11 Rationale and Requirements Engineering

Many of the decisions that have the greatest impact on the software development process are made during requirements analysis. Software Engineering Rationale (SER) can support this process by providing the ability to capture the decisions and reasons behind them, starting at these earliest phases. SER also supports requirements traceability throughout the process by directly mapping the development options chosen to the requirements that provide their rationale and by providing rationale for the requirements, thereby mapping requirements back to their source. In this chapter, we describe how rationale can support requirements engineering.

11.1 Introduction

11.1.1 Requirements Engineering

The key to every successful software project is its ability to meet the needs of its intended customer. This means that the software developers must determine what the requirements are for the software system. The process of identifying requirements, analyzing them to obtain additional requirements, documenting them in a specification, and validating that specification to ensure that it meets user needs is known as requirements engineering (Saiedian and Dale 2000). In provisioned systems (systems developed under contract), the requirement specification serves as the basis for the development contract; in product development, requirements are written based on market analysis and are expected to change if necessary (Kuusela and Savolainen 2000).

Inadequate or deficient software requirements are considered the leading cause of project failure (Alford and Lawson 1979; Hofmann and Lehner 2001). Lindquist (2005) states that analysts report the percentage of project failures resulting from poor requirements management to be greater than 70%. The management problem is especially difficult on systems where the requirements are not stable. It is well known that the later in the

development process a requirement changes the higher the cost to make the change will be. Agile development methodologies, such as Extreme Programming (Beck 1999), have been created to "flatten the curve" and be more responsive to changing requirements.

Requirements are typically broken into two categories: functional requirements that describe what the system should do (functions performed or features implemented) and nonfunctional requirements (NFRs) that describe qualities that the developed system should have. NFRs are often referred to as "ilities" (Filman 1998) since NFRs include qualities such as usability, scalablity, reusability, testability, maintainability, etc. Nonfunctional requirements are difficult to test and verify because they tend to cross-cut functionality of the system and also because they are often difficult to quantify. While they do not describe the functionality desired by the stakeholders they do have a direct impact on how satisfied the stakeholders are likely to be with the final product. Some NFRs involve the development process. Examples of these would be affordability, maintainability, and flexibility.

11.1.2 Objectives of This Chapter

This chapter discusses some of the key areas of requirements engineering (RE) and how they can be supported by the capture and use of rationale. In particular, the chapter focuses on obtaining requirements, requirements traceablity, approaches using nonfunctional requirements, goal-based requirements engineering and how rationale can support requirements change.

11.2 Obtaining Requirements

11.2.1 Requirements Elicitation

The first challenge faced in RE is the difficult task of eliciting requirements from the system stakeholders. Stakeholders are typically referred to as being anyone who is involved in the project or "whose interest the project affects" (Hoffman and Lehner 2001). This is a very broad category and can include the users, developers, marketers, procurers, QA, and any others who might be affected by the use of the system. Sharp et al. (1999) identify four groups of "baseline" stakeholders: users (those who interact with or control the

software and those who use products of the system), developers, legislators (anyone providing guidelines for operation), and decision-makers (managers and finance people in both the developer and user organizations).

After stakeholders are identified, the next challenge is obtaining the requirements. There are many challenges encountered in this process, including stakeholders having difficulty expressing what they want or making technically unrealistic demands; stakeholders describing requirements in the language of their domain, which may not be familiar to the analyst; conflicts in stakeholder requirements; political factors affecting requirements; and the possibility of the business environment changing (Sommerville 2007). The requirements specification can be viewed as a "wish list" for the different groups of stakeholders where the requirements rising from different stakeholder views may be inconsistent or contradictory (Kuusela and Savolainen 2000).

Requirements can be obtained using many methods. These include structured or unstructured interviews, observing the system in use (if the new system replaces an existing one), rapid prototyping to get user feedback, and collaborative approaches such as Joint Application Development (JAD) (Bruegge and Dutoit 2000). The stakeholders may have a difficult time articulating their requirements. The more expert a user is at performing a task, the higher the chance that they will be performing at least parts of it "automatically," making it more difficult for them to describe those steps to another person. This necessitates a combination of direct and indirect elicitation techniques where direct techniques are used to obtain information that can easily be expressed verbally and indirect techniques are used to obtain information that cannot be easily expressed verbally (Hudlicka 1997).

One thing that is typically not done during requirements elicitation is capturing the rationale behind the requirements. There may be many system features identified for potential incorporation into a software system. The rationale can capture the tradeoffs between these features along with the consequences, both desirable and undesirable, of incorporating or not incorporating each of them (Carroll et al. 1998).

The rationale would also be a logical place to capture the source of the requirement. Knowing which stakeholders, and which stakeholder category they fit into, would be useful if questions arise about the requirement that require clarification. Knowing the requirement source would assist in the prioritization of the requirement by identifying the interested parties. Rationale can also associate requirements identified and refined during the requirements engineering process with the original customer requirements and provide "rich traceability" (Dick 2005; Hull et al. 2002). The rationale would also provide the intent behind the requirement, or the stakeholder

goal(s) that the requirement addresses. The mapping of goals to requirements can be used later to determine which requirements would require adjustment if the goals change later in the development process. The rationale would also be a place where dependencies or conflicts between requirements can be identified. Knowing the source and intent of each conflict will be useful when determining the best way to resolve conflict and inconsistency.

11.2.2 Achieving Consensus

An important part of the RE process is the negotiation that needs to take place between the various stakeholders. The different groups approach the system from different viewpoints and may have conflicting goals. The rationale for the requirements is a key element in the negotiation process by providing a means for identifying conflicts and explicitly stating the arguments of all participants. The collection process itself was found to be useful during field trials using itIBIS and gIBIS (the textual and graphical versions of the Issue-Based Information Systems approach) (Conklin and Burgess-Yakemovic 1995). Structured rationale capture assisted with team communication by making meetings more productive. The Compendium approach (Shum et al. 2006) is an IBIS-based collaboration support system that is used to capture stakeholder needs via a "dialogue map" that aids in collaboration by structuring discussion and capturing the "meanings and ideas" of the group.

The role of rationale in requirements negotiation is a key element in the WinWin approach to requirements negotiation (Boehm and Kitapci 2006; Boehm and Bose 1994). An ontology defining the rationale in WinWin was developed by Bose (1995) and describes what the attributes are for the WinWin rationale elements (Winconditions, Options, Issues, and Agreements). The goal of the WinWin approach is to make "winners" of the system's stakeholders. The EasyWinWin tool assists in group facilitation to aid in determining what the win conditions are, prioritizing the win conditions, identifying what the issues are, and capturing the decision rationale (Grünbacher and Boehm 2001). Experiments performed using students demonstrated that the WinWin approach assisted with distributed collaboration, aided in cooperation, reduced friction between team mates, and helped the students to focus on the key issues (Boehm and Egyed 1998). There are more than 100 real-world projects that have used EasyWinWin (Boehm and Kitapci 2006).

An alternative approach to requirements negotiation and validation is the Software Quality Function Deployment (SQFD) approach (Ramires et al. 2005). SQFD builds a matrix that gives correlation values between specifications and requirements where the stakeholders provide the correlation values. The MEG groupware tool was built to support SQFD and added rationale, in an adapted IBIS format, to the SQFD matrix. The IBIS component captures stakeholder positions and arguments. The evaluation of the requirements is achieved using a majority voting scheme where votes are weighted depending on how each stakeholder participated in past decisions (Win-Win, Win-Lose, or Lose-Lose).

11.2.3 Requirements Inconsistency

Since requirements are obtained from a variety of stakeholders and sources, there is a risk that inconsistencies may arise. It is important to identify inconsistencies so they can be handled appropriately, whether through resolution, avoidance, deterring, or ignoring (Nuseibeh et al. 2000). There are a number of approaches to performing consistency checking in requirements. The C-Re-CS system (Klein 1997b) captures requirements and their rationale in a semantic net structure. The system contains exception management services that check for completeness, correctness, and consistency in the requirements; identify problem diagnosis using a knowledge base of general requirements problems; and identify potential resolutions to the problems based on past knowledge of general problems. The knowledge base is structured as a taxonomy of diagnoses from more general to more specific that is traversed similarly to a decision tree based on questions and answers.

Reiss (2002) has developed a constraint-based, semiautomatic maintenance support system that works on the abstracted code, code, design artifacts, or metadata to assist with maintaining consistency between artifacts. The CLIME software development environment checks for consistency between UML class diagrams and source code; between UML interaction diagrams and code; test cases to source code (to ensure unit tests have been run if a method was modified); documentation to source code; source code to documentation; and also checks code and documentation to ensure that certain preset standards (such as naming conventions) are followed (Reiss et al. 2003).

11.2.4 Requirements Prioritization

Recent work on Value-Based Software Engineering has begun to address the problem that software development efforts treat each requirement (and other development artifacts) as if they were of equal value (Boehm 2006b). In reality, some requirements are more important to the stakeholders than others. When decisions need to be made to decide what requirements should be implemented first or should be given the most resources, it would make sense to base these decisions on the relative value of the requirements and prioritize them.

Karlsson and Ryan (1997) propose using a cost-value approach to prioritize software requirements. This method uses the Analytic Hierarchy Process (AHP) (Saaty 1990) to perform pairwise comparisons of the requirements. Customers and users use AHP to provide relative value and software engineers use AHP to provide relative cost. This is an effective method for determining priorities but does have scalability issues for large numbers of requirements.

Rationale can also be used to assist with the requirements prioritization process. The rationale behind each requirement can capture the underlying intent behind the requirement. The Software Engineering Using RATionale (SEURAT) tool (Burge and Brown 2006) allows the rationale for each requirement to be captured. This rationale can serve as a basis for negotiating requirement importance and could potentially be used to compute rankings for the alternatives.

11.3 Requirements Traceability

Requirements traceability typically refers to the ability to trace from the requirements all through the development process. The goal of traceability is to ensure that all system requirements are met. Requirements traceability is a key element in requirements management and is required to assess the impact and consequences of requirements changes (Nuseibeh and Easterbrook 2000).

Requirements can be traced in two directions. Tracing a requirement backwards refers to tracing back from the requirements specification to the origins of the requirement. Tracing a requirement forwards traces from specification through implementation and test. These two directions are referred to as Pre-RS traceability and Post-RS traceability, respectively (Gotel and Finkelstein 1994). The rationale for the requirements and for the developed system can aid in both kinds of traceability.

Pre-specification traceability is one of the more neglected forms of traceability (Gotel and Finkelstein 1994). The ability to know the origins of a requirement can be used later on if the requirement needs further clarification. Unfortunately, this information is often difficult to obtain. One way to track the origin of a requirement would be through the rationale for the requirement. The rationale would provide information on who argued for (or against) its inclusion and what the reasons behind the choice were. This information can be very useful in future development if the requirements need to change.

Post-specification traceability is what most developers think about when they think about requirements traceability—the ability to trace from the requirements through to the test cases in order to ensure that the software system meets its specification. Rationale can assist with post-specification traceability. The requirements, both functional and nonfunctional, can appear in the arguments for and against the many decisions made when designing and implementing the software. Each alternative chosen would eventually map to some development artifact, whether a section of a document, elements in a UML diagram, or the code itself. The arguments for choosing that alternative consist of requirements and nonfunctional requirements. The mapping from the alternative to its implementation would then provide traceability to those requirements. An example of this is the SEURAT system (Burge and Brown 2006), which captures traceability between code elements and alternatives.

The "rich traceability" proposed by Hull et al. (2002) supports both pre- and post-specification traceability by representing requirements at different levels—stakeholder requirements, system requirements, and design requirements. Rich traceability contains "satisfaction arguments" that can be supported by domain knowledge as well as information from other sources such as the output of modeling tools. These satisfaction arguments indicate how requirements relate to each other, in particular they capture when all of the requirements at one level are necessary to satisfy requirements at the level above (conjunction) or if any one requirement is needed (disjunction). For example, it may be necessary that all of a set of system requirements must be satisfied to satisfy the stakeholder requirement they relate to or it may only be necessary that one be satisfied.

Nonfunctional requirement (NFR) traceability is also important. The relationship between rationale and NFRs is described in the following section. Surveys of NFR traceability approaches can also be found in Hayes et al. (2005) and Cleland-Huang et al. (2005).

11.4 Rationale and Nonfunctional Requirements

While functional requirements describe the function of a system or device, nonfunctional requirements describe how the system or device should accomplish that function given "the constraints of a non-ideal world" (Thayer and Dorfman 1990). Nonfunctional requirements often refer to software quality and are related to Boehm et al.'s "Quality Characteristics." (Boehm et al. 1979). Roman (1985) describes NFRs as restricting the types of solutions under consideration. NFRs are not directly related to specific system components and often involve aggregate system behavior (Manola 1999). Research involving nonfunctional requirements and their impact on software development is taking place in a number of areas, many of which fall into the category of "separation of concerns" (Workshop 2000; Ossher and Tarr 1999). Concerns can fall into many, often overlapping, categories and can describe concerns about features, requirements, extensibility, performance, and reliability. Many categories of concerns have been proposed but the common thread is that each category describes attributes of a system that "cross-cut" the system's structure and/or functionality.

Functional requirements describe the functionality that a system needs to provide in order to satisfy the needs of its stakeholders. Nonfunctional requirements describe how well the system needs to perform that functionality or, in some cases, how the development effort needs to proceed in order to meet the needs of the customer and the developing organization. One way that the NFRs can be captured during requirements engineering and throughout development is in the rationale for the system. The NFRs would appear as arguments for and against different alternatives considered. The rationale can be analyzed to assess the impact of various NFRs on the software product and to determine how the decisions made might change if NFR priorities change.

11.4.1 Nonfunctional Requirement Categorization

When working with NFRs, it is often useful to work with a set vocabulary, or ontology, of terms. In rationale-based systems, a common vocabulary of keywords is needed to support semantic inference (Burge and Brown 2000). There are several different ways that NFRs have been organized or grouped. Bruegge and Dutoit (2000) referred to NFRs as "design goals" and broke them down into five groups: performance, dependability, cost, maintenance, and end-user criteria. Chung et al. (2000) provide an unordered list of NFRs and also hierarchies of NFRs for performance and auditing.

Some categorizations emphasize NFRs that relate to software quality and have formed quality measure hierarchies. The ISO/IEC 9126 software product quality standards (Jung et al. 2004) give six characteristics (functionality, reliability, usability, efficiency, maintainability, and portability) as well as 27 subcharacteristics. The CMU Quality Measures Taxonomy (CMU 2002) organizes quality measures into Needs Satisfaction Measures, Performance Measures, Maintenance Measures, Adaptive Measures, and Organizational Measures.

11.4.2 The NFR Framework

The view that quality characteristics are important when developing a software system was the driving force behind the development of the NFR Framework (Chung and Nixon 1995). The NFR Framework uses nonfunctional requirements, represented as Softgoals, to drive the software design process (Chung et al. 2000). This process produces the design, because the process is driven by the NFRs—its rationale. The NFRs are represented in a softgoal interdependency graph. The graph allows traceability from requirements to design decisions and from design decisions back to the requirements considered (Chung and Yu 1998). If requirements are changed, the goal graph can capture a historical record that relates new requirements to the old ones (Chung et al. 1996).

Cysneiros and Leite (2004, 2001) focused on how NFRs could be incorporated into the conceptual models represented in UML. They chose to create two views of the system: an NFR view, built on the NFR Framework (Chung et al. 2000) and a functional view, captured in UML. These two views should be connected at "convergence points." A Language Extended Lexicon (LEL) was built to contain the vocabulary used for the functional requirements and links to the NFRs. The LEL is generated first and is used in constructing the functional and nonfunctional views.

The NFR Framework was also used to support the Goal-Centric Traceability (Cleland-Huang et al. 2005) approach. Goal-Centric Traceability consists of four phases: goal modeling, impact detection, goal analysis, and decision-making. The goal modeling phase uses Chung's Softgoal interdependency graph (SIG) (Chung et al. 2000) to capture the NFRs and tradeoffs. Impact detection automatically creates links between the SIG elements and a functional model of the system captured in UML class diagrams using ontological keywords. Goal analysis propagates changes made to the goal contributions by the user through the SIG to determine their impact.

11.4.3 SEURAT Argument Ontology and NFR Prioritization

The ability to inference over the rationale has many different uses. One use, demonstrated in the SEURAT system (Burge and Brown 2006), is to evaluate the impact of changing priorities over the life-time of a system. This capability was supported by the use of an Argument Ontology (Burge 2005). This ontology, based on the NFR taxonomies described earlier (Bruegge and Dutoit 2000; Chung et al. 2000; CMU 2002; Jung et al. 2004) and extended to incorporate additional criteria, contains a hierarchy of reasons for making software decisions. The base elements of this ontology are Affordability Criteria, Adaptability Criteria, Dependability Criteria, End-User Criteria, Needs Satisfaction Criteria, Maintainability Criteria, and Performance Criteria. The hierarchy then subdivides these items into more detailed criteria (up to four levels deep). The Argument Ontology contains 277 terms and is documented in Burge (2005).

SEURAT uses the rationale to re-evaluate the support for each decision whenever the importance (priority) of an element in the argument ontology changes. This can show which (and how many) alternatives may need to be reconsidered. Another use of rationale supported by SEURAT is to detect relationships between functional requirements and the NFRs in the Argument Ontology. This can be done by looking for the ontology entries that appear in arguments along with the functional requirements. This may indicate a relationship between the goals depicted in the ontology and the functional requirements.

The rationale can also be used to analyze the reasons for and against the decisions made in order to determine how, and by how much, the goals determined during the requirements engineering process ended up influencing the final system. This is something that can be done during development to ensure that the program is staying on track and that the decisions are made in accordance with customer priorities and also after development to learn what might be the important factors to consider when developing future systems.

11.4.4 NFRs and Conflict Representation and Detection

The relationships between NFRs and the relationships between NFRs and FRs can be used to identify conflicts between requirements. In the case of NFR-to-NFR conflicts, it is important to determine how these requirements interact to avoid situations where one is met at the expense of another. This need to achieve “balance of attribute satisfaction” was the impetus behind the Quality Attribute Risk and Conflict Consultant

(QARCC) (Boehm and In 1996). Given a win condition generated using WinWin, QARCC uses a knowledge base of architecture (product) and process strategies for achieving quality attributes to check for conflicts. The knowledge base identifies the positive or negative impact that an architecture strategy has on affected quality attributes. The quality attributes are stored in a hierarchy where attributes at the highest level of abstraction, the "primary quality attributes" are mapped to stakeholder roles. Conflict detection is supported by the rationale captured in SEURAT by storing tradeoffs between quality attributes as background knowledge that is then used to detect conflicts. This differs from the approach used in QARCC by capturing the tradeoffs directly and not relative to a specific architectural decision.

Egyed and Grünbacher (2004) use the quality attributes to detect conflicts in functional requirements. In their approach, functional requirements have requirement attributes that relate to qualities such as efficiency, usability, and security. A cooperation and conflict model gives the relationships between qualities (positive, negative, or no effect). If a requirement has quality attributes that conflict with those of another requirement, that may indicate a conflict between the two requirements. This approach in and of itself would likely generate numerous false positives so it is augmented with trace analysis to only report conflicts between requirements that effect the same part of the code. The traces are generated by running the test scenarios that test each requirement.

11.5 Goal-Based Requirements Engineering

Requirements engineering can be viewed as the process of transforming stakeholder needs, or goals, into requirements that describe the system that will meet those names or goals. Even in cases where the stakeholders explicitly express their requirements, the system may be more successful if the goals behind those requirements can be expressed so that alternative ways to meet those goals can be explored (Antón and Potts 1998). In this section, we will look at two approaches involving goals: Goal-Based Requirements Analysis (GBRAM) (Antón and Potts 1998) and Goal Oriented Requirements Engineering (GORE) (van Lamsweerde 2001).

11.5.1 Goal-Based Requirements Analysis

In GBRAM (Antón and Potts 1998), goals are defined in two phases: goal analysis and goal refinement. In goal analysis, the analyst explores

various information sources to identify possible goals and classify them according to goal dependencies. In goal refinement, the goal set is pruned if necessary, goals are analyzed to identify obstacles towards the goals, and goals are operationalized (turned into formal requirements).

Specifications and scenarios, sometimes in the form of use cases, (Antón et al. 2000) are used as in puts to the goal identification process. One method used to identify goals is to look for verbs such as "avoid" or "improve" that are then followed by a desirable or undesirable condition. These verbs are also uses to categorize goals into categories based on the verb used. This categorization is used to separate user goals from system goals. User goals are identified as "achieve" goals while system goals (how the system responds to the user goals) are identified as "make" goals. The categorization differentiates between providing capability and providing information by using "notify" and "inform" to describe providing information and using "provide" and "allow" to describe providing capability (Antón et al. 2000). The goal categorizations used vary depending on the domain. The CommerceNet Web Server project described in Antón and Potts (1998) used avoid, ensure, improve, increase, keep, know, maintain, make, and reduce while the e-commerce system analyzed in Antón et al. (2000) used allow, achieve, make, provide, inform, ensure, and notify as the goal categories.

11.5.2 Goal-Oriented Requirements Engineering

Goal-oriented requirements engineering (GORE) focuses on the use of goals to drive requirements engineering (van Lamsweerde 2001; van Lamsweerde 2004). Goals can be functional goals, which are then used to build use cases and other "operational models" or quality goals that describe "preferred behavior" and are used to compare different alternatives as well as posing constraints (van Lamdsweerde 2004). The level of abstraction can also vary from high-level goals that are strategic to low-level goals that describe technical concerns (van Lamsweerde 2001).

The Keep All Objectives Satisfied (KAOS) method (van Lamsweerde and Letier 2000) represents goals and obstacles (undesirable conditions) in a formal temporal logic. The KAOS construct specification consists of two levels: a semantic net layer declaring the concept and its relationship with other concepts and a formal assertion layer that gives a formal definition. The second, formal, layer is optional and is used for formal reasoning while the semantic net layer supports modeling, traceability, and reuse. The goal specification defines the goal, and the property it should hold (achieve, cease, maintain, avoid), what other objects are involved, the parent

goal, subgoals that it should be refined to, and an informal description of the goal. The specification also contains the formal layer expressed in temporal logic. While goals describe desired behaviors, obstacles describe undesirable behaviors. Obstacles can be broken into five types: non-satisfaction obstacles that keep goals from being satisfied; non-information obstacles that obstruct information dissemination; inaccuracy obstacles that obstruct object state consistency; hazard obstacles that interfere with safety goals; and threat obstacles that interfere with threat goals.

van Lamsweerde and Letier (2000) define a requirements elaboration method that elaborates and operationalizes goals while also defining obstructions to those goals. This process starts with elaboration, where goals are refined; object capture that determines what objects are involved (objects can be entities, relationships, or events); operation capture that finds object state transitions; operationalization, which determines pre and post conditions; and finally responsibility assignments to identify alternative assignments and select alternatives based on nonfunctional goals. Obstacles and alternative resolutions are identified during the elaboration phase.

The GORE process is also supported by a software environment, Goal-Driven Requirements Analysis, Integration, and Layout (GRAIL), which supports editing, semantic checking, and views (Darimont et al. 1997). GRAIL contains a text editor for requirements acquisition and to check syntax and semantics. GRAIL can also present a graphical view of the specification.

11.5.3 Relationship to Rationale

These methodologies both utilize rationale. In GBRAM (Antón and Potts 1998), the rationale for requirements provided by the stakeholder is used during the refinement process to determine if there are additional requirements that need to be generated. As the refinement process proceeds, the rationale is tracked so that any unresolved issues can be monitored and eventually resolved. Each requirement generated in the GBRAM process is annotated with rationale: the questions, answers, alternatives, and scenarios that were generated and used during refinement.

Defining requirements using the GORE method produces the rationale for the requirements in the form of the goals that they were derived from. The goal hierarchy that resulted in the final requirement definition can be traced back to determine the rationale.

There are a number of places within the methodology where decisions need to be made. One is in the assignment of responsibility for the terminal

goals to “agents”: entities (humans, programs, devices, etc.) that perform operations or agents that monitor an object. Assumptions are defined as terminal goals that are assigned to “agents in the environment”, while requirements are defined as terminal goals that are assigned to “agents in the software” (van Lamsweerde and Leiter 2000). The alternatives are captured in the GORE process and the selection criteria can be captured as well.

There are also alternative resolutions to obstacles defined during the goal elaboration phase. The resolution strategies range from obstacle elimination to obstacle tolerance (van Lamsweerde and Leiter 2000). The choice of resolution strategy depends on the likelihood and severity of the obstacle. The alternative resolutions and reasons for resolution selection should be documented in the rationale.

11.6 Adapting to Changing Requirements

As stated earlier, failure to manage requirements, or more specifically manage requirements change, is a major cause of project failure. Managing requirements change requires addressing the following issues: identifying (the need for) change, impact analysis, determining when changes conflict, negotiation, prioritizing changes, change measurement, risk assessment, change estimation, planning (scheduling), and change learning (Lam and Shankararaman 1999).

These issues are strongly related to each other. For example, negotiation is heavily involved when determining the need for change, prioritizing changes, and scheduling change. Rationale has been shown to be an effective strategy in supporting negotiation by allowing the views of all the participants to be captured in a formal or semiformal manner. The ability to use the rationale in evaluating alternatives can be helpful in prioritization as well, especially if the rationale captures the importance of different evaluation criteria.

Impact analysis and risk analysis are closely related. A requirement having a higher impact on the system will bring a higher risk. If the requirement is a change to an existing one, the rationale could be used to determine what parts of the system were affected by the original requirement so that those could be modified. This is supported by systems such as SEURAT (Burge and Brown 2006) which use requirements as part of the argumentation and map the selected alternatives to the code that implements them. The ability to perform impact assessment also assists with change measurement since the impact on the system is related to the

amount of change needed. The impact assessment results will assist with change estimation as well.

One issue that rationale is especially helpful with is supporting the process of determining when changes conflict. The rationale records the intent behind the current choices made in a system and should also capture what tradeoffs were made. New requirements can be assessed against known tradeoffs. Another use of rationale is to avoid repeating mistakes that occurred in the past. If a change is proposed that was rejected earlier, the rationale will capture that decision and inform the analyst that there is a potential problem.

The goal of "change learning" is to collect information about changes that have occurred so that when similar changes happen in the future, information about that change will "reduce surprise" (Lam and Shankararaman 1999). Change information for a change, or type of change, can be captured in its rationale. The rationale would provide the reasons for the change, how it was made, and other pertinent information such as cost. Rationale also helps with learning by providing the intent behind the original requirements.

11.7 Summary and Conclusions

Requirements engineering is a crucial component of all software developments. The ability to successfully capture stakeholder needs and represent them in a way that they can then be used to drive software development has a significant impact on the success of software projects.

In this chapter, we describe the requirements engineering process and some key aspects including requirements elicitation, negotiation, prioritization, and traceability. We also discuss research in nonfunctional requirements and goal-based requirements engineering. These areas have strong ties to rationale. By using NFRs to drive system design, the NFR framework captures the rationale for each decision. Goal-based requirements engineering examines the goals that drive each requirement, i.e., its rationale.

Because of the criticality of requirements, and the high costs incurred if requirements are incorrect, incomplete, or mismanaged, capturing the rationale for the requirements should be a necessary step in the RE process.

http://www.springer.com/978-3-540-77582-9

Rationale-Based Software Engineering
Burge, J.E.; Carroll, J.M.; McCall, R.; Mistrík, I.
2008, XXXV, 316 p., Hardcover
ISBN: 978-3-540-77582-9