, negotiating, specifying, validating, and managing the requirements of a system under construction. The main product of this process is the requirements specification, which contains precise descriptions of the system and of its environment. The requirements specification is then used as an input for the design, implementation, and testing processes. It aims to be complete, correct, and clear for facilitating the construction of the "right" system. However, the requirements specification does not contain other useful knowledge generated during the requirements engineering process, such as the justification for the selected requirements, the trade-offs made by stakeholders, and the alternatives that were discarded.

Rationale[1] methods aim at capturing, representing, and maintaining records about why developers have made the decisions they have [4]. Rationale includes the problems developers encountered, the options they investigated, the criteria they selected to evaluate options, and, most important, the debate that lead to making decisions. Rationale can serve two different purposes: support negotiation and capture additional information. By making explicit the main decision making elements, rationale facilitates negotiation among stakeholders by systematically clarifying the possible options and their evaluation against well-defined criteria. By capturing rationale, stakeholders can later

1. Historically, much research about rationale focuses on design and, hence, the term *design rationale* is most often used in the literature. Instead, we use the term *rationale* to avoid confusion and to emphasize that rationale models can be used during all phases of development, including requirements engineering.

examine the justification of certain decisions, for example, when revising the system as a consequence of evolving requirements.

The application of rationale methods to requirements engineering or, more generally, to software development, is not new. In requirements engineering, several research projects have studied the use of rationale methods to improve elicitation [21], support scenario analysis [18][19], and support negotiation and improve shared understanding [2]. In software engineering and in engineering design, general methods have been proposed to capture rationale as a graph of issues [6][14]. Many research efforts have focused on supporting negotiation, decision making, and the capture of rationale. It has generally been assumed that capturing rationale is beneficial to the later phases of development, however, few have investigated the actual use of rationale information, for example, when revising requirements decisions.

In this paper, we propose a process for capturing and maintaining rationale in use case driven software development. Our goal is to capture rationale for supporting the evolution of the system and its requirements. Capturing rationale during requirements is attractive given that requirements errors and requirements changes are the most costly during development. Moreover, attaching rationale with requirements information, in particular with use cases, can have a high impact on all phases of development given that use cases are used throughout development. We are interested in investigating the following questions:

1. How should rationale be captured during requirements?
2. How is rationale of requirements used downstream?
3. Which subsets of this information are useful when revising the system or its requirements (and which are not)?
4. How should this information be structured and presented to the developer?

We investigate these questions using an iterative and experimental approach. First, we devised a process to answer question 1. Presently, we are developing a tool to support this process. This tool also gives an ad-hoc answer to question 2. The tool will be used in Fall at the Technical University Munich by students developing a large software system during a 4-month project course. It will also allow to monitor the use of the captured rationale in the course in order to answer questions 2 and 3. Based on the experiences made we will improve our answers to questions 1 and 4.

This paper describes our concepts and plans for the experiment. In Sect. 2, we describe the use case driven approach to requirements engineering, which emphasizes clear relationships between user tasks, use cases, scenarios, and system services. In Sect. 3, we describe a general process to capture rationale. As in other methods, we capture and represent rationale as a graph. However, we propose that, in addition, discussions and rationale capture are monitored and guided by a facilitator with the intent of eliciting additional information and improving its structure. We argue that this approach can not only improve the quality of requirements decisions but also the quality of decisions made later, such as during system design or changes to requirements. In Sect. 4, we describe the integration of the processes for capturing requirements and rationale. In Sect. 5, we sketch the use cases for the envisioned tool. In Sect. 6, we describe the experimental context in which we plan to evaluate these processes and their tool support. In Sect. 7, we conclude with a discussion of related work and future research directions.

2 Capturing Requirements as Use Cases

Use cases are a popular addition to object-oriented software development. They have first been proposed by Jacobson [9] and are now part of the (Rational) *Unified Software Development Process* [10]. However, there is no single accepted definition of use case. Two major issues are the relationships of use cases to goals and to scenarios [5][22]. In the following, we propose our definitions for these concepts and we describe their relationship with our proposed requirements engineering process.

2.1 Concepts

Actors are external entities that interact with the system. Examples of actors include a user role (e.g., a bank customer) or another system (e.g., a central database).

A *Scenario* is a concrete sequence of interactions between an actor and the system. Because of their exemplary nature, scenarios are particularly useful for the elicitation of requirements and often serve as a blueprint for a use case. They can also be used as test cases for use cases.

A *Use Case* is a general sequence of interactions between an external actor and the system. A use case, thus, describes a collection of scenarios. Use cases are used to capture the functional requirements of the system. In contrast, functional requirements in structured methods are captured as *System Services*. Use cases capture significantly more information as they also describe the context surrounding the system services, such as the users' work processes and their physical environment.

A *User Task* is a unit of work that is meaningful to the user. It is part of the environment in which the system operates, often a step in an encompassing business process. Only by knowing the user tasks in detail a system with maximal support to the user can be designed [8]. A use case describes how a user task can be achieved through a sequence of interactions with the system. Thus, the user tasks make the functional and nonfunctional goals of the users in the use case explicit.

Fig. 1 depicts the relationship between User Task, Use Case, and System Service. A user task describes environment specific phenomena independently of the system. For example, "Withdrawing Money from a Bank Account" is a user task. A system service describes a system specific phenomenon independently of a user task. For example, "Authenticating with the Automated Teller Machine (ATM)" is a system service. A use case is a sequence of interactions between a user and the system whose purpose is to accomplish a specific user task. For example, the "Withdraw Money from ATM" use case would describe the general sequence of services a user needs to invoke to withdraw money from a bank account using an ATM.

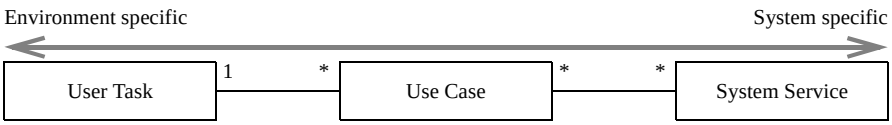

Fig. 1. Relationship between User Task, Use Case, and System Service (UML class diagram).

Nonfunctional constraints are constraints on the use cases, the system services, or the system. "The user should be able to withdraw a sum of money within a minute of authenticating with the ATM" is a performance constraint on the maximum duration of the

"Withdraw Money from ATM" use case. "Every command should provide feedback to the user within 1 second" is a constraint on all system services. "The ATM should have a 95% availability" is a reliability constraint on the system. For the purpose of tool support, all *Requirements Elements*–namely use cases, scenarios, system services, nonfunctional constraints and glossary entries–are stored in an *Option Base*. The *Requirements Specification* includes all the use cases, definition of system services, and nonfunctional constraints that are necessary to describe a system completely.

2.2 Processes

The input of our requirements engineering process is a *Problem Statement*, written by the client and the requirements engineers, describing the user tasks that the system should support. The problem statement serves two purposes: First, it provides an initial description of the environment of the system (e.g., a set of actors, business processes, and user tasks); second, it establishes the scope of the work supported by the system (i.e., which business processes and user tasks should be supported and which should not). We are well aware that producing an adequate problem statement requires a requirements process in itself. However, here we concentrate on the specification of requirements to be used as input to software design.

The requirements engineering process we describe below is iterative and incremental. The requirements engineers may decide to write and refine only a limited set of use cases at the time (i.e., a depth first approach), or, conversely, work concurrently all use cases (i.e., a breadth first approach). Each iteration, however, is composed of the activities depicted in Fig. 2. Note that this use case diagram describes the steps of the requirements engineering process and, at the same time, the use cases for the tool support we envision.[2] Note also that we distinguish in the diagram user tasks from use cases by labelling them UT and UC, respectively.

We stipulate the requirements activities to be carried out within the following steps:

1. *Create Use Case*. This step develops an initial draft of one use case for each user task which determines which parts of the user tasks are realized by the system and which are realized by the user. The initial use cases are high level and usually do not focus on system services or nonfunctional constraints.
2. *Create Example Scenario*. This step develops example scenarios for each use case and ensures that each high-level use case appropriately addresses each user task. The development of scenarios also initiates discussions about how the functionality provided by the system should be organized into system services.
3. *Create System Service*. This step refines each use case in terms of system services and defines the system services more precisely to find a correspondence between each interaction and each system service.
4. *Define Constraint*. This step identifies and describes nonfunctional constraints for each use case. These constraints describe properties that the system must have in order to be useful to the user. This step may also result in nonfunctional constraints that are applicable to the complete system.

2. The "Create User Task" use case is not part of our process, but necessary to input the problem statement into the tool.

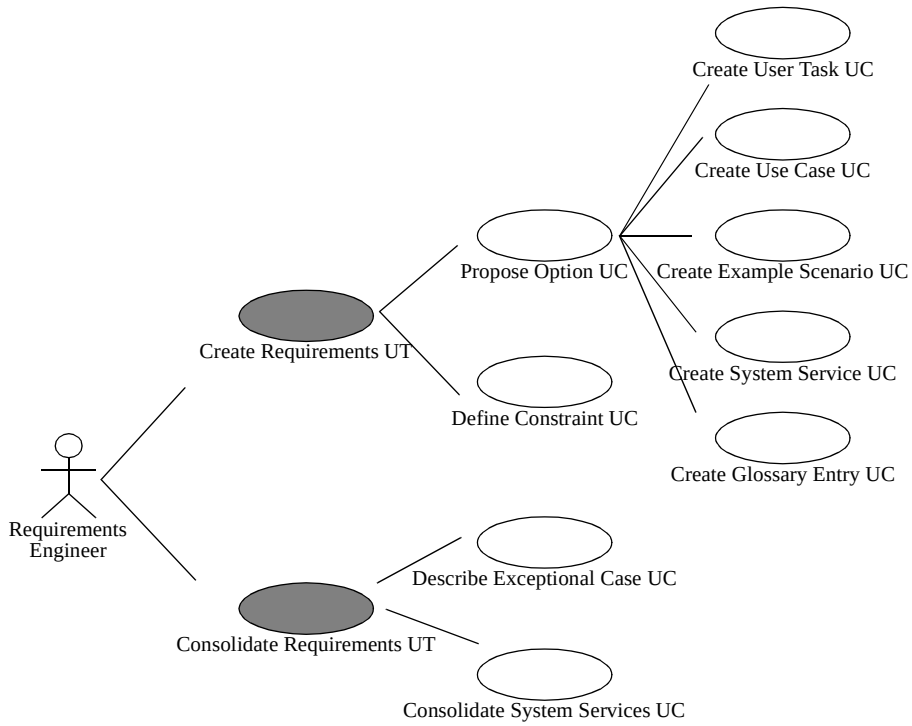

Fig. 2. The requirements capture process (UML use case diagram, gray ovals depict user tasks).

5. *Describe Exceptional Case*. This step describes the response of the system under error conditions, such as wrong user input or component failure. Exceptional cases are also described as flow of events but are separated from common cases for clarity.
6. *Create Glossary Entry*. All terminology specific to the use case is captured in a glossary. This includes terminology specific to the user tasks as well as terminology specific to the system services described in the use cases.
7. *Consolidate System Services*. This step finds redundancies among use cases. Similar services are consolidated into a single service. This results into a simpler and more consistent system.

The first five steps built on one another, while the last two steps are concurrent to all the other ones. Fig. 7 in Sect. 5 shows an example of a consolidated use case. Preconditions include actor visible constraints that are necessary for the successful execution of the use case as well as invisible constraints that are specific to the system state. The exit conditions describe the outcome of successful use case execution, including actor visible conditions as well as conditions on system state. The flow of events captures the normal use case interaction as well as references to possible exceptions. The interactions during exceptions are described separately and include an exit condition describing the outcome of the exception handling.

Requirements engineering is fundamentally iterative. Each of the seven steps we describe is executed several times on the same subset of use cases. While executing one of these steps, questions can be raised about other use cases or other steps. Describing a

scenario can trigger questions about its corresponding use case, which is then changed. When raising, discussing, and answering questions, requirements engineers generate rationale information that we are interested in capturing and using downstream. In the next section, we describe the processes aimed at capturing and structuring this information.

3 Capturing Rationale as an Issue Model

Rationale is the justification of decisions [3]. Rationale methods aim at capturing the justification of decisions, first, to improve their quality and, second, to record the reasoning that went into them for the event when they are revised. Argumentation-based rationale is an approach that represents rationale as a graph of rhetorical steps, also called an *issue model*. Many different models have been proposed, including IBIS [11] and QOC, [14] to name a few. We first describe our model, which is a refinement of the QOC model. We then describe the processes of capturing and maintaining rationale.

3.1 Concepts

To each decision corresponds a *Question* that needs to be solved for the requirements process to proceed. Questions can indicate a problem with the proposed system (e.g., "Requiring the user to input manually: Is there a simpler way to authenticate the user?"), a problem with the domain description (e.g., "Are bank customers allowed overdraw their accounts?"), or a clarification (e.g., "What are all the conditions a bank customer needs to meet before using an ATM?").

Options are possible solutions that could address the question under consideration. These include options that were explored but discarded because they did not satisfy one or more constraints. For example, a biometric sensor for reading finger prints is currently too expensive for an ATM. Similar options can be grouped into a main option and a number of refined options.

Criteria are desirable qualities that the selected option should satisfy. For example: "The cost of an individual ATM should be minimized given that a bank can have many ATMs." Or: "The security of the system needs to be reasonably high such that the cost of fraud does not outweigh the benefits of providing an ATM service." Criteria are essentially high-level nonfunctional constraints. Hereafter, we refer to criteria as nonfunctional constraints.

Requirements engineering and software development are not algorithmic. Users and requirements engineers discover questions, try different options, and argue their relative benefits. It is only after much discussion that a consensus is reached or a decision imposed. This argumentation on all aspects of the decision process, including nonfunctional constraints, explored options, and questions is captured as *Argument* nodes which can be attached to any other node in the issue model.

A *Decision* is the resolution of a question representing the selected option. Decisions are already captured in the use cases during requirements engineering. We only need to capture the relationship between decisions and their corresponding rationale. A question that has been closed can be reopened, in which case the decision becomes an obsolete one. Fig. 3 depicts an example issue model for describing possible authentication mechanisms for an ATM.

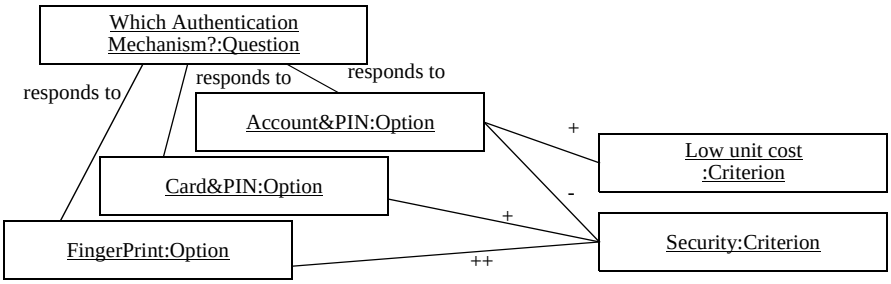

Fig. 3. An example of issue model (UML object diagram). The authentication mechanism of the ATM can be either a magnetic card requiring a PIN, a biometric sensor, or an account number and a PIN that the user needs to memorize. Each option is then evaluated against a set of criteria.

3.2 Processes

The rationale of requirements is captured by two processes. The first process, the capture process executed by a requirements engineer or a reviewer, focuses on capturing rationale, whereas the second process, the maintenance process executed by the rationale maintainer, focuses on consolidating and restructuring the rationale for future use.

The capture process is composed of the following steps (see Fig. 4 where, again, process steps coincide with tool use cases[3]):

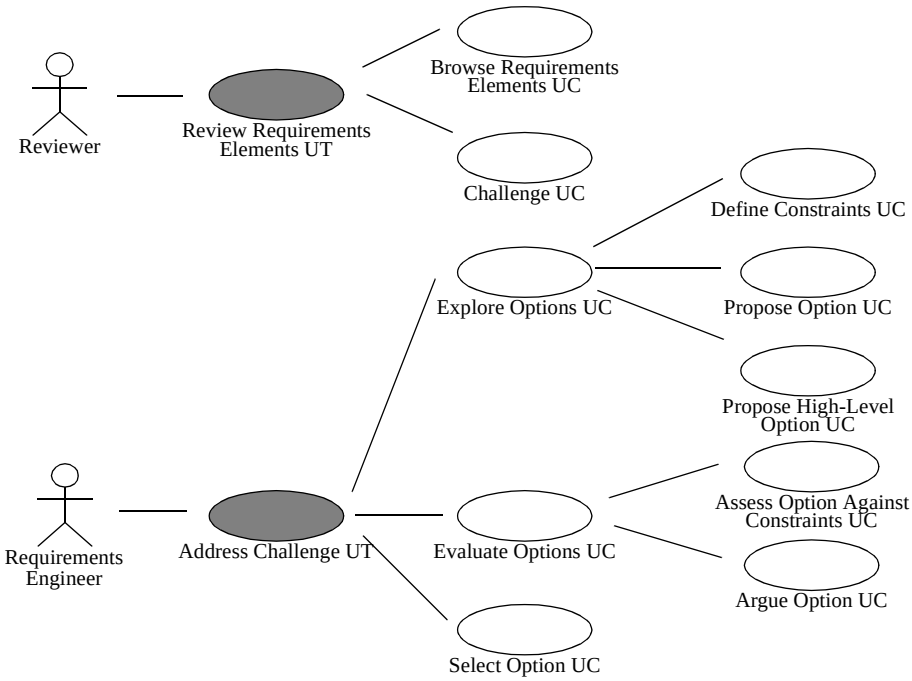

Fig. 4. The rationale capture process (UML use case diagram, user task indicated in gray).

3. The “Browse Requirements Elements” use case is not part of our process but is necessary to access the requirements elements through the tool.

1. *Challenge.* A reviewer reads some part of the requirements specification and challenges problem areas with questions.
2. *Explore Options.* Questions can result in the discussion of possible changes in the requirements specification. A possible option that is always available is the status quo, that is, not to change the requirements. Clarification questions are addressed with options to improve the requirements specification without necessarily resulting in changes to the system. An option can be completely specified by writing out the corresponding use cases *(Propose Option)* or can be simply described as a high-level option *(Propose High-Level Option)*. In both cases, the options should contain enough detail to enable the requirements engineer to evaluate and compare the proposed options *(Define Constraints)*.
3. *Evaluate Options.* Once a sufficient number of options have been proposed, requirements engineers need to evaluate them and refine them to satisfy nonfunctional constraints *(Assess Options Against Constraints)*. During this step, requirements engineers also create arguments supporting and opposing options *(Argue Option)*.
4. *Select Option.* Once requirements engineers have evaluated and refined (most or) all options, requirements engineers create a decision by selecting an option which can result in minor or substantial change in the requirements specification. Note that a clarification question can be resolved without any changes. Note also that addressing a question may invalidate previous options and revisit earlier decisions. We discuss this point in more depth in Sect. 4.

During the capture process, requirements engineers may skip any of the above steps. Options can be generated and evaluated without an explicit question. Decisions can be taken and changes implemented without explicit discussion. It is desirable, however, that at least some of the components of the decision are recorded so that the rationale maintenance process can recover the missing parts.

The capture process can be executed at any time. We anticipate, however, that it will occur when requirements engineers review the requirements specification, either when validating the requirements or in the process of executing a requirements step. The maintenance process, however, is executed by the rationale maintainer whose responsibility is to keep the content and structure of the rationale up to date. The maintenance process is composed of the following steps (see Fig. 5):

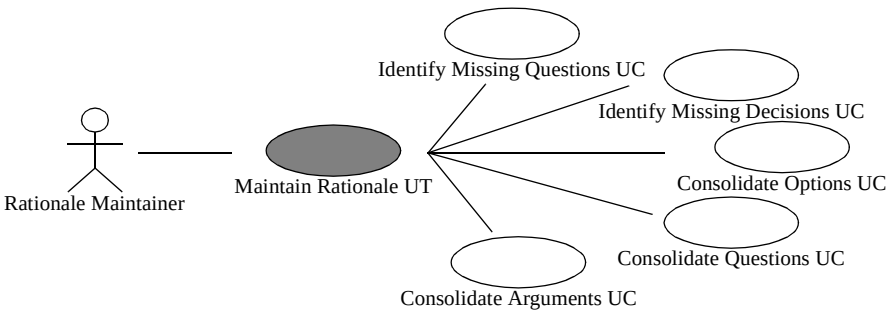

Fig. 5. The rationale maintenance process (UML use case diagram, user task indicated in gray).

1. *Identify Missing Questions*. Given that requirements engineers may skip steps in the capture process, there can be options that were captured without their corresponding question. In most cases, the implicit question can be made explicit using the options.
2. *Identify Missing Decisions*. Most decisions occur during meetings or face-to-face conversations. Consequently, they may be implemented in the requirements specification but not captured in the issue model. The rationale maintainer can identify these decisions by ensuring each change is associated with a decision.
3. *Consolidate Options.* When discussing a question, the requirements engineers may propose similar options. The rationale maintainer consolidates identical options into single nodes and restructures similar options.
4. *Consolidate Questions*. When reviewing requirements elements, reviewers may raise similar questions. The rationale maintainer consolidates identical questions into single nodes and restructures similar options.
5. *Consolidate Arguments.* Arguments often constitute the bulk of rationale information [15]. Arguments are usually unstructured and may apply to several options and decisions. The rationale maintainer summarizes verbose or redundant arguments and adds missing links to relevant rationale nodes.

4 Integrating Requirements and Rationale

In the previous two sections, we described two groups of processes: one for capturing and consolidating requirements and one for capturing and maintaining rationale. The second group of processes represents additional overhead for developers. Capturing and maintaining rationale will yield benefits only if both process groups and their corresponding tool support are integrated. Indeed, the integration of rationale methods and tools with various aspects of development is a fundamental issue that has received too little attention in rationale research [12]. In this section, we describe the concepts and process steps which are related to the integration of requirements and rationale capture.

4.1 Concepts

We identify three areas where additional associations need to be created:

Questions/Requirements associations. The association between a question and the requirements that are challenged needs to be captured. This enables a reviewer to specify which parts of the requirements are challenged and for a requirements engineer to list all questions for a given requirements element.

Option/Requirements associations. An option can be thought of as an aggregate change on the option base. The association between an option (or a high-level option) and the requirements elements that the option proposes, removes, or modifies also needs to be captured. When evaluating an option, this enables the requirements engineer to assess the impact of an option. When understanding the requirements, this allows a reviewer to trace back the source option or question that lead to a specific requirement.

Requirements elements status. Given that requirements engineer can propose new requirements elements as part of an option but that these requirements elements can be discarded in favor of another option, each requirements element in the option base needs to include a status attribute. The requirements status can take three values:

- *current*, if the requirements element is part of the current option,

- *proposed*, if the requirements element is part of an option that has not been selected,
- *discarded*, if the requirements element was part of the current option but has been discarded in favor of another option.

4.2 Processes

To integrate the requirements and rationale processes, we modify the steps *Propose Option*, *Challenge*, and *Select Option*, and introduce three new steps, *Realize High-Level Option*, *Discard Current Requirement*, and *Make Proposed Requirement Current*.

The *Propose Option* step (Sect. 2) sets the initial value of the status attribute of each new requirement to *proposed*. This allows requirements engineers to distinguish between requirements they have just entered with those that are part of the current option.

The *Challenge* step (Sect. 3) creates associations between the question and the requirements being challenged. These associations make explicit relationships between rationale and requirements and allow reviewers and requirements engineers to trace changes to specific problems.

The *Select Option* step (Sect. 3) can have two variations (Fig. 6). Either an option is selected or a high-level option is selected. If a high-level option is selected, it is first realized by creating all proposed use cases and modifying existing use cases. This is accomplished using the *Realize High-Level Option* process step. At the end of this step, a new option is created and linked with the corresponding requirement. The Select Option step then invokes the *Discard Current Requirement* step to change the status of any requirement that needs to be discarded. The *Make Proposed Option Current* step is then invoked to change the status of the proposed requirements to *current*.

The differentiation between high-level options and options enables requirements engineer to debate several options without fully developing them a priori. We anticipate this will encourage the capture of more information about discarded options.

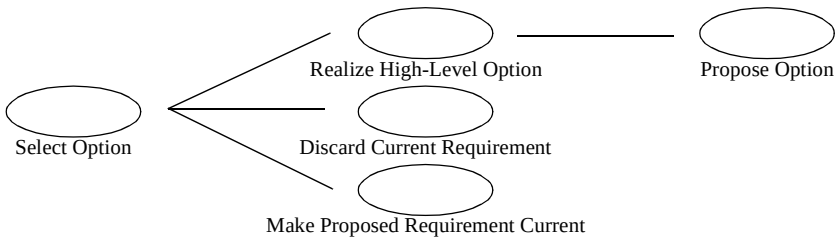

Fig. 6. Refined Select Option use case (UML use case diagram).

5 Tool Support

We have used our own process for identifying the requirements for the tool support for our process. Fig. 7 shows the *Realize High-Level Option* use case as an example for such a tool use case.

We plan to explore two tool options: One is to use and customize a commercial requirements management tool, such as DOORS [7], the other is to implement a prototype Web application. The advantage of the requirements management tool is that it supports well the linkage of requirements with rationale elements and the browsing of both with all sorts of filter functions. This, in particular, supports rationale maintenance and inves-

Use Case Name	Realize High-Level Option
Actor	Requirements Engineer
User Task	Address Challenge
Precondition	The requirements engineer is authenticated with the system. The requirements engineer is in the process of selecting this high-level option as the current option to address a question.
Exit Condition	A new option is created in the option base, including new requirements elements, and possibly revising or removing older requirements elements. New arguments may be added to the option base. Associations between the new option, the new arguments, and the modified requirements elements are created.
Flow of Events	The requirements engineer selects a high-level option. The requirements engineer first reviews the list of new use cases to be added to the system, use cases to be revised, and old use cases to be removed, using the Browse Requirements Elements use case. The requirements engineer then creates a new option based on the high-level option using the Propose Option use case. The requirements engineer revises any new or existing use cases, as needed. The requirements engineer revises or creates additional scenarios, glossary entries and system services, associated with the modified use cases, as needed. The requirements engineer can create and associate arguments with any of the modified requirements elements to justify their existence. Once the requirements engineer indicates to the system that the high-level option has been realized, the system checks whether all elements of the high-level option have been dealt with and creates associations between the new option in the option base and the modified requirements elements.
Nonfunctional Constraints	None
Exceptions	The requirements engineer exits the tool before indicating to the system that the high-level option has been realized. The system offers the requirements engineer to undo his changes or to complete the high-level option. The requirements engineer fails to create a new use case specified in the high-level option. The system prompts the requirements engineer whether to remove the use case from the high-level option or to create it.

Fig. 7. Example of consolidated use case for the envisioned tool support.

tigation of the option base. On the other hand these tools are quite comprehensive and therefore require substantial effort to learn and use efficiently, which is a drawback when conducting experiments with students.

The advantages of a prototype Web application are that it can have a simple and intuitive interface for users familiar with Web browsers and can support a number of concurrent collaborating users. The Web application also makes it easier to instrument and capture the data for evaluation, such as accesses to specific elements of the requirements or the rationale. The disadvantages of a Web application is that it provides less functionality than a commercial tool.

We plan to explore both approaches, the commercial tool for the purpose of refining and improving our process with individual experienced developers, the Web prototype for the purpose of conducting experiments with groups of students.

6 Experimental Context

In this section, we describe the goals, research questions, and hypotheses of the planned experiment. The goal and the research questions can be summarized according to the Goal Question Metric (GQM) paradigm [1]. The quality of entries will be judged ac-

Analyze	tool usage
for the purpose	of understanding rationale usage
with respect to	1. number, quality, and sequence of requirements elements created during requirements specification 2. number, quality, and sequence of rationale elements (e.g., questions, options, decisions) created during requirements specification by either a requirements engineer or a maintainer 3. number, quality, and sequence of the associations between requirements and rationale created during requirements specification 4. number of status changes of requirements elements during requirements specification and design 5. number of accesses to the different requirements elements and rationale elements during design 6. number of accesses to the different rationale elements during requirements change
from the viewpoint of	the researcher
in the context of	a 4-month software development course at the Technical University Munich.

cording to adherence to the given templates and the subjective judgements of the authors of this paper. We list below the hypotheses regarding the above questions:

1. We expect the number of use cases to be less than the numbers of the other requirements elements, since one use case gives rise to several of the other elements. We expect the quality of the scenarios to be higher than the quality of the other elements, since typically it is easier to create exemplary information than to create abstractions like use cases, system services, or glossary entries. Regarding the sequence, we expected adherence to our process, since the subjects are students.
2. We expect most rationale elements created by a requirements engineer to be questions and decisions, because we expect most questions are usually request for clarification and non contentious, as in other research projects [2]. We expect options and high-level options to be created only in the context of contentious questions. We expect the quality of rationale elements created by the rationale maintainer to be superior than the rationale elements created by the requirements engineer.
3. We expect that most associations between requirements elements and rationale will be generated by the tool and few additional associations will be entered by the rationale maintainer, as these associations are often already implicitly contained in the description of the rationale elements.

4. We expect more status changes during design than during requirements specification, since the students do not have access to the users. Thus, they have a lot of freedom in fixing the requirements, but - being students - they will lack experience on how easy it is to realize the requirements.
5. We expect more accesses to use cases than on the other requirements elements, since use cases capture more context than the other entries.
6. We expect more accesses to arguments than to the other rationale elements, since they are most important to understand decisions.

By validating or rejecting the above hypotheses, we will have deepened our understanding of the creation and use of rationale during use case driven software development. In addition, we will identify shortcomings in the tool support and the requirements process.

7 Conclusion and Related Work

We have proposed an integrated process for capturing requirements as use cases and capturing the rationale for decisions taken in the requirements specification. The emphasis of our approach is on support for documenting knowledge from the requirements engineering process as much as is possible and helpful for use in dependent development processes like software design or testing and for requirements change. Another focus is to make rationale capture and usage for developers as easy as possible through provision of a tool and a defined process, as well as by stipulating support through a rationale maintainer.

Through these two foci we complement the existing research on integration of rationale within requirements engineering:

Inquiry-based requirements analysis is a method for incrementally refining a requirements specification and capturing requirements discussion [18]. *ScenIC* [19] is an instantiation of this method. It aims mainly at improving the quality of the requirements documents by supporting semantic, episodic, and working memory for project attention management. Thus, ScenIC supports the requirements elicitation process through an explicit notion of goals and episodes, while our approach supports requirements documentation and the interface to dependent software development processes through a clear distinction between user tasks, use cases and system services. This follows the general principle of combining goal based and object/activity based approaches put forward in [16]. The focus on tasks and nonfunctional requirements distinguishes also our approach from other approaches to contextualize use cases through goals as for example, in [5], [13], and [17].

The *Theory-W based spiral approach*, supported by the *WinWin* tool, aims at supporting requirements negotiation with a computer tool tracking each stakeholder's "Win" conditions and their resolution [2]. Experimental validation of the Theory W model shows that the use of an issue model for negotiation support enhances trust and shared understanding among shareholders, even in the presence of uncertainties and changing requirements. In our approach, we attempt to generalize these results to the other aspects of software development.

SCRAM [21] uses rationale to improve the elicitation and validation of requirements with users. Their focus is to improve the quality of requirements by eliciting more in-

formation and more kinds of information by making the requirements rationale visible to the users. In an earlier study [20], Sutcliffe observed that the availability of rationale information lead users to ask more questions and more open ended questions during elicitation sessions. We stipulate that the availability of rationale information and their association to use cases will lead to similar benefits among developers during later phases of software development.

References

[1] V.R. Basili, G. Caldiera, & H.D. Rombach, "Goal Question Metric Paradigm", In J.J. Marciniak (ed.), *Encyclopedia of Software Engineering*, vol.1, pp.528–532, John Wiley & Sons, 1994.

[2] B.Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, & R. Madachy, "Using the WinWin Spiral Model: A Case Study," in *IEEE Computer*, pp. 33–44, July 1998.

[3] B. Bruegge & A.H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 1999.

[4] S. Buckingham Shum & N. Hammond, "Argumentation-based design rationale: what use at what cost?" *International Journal of Human-Computer Studies*, vol. 40, pp. 603–652, 1994.

[5] A. Cockburn, "Goals and Use Cases", *Journal of Object-Oriented Programming*, vol. 10, no.5, pp. 35–40, 1997.

[6] J. Conklin & K. C. Burgess-Yakemovic, "A process-oriented approach to design rationale," *Human-Computer Interaction*, vol. 6, pp. 357–391, 1991.

[7] QSS, http://www.qssinc.com.

[8] "Benutzer-orientierte Gestaltung interaktiver Systeme", Normentwurf, DIN EN ISO 13407, 1998

[9] I. Jacobson, M. Christerson, P. Jonsson, & G. Overgaard, *Object-Oriented Software Engineering—A Use Case Driven Approach*. Addison-Wesley, Reading, MA, 1992.

[10] I. Jacobson, G. Booch, & J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999.

[11] W. Kunz & H. Rittel, "Issues as elements of information systems," *Working Paper No. 131*, Institut für Grundlagen der Plannung, Universität Stuttgart, Germany, 1970.

[12] J. Lee, "Design Rationale Systems: Understanding The Issues," in *IEEE Expert*, pp. 78–85, May/June 1997.

[13] J.C.S. do Prado Leite, G. Rossi, F. Balaguer, A. Maiorana, G. Kaplan, G. Hadad & A. Oliveros, "Enhancing a Requirements Baseline with Scenarios", *International Symposium in Requirements Engineering, RE'97*, pp. 44–53, 1997.

[14] A. MacLean, R. M. Young, V. Bellotti, & T. Moran, "Questions, options, and criteria: Elements of design space analysis," *Human-Computer Interaction*, vol. 6, pp. 201–250, 1991.

[15] T. P. Moran & J. M. Carroll (eds.), *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates, Mahwah, NJ, 1996.

[16] J. Mylopoulos, L. Chung, & E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis", *Communication of the ACM*, vol. 42, pp. 31–37, 1999.

[17] K. Pohl & P. Haumer, "Modelling Contextual Information about Scenarios", *International Workshop on Requirements Engineering: Foundations of Software Quality, REFSQ'97*, pp. 197–204, 1997.

[18] C. Potts, K. Takahashi, & A. I. Anton, "Inquiry-based requirements analysis," *IEEE Software*, vol. 11, no. 2, pp. 21–32, 1994.

[19] C. Potts, "ScenIC: A Strategy for Inquiry-Driven Requirements Determination", *International Symposium on Requirements Engineering, RE'99*, pp. 58–65,1999.

[20] A. Sutcliffe, "Requirements Rationales: Integrating Approaches to Requirement Analysis," In Olson G.M., Schuon S, (eds.) *Proc. of Designing Interactive Systems, DIS' 95*, pp. 33–42, ACM Press, New York, 1995.

[21] A. Sutcliffe & M. Ryan, "Experience with SCRAM, a SCenario Requirements Analysis Method," In *Proc. of the 3rd International Conference on Requirements Engineering*, pp. 164–171, April 1998.

[22] C. Rolland, G. Grosz, & R. Kla, "Experience with Goal-Scenario Coupling in Requirements Engineering", *Proc. of International Symposium on Requirements Engineering, RE'99*, pp.74–81, 1999.