3 Rationale and Software Engineering

Software engineering, the process of developing software-intensive systems, is a complex area. This chapter introduces software engineering as well as the potential benefits in capturing, maintaining, and reusing rationale to support it.

3.1 Introduction

3.1.1 Software Engineering

According to the IEEE (IEEE 1993), software engineering is "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software." A more detailed definition of software engineering, and the one that we use during this book, was provided by Finkelstein and Kramer (2000):

> SE focuses on: the real-world goals for, services provided by, and constraints on such systems; the precise specification of system structure and behavior, and the implementation of these specifications; the activities required in order to develop an assurance that specifications and real-world goals have been met; the evolution of such systems over time and across system families. It is also concerned with the processes, methods and tools for the development of software intensive systems in an economic and timely manner.

Both the IEEE and the Finkelstein and Kramer definitions stress the necessity for a disciplined process of software development. This discipline is what puts the "engineering" in software engineering.

3.1.2 Software Engineering Rationale

Much of the research on rationale has addressed design rationale (DR). In domains such as engineering design, most critical decisions are made at design time and that is when the majority of the rationale is captured and used. During software development, while most development methodologies include a phase called design, decisions that drive software development are made throughout the process. We therefore view rationale as something that can be captured and used at all stages. In this book we use the term *software engineering rationale* (SER) to encompass all different types of rationale in many SE processes (Dutoit et al. 2006b) and to serve as a base for examining how SER can support the entire software engineering process.

3.1.3 Objectives of This Chapter

This chapter begins with a description of how rationale can be used to help define and implement the software process. This is then followed by a description of how rationale can support project management. The remainder of the chapter introduces how and when rationale can be used in software development.

3.2 Rationale and the Software Process

3.2.1 Software Process Definition and Implementation

In order for software development to be performed in a systematic and disciplined approach, it is necessary to follow some defined software engineering process. There is no single software development process that fits all types of software development. Instead, the software process used should be chosen, or defined, to best meet the organizational needs of the software developers as well as any process requirements that may be mandated by the client. According to the IEEE Software Engineering Body of Knowledge (SWEBOK) (IEEE 2004a), software engineering process (SEP) definition/development can be broken into four sub-areas: (1) Process Implementation and Change, (2) Process Definition, (3) Process Assessment, and (4) Process and Product Measurement.

The Process Implementation and Change subarea defines what needs to be known in order to either implement a new software engineering process or to change an existing one. This includes the definition of the infrastructure needed for process management, determining how the process will be managed, and selecting an appropriate quality improvement model.

The Process Definition subarea involves selecting the appropriate software lifecycle model, the software lifecycle process, determining the appropriate notation to describe the software process, adapting the selected process to meet the needs of the specific organization, and determining how, or what portions of, the process can be automated using process support tools.

The Process Assessment subarea utilizes assessment models. The Capability Maturity Model (CMM) (SEI 1997) and CMMI (CMMI 2006) are two examples. Process assessment also requires process assessment methods that can use information about the process to give it a rating, or "score."

Finally, the Process and Product Measurement subarea describes the need to measure process outcomes (its success at meeting process outcomes) and to perform product measurement to look at its size, structure, and quality. Of course, deciding what data to collect is insufficient; it is also necessary to decide how to assess the quality of the measurement results. A rigorous quality improvement process also involves collecting measurement data over time into a repository, modeling the information, and determining how the information can be fed back into the process on future projects.

3.2.2 Rationale and SE Process Decision-Making

We will describe the role of rationale in the software lifecycle and in software process improvement later in the book. Here, we will address how rationale can support the process definition process described in the SWEBOK as outlined above. Determining what the appropriate software development process is, and how that process should be managed and measured, involves making a number of very crucial decisions. The decision-making process involves determining the software process goals, the alternative means for achieving those goals, and evaluating those means to determine which alternatives best suit the goals of the specific organization and project.

Process Implementation and Change. In order to implement a new process, or change an existing one, many decisions must be made. What are the requirements for the new/adapted process? What changes should be made to the current process infrastructure? How is the process going to be managed? Which quality improvement model best suits the needs of the project? The rationale for the choices made when making these decisions can be used to determine if the reasons for these choices are consistent with project goals. It can also be compared with that from prior projects to see where past processes can be reused or adapted to meet new process needs.

Process Definition. The choice of process, and how rigorous that process should be, will have a significant impact on the software project. There are tradeoffs that need to be made between having a well-defined and rigorously monitored process and the cost and time that this may entail. Software life-cycle models are not "one size fits all." Selecting the appropriate model for a specific project involves careful examination of alternative lifecycles and their advantages and disadvantages relative to the needs of the organization. There are also many choices that need to be made when deciding if, and how, the process requires adaptation to meet specific organizational goals. It is important that adaptations are consistent with the goals of the lifecycle and do not counter its advantages. Process automation is also an area where decisions must be made. If automation is a high priority, it may prove to be a key driver in selecting the software process. The process may be chosen based on the tool support available and what that tool support is likely to cost.

Rationale should be recorded for the reasons behind the choices made. The explicit articulation of tradeoffs made will ensure that the choices are made for the right reasons and, if these decisions are revisited for future development efforts, that the effort that went into making these crucial decisions can be assist in making the correct decisions in the future.

Process Assessment. The choice of how a process will be assessed may or may not be under the control of the software development organization. In either case, the standards used to evaluate the software process can be captured in the rationale as criteria used to assess the other decisions made during the process definition process. The process outcomes identified will be the main criteria used to determine the process measurement strategy.

Process Measurement. There are many aspects to the software process and product that *could* measured during development. The question is, which of these *should* be measured in order to assess the software projects success at achieving process outcomes. Again, there are tradeoffs to be made between the time and effort it takes to perform process and product measurement against the value of the information obtained. Choices may be made based on tool support available to assist in this effort.

Capturing the rationale for these decisions can help to clarify what measurement options should be considered and what the reasons are for choosing them. The knowledge captured in the form of rationale can also assist future projects when they need to make similar decisions.

3.3 Rationale and Project Management

The Project Management Institute defines project management as "the application of knowledge, skills, tools, and techniques to project activities to meet project requirements" (PMBOK 2003). This definition is rather general but it is commonly understood that good project management is essential to ensuring that a project meets its goals of delivering quality software on time and within budget. Management needs to work successfully with the client to ensure that their needs are understood and met while also working with the developers to ensure that they have the knowledge and resources necessary to successfully develop the software product.

As in software development, rationale can play multiple roles. Rationale can assist with guiding and capturing the decision-making process when developing the management strategy for a project. As with software development processes, there is not one management solution that will work under all circumstances. Processes used in the past require tailoring to meet the needs of specific projects and the skills of specific teams. Rationale captured for management choices in the past can be used to determine if those choices are still valid for future projects.

Examples of some management choices include:

- Status reporting requirements for project teams
- Project team structure (size, distribution of responsibility, communication strategy)
- Necessity of hiring consultants with key technical expertise
- Frequency and duration of status meetings
- Role of software tools in the software project

Criteria for making these choices might include:

- Team member expertise and experience
- Team familiarity—experience of team members with each other
- Value of permanent employees learning new technology for future projects
- Budget provided for tool aquisition
- Management experience
- Customer flexibility (in terms of both deliverables and schedule)

It is critical that these key management decisions be made based on an understanding of the criteria that impact their success. Using the rationale to capture and evaluate these decisions helps to ensure that the management strategy selected best suits the needs of the client, product, and team.

Rationale can also assist with many project management or related tasks. Charette (1996) states that "large project management is risk management." The identification of risks is a crucial factor in successful software development. Capturing these risks, alternative mitigation strategies proposed, and the mitigation strategy used serves to both clarify the risk management process for the current project as well as form a knowledge base of risks, strategies, and outcomes for use in future projects.

Another aspect of software development where project management plays a key role is in the reconciliation of stakeholder viewpoints. Theory-W (Boehm and Ross 1989) is a software project management theory where the main goal is to "make everyone a winner." Theory W is based on Fisher and Ury's (1981) negotiation approach, where a key part of the negotiation involves identifying options and evaluating those using objective criteria. In Theory W, the key to a successful negotiation is to identify the stakeholder win conditions and to find options that create the win–win situations. The generation of these options and win–conditions is supported using the WinWin support system (Boehm et al. 1995). The information captured in WinWin is, in essence, the rationale behind the software requirements (Boehm and Kitapci 2006).

One of the more successful uses of argumentation-based rationale is to assist with structuring discussion during project meetings. The Issue-Based Information System (IBIS) notation (Kunz and Rittel 1970) is the basis of several systems applied to capture discussions in meetings. The indented text IBIS (itIBIS) system was used at NCR to capture project team meetings (Conklin and Burgess-Yakemovic 1996). This helped to focus discussion and point out potential problems with the requirements. Converting the textual rationale into a graphical form (gIBIS) exposed several problems with the proposed design that would probably not have been detected otherwise. The use of IBIS to aid in collaboration has continued with the Compendium project (Buckingham Shum et al. 2006) to perform "Dialogue Mapping." In their approach, a trained facilitator uses Compendium to capture discussion in an IBIS format during meetings. The results of the discussion can be displayed in real time to allow meeting participants to view, and reflect on, the discussion taking place.

3.4 Rationale and Software Development

The previous sections described how rationale can assist with defining the software development process and in managing the implementation of that process. Here, we highlight uses of rationale during the software development process by describing why rationale is needed, what some of the uses of rationale are, when rationale can be used during the process, and finally how it can be used. These areas are our primary focus during the remaining chapters of this book.

3.4.1 Why Capture Software Engineering Rationale?

Earlier in this book we defined rationale and its importance in software engineering. The success of any software project is dependent on the right choices being made during its development.

Software engineering contains many key challenges that can be addressed by the capture and use of rationale:

- *Software system longevity*. Software systems have been shown to remain in operation longer than the original developers probably anticipated. This longevity, and the need to continually evolve software to keep it viable, means that it is essential to understand the reasons behind development decisions made years earlier.
- *The Iterative nature of software development.* Many current software development processes utilize some form of iteration in order to increase their ability to adapt to changing requirements and technology. As development progresses, criteria appearing in the later iterations may affect decisions made in the earlier ones. The rationale can help to assess the impact of the changing criteria and guide the developer in making changes that implement the new functionality with minimum risk to that implemented earlier.
- *Stakeholder involvement*. There are many different stakeholders in a software development effort who have their own, sometimes conflicting, goals for the system. For example, the customer is concerned with the functionality provided by the system; the end user is concerned with how well it helps them perform their tasks and how easy it is to learn and use; the developers are concerned with how difficult it will be to implement; the managers are concerned with how long implementation will take and how much it will cost; all stakeholders are concerned with the reliability of the delivered system; etc. Capturing the decision-making process, and the stakeholders having input into that process, can

serve as a basis for negotiation. Rationale also captures how the different stakeholder priorities affect the developed system.

- *Knowledge transfer*. Significant amounts of expert knowledge are involved in the development of a large software system. This is information that will be lost if it is not documented, particularly at times of high turnover in the software industry. Rationale can serve as a key component in an organization's knowledge management strategy.
- *Increasing size and complexity of software systems.* Software systems have long since passed the point where their design is simple enough to exist in the heads of their developers. Rationale can assist as a "memory aid" to assist developers in remembering why they made their earlier decisions. Rationale can also be used to index into the code and documentation to determine the impact of changing decisions on the software.

3.4.2 What are the Uses of Software Engineering Rationale?

In order to convince software developers that capturing rationale is worth their time and effort (and convincing software managers that capturing rationale is worth some additional up-front costs), it is essential that the rationale is useful both during the initial requirements and design stages and later as the software is maintained and reused. We have identified several key areas of rationale use:

- *Presentation*. The use that immediately comes to mind for rationale is its ability to document the decision-making process. The ability to browse through, or query, the rationale-base to learn more about the decisions can assist developers in learning about the software, preventing the duplication of past work, and avoiding errors. The usefulness of the presented rationale will be dependent on the method of presentation. Ideally, presentation should be done within the same tools that are already in use to develop the software. The developer will be far more likely to know that the rationale is available and take it into account when making decisions if they do not need to use an additional tool.
- *Evaluation*. The CMMI (CMMI 2006) Decision Analysis and Resolution (DAR) process area stesses the importance of performing a "formal evaluation" of selected issues by evaluating alternative solutions (that address those issues) against criteria. Rationale can support this type of calculation by providing detailed information about the solution alternatives and their relationship to the decision criteria

(such as requirements, quality attributes, and assumptions). This information can be used to rate or rank the alternatives to evaluate the quality of the decision results. Rationale also supports usability evaluation, as demonstrated by the Scenario-Claims Analysis approach (SCA) (Carroll and Rosson 1992).

- *Collaboration*. Later in this book we describe how software development is almost always collaborative work. Rationale's importance to collaboration during software engineering was highlighted in Jim Whitehead's talk as part of the Future of Software Engineering track of the 2007 International Conference on Software Engineering (Whitehead 2007). Whitehead views architecture and design as "argumentative proceses" and proposes rationale capture, in the form of "collaborative argumentation" as an effective means of supporting these processes. The ability for rationale to support and capture this the negotiation required during software development has been demonstrated by many approaches, such as the WinWin (Boehm et al 1995) and Compendium (2006) systems described earlier.
- *Change analysis*. As mentioned earlier, software development is an iterative process. Software requires change both during the development process, as more information is learned about the requirements and incorporated into the software, and afterwards as it enters the maintenance and evolution stage of its life. Software may require changing for a multitute of reasons but one thing remains certain—the need to understand how the proposed changes impact the existing software. This includes both determining where the changes need to be made and also how those changes may affect the ability of the software to meet the requirements, quality criteria, etc. that were the basis of the decisions made during its initial development. With appropriate tool support, rationale can be used to identify change location and change impact. Rationale-based consistency checking can aid in consistency management—an ongoing process during software development and maintenance.

3.4.3 When can Software Engineering Rationale be Used in Software Development?

As mentioned earlier, rationale can support many aspects of software development and is not constrained to the design stage. These aspects include the "standard" development stages of requirements, design, etc. and also the cross-cutting areas of project management and reuse.

- *The software lifecycle*. Rationale can play a role in any of the software lifecycles selected to guide the software development process. Rationale also has a role in software process improvement, as mentioned earlier in this chapter.
- *Requirements engineering*. Rationale is involved in software requirements in several ways. One is in requirements elicitaiton and documentation. The rationale is a natural place to capture the relationship between the software requirements captured during elicitation and the source of those requirements. This provides a "rich traceability" back to the original customer requirements (Dick 2005; Hull et al. 2002). As with all aspects of software development, negotiation plays a role in requirements engineering as all stakeholders need to agree on what the requirements are. This negotiation and the parties involved can be captured in the requirements rationale. Requirements also appear in the rationale for the system as the arguments for and against alternatives. Capturing this information, and associating it with the code that implements the alternatives, is a form of requirements traceability (Burge and Brown 2007).
- *Software design.* Since much rationale research has been in the area of design rationale, it is no surprise that rationale for software design, and more specifically software architecture, is an active research area. Software architecture, while traditionally thought of in terms of components and connectors, is seen by some as "a composition of architectural design decisions" (Bosch 2004). This decision-centric view has encouraged more research into capturing the knowledge behind those decisions, as shown by workshops such as that on SHaring and Reusing Architectural Knowledge (SHARK).
- *Software verification and validation*. This is an area where the capture and use of rationale remains largely unexplored. Still, decision-making in software engineering does not stop when the development is complete. The planning and execution of an effective testing strategy requires making complex tradeoffs between cost and quality to ensure that the software meets the needs of its users while keeping testing costs under control. Rationale for the choices made when selecting testing methodologies and tools should be captured so that it will be available for use by subsequent projects or if the testing strategy of the current project requires re-evaluation.
- *Software maintenance*. One of the areas where the availability of rationale can be most valuable is during software maintenance. The challenge of software maintenance is ensuring that software evolves without damage to, or reduction in, the functionality needed by its users.

This is difficult because the maintainers may not be the same people who initially developed the code and often have a steep learning curve to understand an unfamiliar piece of software. The ablity to utilize the past experience of software developers via access to their rationale supports these goals.

- *Software reuse*. Reuse has often been refered to as “the holy grail” of software engineering. The ability to reuse software systems or components has shown great promise in allowing software delivery with fewer defects, higher quality, and in significantly less time. There are many types and levels of software reuse and, while all have advantages, reuse is not without its risk. Rationale can play several roles in reuse. One is to support decision-making about if and when reuse is appropriate for any given project. There may be some cases where the risk outweighs the benefits. Another use is to capture the reasons behind the decisions on what should be reused. There may be several reuse alternatives that should be examined. Rationale can also be used to evaluate reuse candidates. If the rationale behind those candidates is available, this information would provide valuable insight into the decisions that went into their design.

3.4.4 How Can We Support Software Engineering Rationale Use in Software Development?

In order for Rationale-Based Software Engineering to live up to its promise, we need to develop Rationale Management Systems that support its capture and use. As in any software development project, the first step is to identify the requirements. What are the uses of rationale that such a system needs to support? How does rationale, as we currently understand it, support software engineering and when does it fall short? How do we address those shortcomings?

Later in this book we provide two frameworks, one defining the key concepts in Rationale-Based Software Engineering and their relationships (the Conceptual Framework) and one that provides a framework for RMS development that supports the key features of RBSE needed to support software development (the Architectural Framework).

3.5 Summary and Conclusions

Rationale can play many roles throughout the software development process, both descriptive—by providing a richer view into the decision-making process, and prescriptive—by guiding that process and evaluating its results. There is however a small literature of doom-and-gloom discussions that dismiss the value of rationale relative to its cost, some even implying that the additional cost could make the difference between software project success or failure (Grudin 1996). Cost is an important factor in the equation, but it not a simple linear factor. Indeed, most nihilistic accounts of rationale describe development projects where rationale practices were implemented narrowly, manually, and incompletely.

Rationale provides technical leverage throughout all the processes and activities of software development. A broad approach to capture and reuse of rationale is required to enjoy multiplicative benefits of pervasive rationale practices. Software tools to support partial automation of rationale management can reduce the cost side of the equation even further. Finally, implementing rationale practices thoroughly in development organizations is critical. Process improvement efforts such as the CMM and CMMI involve rigorous documentation of software development that takes both time and effort. Initial studies on the CMMI (Goldenson and Gibson 2003) show that many of the companies studied showed cost, schedule, and quality improvement after adopting the processes.

When rationale practices are adopted broadly and with appropriate tool support, and when they are adopted thoroughly in development organizations, rationale has the potential to yield benefits that far outweigh its costs.