10 Rationale and the Software Lifecycle

Software development can be modeled using a number of different lifecycle, or process, models. These include the waterfall model, the spiral model, the Unified Process, the V-Model, and others. In this chapter, we will describe these models and how rationale capture and use supports the development process followed in each of them.

10.1 Introduction

10.1.1 Software Engineering Process

The software engineering process and the software lifecycle are closely related concepts. The software lifecycle refers to the stages of software development that take place over the lifetime of the software. The Institute for Electrical and Electronics Engineers/Electronic Industries Association (IEEE/EIA) defines the primary lifecycle processes to be acquisition, supply, development, operation, and maintenance (IEEE/EIA 1996). There are also supporting processes and organizational lifecycle processes (IEEE/EIA 1996). Supporting processes include documentation, configuration management, quality assurance, verification, validation, joint review, audit, and problem resolution. Organizational lifecycle processes include management, infrastructure, improvement, and training. While the International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) standards described earlier take a high view, the most typically mentioned lifecycle stages encompass the development and maintenance lifecycle processes and include requirements analysis and specification, design, implementation, integration, verification and validation (testing), installation/deployment, maintenance, and retirement. Software lifecycles are modeled by a variety of software process models that define how the development stages progress. The lifecycle model defines the "skeleton and philosophy" of the process (Fuggetta 2000).

The software process is what controls and monitors the development described by the lifecycle model. The software process is defined by Fuggetta (2000) to be "the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain a software product."

Rationale can play a role in software process by capturing the reasons behind both process and product decisions. The product rationale captures the reasons for decisions that directly impact the delivered product, while the process rationale describes the reasons behind the process selected to guide the product development. Process decisions are important because the process chosen needs to fit the size of the project, the experience level of the development team, and the development tools available.

10.1.2 Objectives of This Chapter

In this chapter, we describe the stages of the software development lifecycle and how rationale applies to each of them. We also describe a number of software lifecycle models. We conclude with a section on software process improvement.

10.2 Development Activities and Rationale

The software lifecycle consists of a number of stages of software development. In this section, we briefly describe a typical set of development stages and how rationale can be captured and used in each of them.

10.2.1 Project Planning and Management

While project planning and management is listed first among the stages, planning and management are ongoing activities throughout the development process. Project planning involves many decisions: delivery date, staffing needs, budget, milestones, deliverables, etc. These decisions involve many tradeoffs. For example, one tradeoff might be assessing the importance of short time-to-market versus the amount of functionality provided or the quality level of that functionality (how much time to spend on validation and verification). These decisions and the reasons for the choices made should all be captured in the rationale. The process of recording deliberation during planning as rationale assists with collaboration and negotiation.

Management decisions can also be captured in the rationale for the project. Rationale can support collaboration, risk management, success criteria reconciliation, process improvement, and knowledge management.

10.2.2 Requirements

Requirements engineering is arguably the most crucial stage in the software lifecycle. Failing to capture and refine requirements adequately is considered to be a leading cause of project failure (Alford and Lawson 1979; Hofmann and Lehner 2001). Rationale can support requirements elicitation by capturing reasons behind requirements and allowing comparison with stakeholder needs, enabling requirements negotiation by capturing the deliberation process, assisting inconsistency management by allowing comparison of priorities across requirements, and in requirements prioritization, a key element of Value-Based Software Engineering (Boehm 2006b) by associating priorities to the criteria behind each requirement, both functional and non-functional.

Rationale can also play a large role in requirements traceability by providing the means to associate the decisions made later in the development process with the requirements that drive them. This applies to both the functional requirements as well as nonfunctional ones. Both types of requirement can appear in arguments for and against alternatives that are captured in the rationale.

10.2.3 Design

Much of the research involving rationale has been in the area of design rationale—the reasons behind design decisions. In software, there are several levels of design that take place depending on the size of the system being built. High-level design is often referred to as architectural design. This stage involves designing or selecting the software architecture. The choice of architecture is often driven by the "quality requirements" (non-functional requirements) of the system. For example, Attribute-Based Architectural Styles (ABAS) (Klein and Kazman 1999) associate software architectural styles with quality attributes such as performance, availability, and modifiability.

The design process progresses from the high-level decisions made when performing architectural design into the lower-level decisions in detailed design as classes, or modules, are designed. The rationale can be used to capture the decisions made at this point in the process and eventually linked to the code that will implement the alternatives selected.

10.2.4 Implementation

Implementation involves translating the design into the executable source code. There are still decisions made during this part of the process and the rationale for these decisions should be captured. The rationale can be evaluated to ensure that the reasons chosen are consistent with those given at earlier stages of development. The rationale can also be used during software maintenance to describe why the software was implemented the way it was and to help prevent new decisions from counteracting those intentions.

10.2.5 Verification and Validation

In order to ensure that the developed system provides the functionality needed by the customer and that it meets its specification, it needs to be tested. The evaluation process is typically described as verification and validation (V&V). While we often describe this stage as occurring after implementation, in reality V&V activities should take place all the way through the development process. Test planning should be started when the project planning is performed, requirements should be examined to ensure that they are testable, unit testing should be performed during implementation, system testing is performed prior to deployment, and regression testing (as well as any new tests) must be performed when changes are made during maintenance.

Boehm gave an often-cited definition of the difference between validation and verification—validation asks "are we building the right product?" and verification asks "are we building the product right?" (Boehm 1979; Sommerville 2007). Verification involves ensuring that the software conforms to its specification while validation involves checking that the software does what the customer needs it to do.

Rationale can support software testing by providing insight into how quality factored into software decisions. This information can be used to determine where testing efforts should be concentrated. Collecting rationale for the testing effort itself would be useful in assisting with making testing decisions and in using the reasons behind testing choices and the results of these decisions to point out testing strengths and weaknesses that can be applied to future projects.

10.2.6 Maintenance

A successful software system is likely to require some form of maintenance over its lifetime. These changes can be challenging, especially if the original developers are not available. This is an area where rationale is especially valuable. Knowing the intent behind the decisions made when developing the software can help to prevent problems or inconsistencies being introduced during maintenance. If the rationale captures the assumptions made when initially building the system it can be used during maintenance to suggest where changes need to be made if those assumptions change. This assistance is provided in the Software Engineering Using RATionale (SEURAT) system (Burge and Brown 2006).

10.2.7 Retirement

If, or when, to retire a software system is potentially the last decision that needs to be made during the system's lifetime. The decision on whether to repair (maintain) or replace a system needs to be well thought out. This deliberation can be supported by and captured with rationale. The rationale for the decision would also be valuable if the retired system ends up being reinstated or reused later.

10.3 Software Lifecycle Models

There are a number of different categorizations for software lifecycle process models. Here we have chosen to break them into three categories: sequential models where development typically proceeds linearly through the phases, iterative models where iteration is built into the models, and a third category for models that do not fit into either of the two categories or that span categories.

10.3.1 Sequential Models

10.3.1.1 Waterfall Model

The waterfall model was originally defined by Royce (1970). In this model, development proceeds through the stages in a sequential fashion as shown in Figure 10.1. Each stage (shown as a box in the figure) needs to complete before the next stage can begin. The example shown here

includes feedback loops indicating that it is possible to go back to make modifications to work done earlier if necessary. The stages vary slightly between different depictions of the model but typically include requirements, design, implementation, and testing, and may also include maintenance, deployment, and retirement.

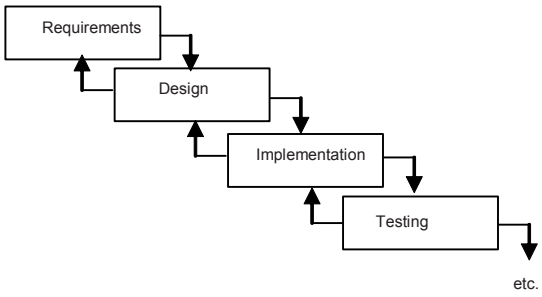

Fig. 10.1. Waterfall Model

The waterfall model has fallen somewhat out of favor. The separate stages are seen as being inflexible and less responsive to changing requirements. The model does, however, have the advantage that it is easy to assess where in the process a software project is, something not always clear with more iterative methods. This model resembles models used in other kinds of engineering projects and is often used when the software is part of a larger systems engineering project (Sommerville 2007).

Each of the stages captured in the waterfall model will include many decisions that will have a large impact on the later stages. Capturing the rationale for these decisions will help to ensure that decisions made in later stages will be consistent with earlier ones.

10.3.1.2 V-Model

The V-model is similar to the waterfall model but also includes the verification activities and how they relate to development stages. A key difference between the V-model and the waterfall model is that the level of abstraction is explicit (Bruegge and Dutoit 2004). Figure 10.2 shows a simplified V-model, adapted from Bruegge and Dutoit (2004) and Jensen and Tonies (1979). As with the waterfall model, capturing rationale can help with the traceability of decision criteria throughout the process.

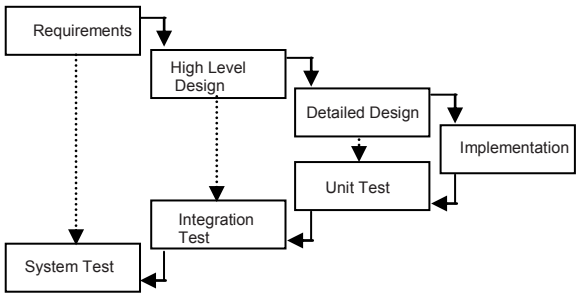

Fig. 10.2. V-Model

10.3.2 Iterative Models

Iterative models differ from sequential ones in that they depend on the software being built in a series of iterations. In this section we briefly describe some of the more common models.

10.3.2.1 Incremental Delivery

Incremental delivery consists of portioning the system into a series of releases. The initial requirement development and architectural design is done for the system as a whole but the functionality is delivered incrementally. This method has several advantages including making the software available to the users earlier, gaining experience with early increments to help refine requirements for later ones, reducing the risk of project failure, and ensuring that the most important functionality (typically developed in the earlier increments) receives the most testing (Sommerville 2007).

10.3.2.2 Spiral Model

The Spiral Model, developed by Boehm (1986), depicts the software development process as a series of increasingly more developed prototypes. The spiral moves through four quadrants. The first quadrant looks at objectives, alternatives, and constraints on the next development cycle. The second quadrant evaluates the alternatives proposed in the first quadrant and identifies and resolves risks. The third quadrant develops and verifies that level of the product (the prototype), and the fourth plans out the next phase

or phases. This model both explicitly addresses risk and, by the alternative identification and evaluation steps in the first two quadrants, the rationale.

Rationale is supported in the Theory W (win-win) extensions to the spiral model (Boehm and Bose 1994). In Theory W, stakeholders are identified for each revolution through the spiral along with their "win conditions." These win conditions are used in defining objectives, constraints, and alternatives. The win conditions and the alternatives generated during the spiral model process form the rationale for the system.

10.3.2.3 Unified Process

The Rational Unified Process (RUP) (Kruchten 1999) and its more general form, the Unified Software Development Process (Jacobsen et al. 1999), consists of four phases, with multiple iterations taking place during each phase. The four phases are inception, where the initial business case is defined; elaboration, where requirements and risks are defined; construction, where the system is designed, programmed, and tested; and transition where the system is moved into its operational environment (Sommerville 2007). Within each of these phases, there are nine core workflows: business modeling, requirements, analysis and design, implementation, test, deployment, project management, configuration and change management, and environment. The amount of effort spent in each of these workflows depends on the development phase. For example, more time is spent on business modeling and requirements in the inception and elaboration phases and less in the construction and transition phases. Similarly, the amount of implementation slowly increases in the first two phases, which may involve simple prototypes, reaching its highest level in the construction phase when the actual system is built. The Rational Unified Process was developed by Rational Software and is supported by its products.

The Unified Process is a generic and comprehensive process that attempts to cover all aspects of software development. Because of its comprehensive nature, it can be seen as being too unwieldy for smaller development projects. The process can, however, be adapted to work with smaller projects (Hirsch 2002; Pollice et al. 2003). Process rationale can be captured to document how the process was tailored, and why. This information can then be used to transfer the lessons learned to future software projects using the same or similar processes.

10.3.2.4 Extreme Programming

Extreme Programming (XP) can be viewed as a variant on incremental delivery (Sommerville 2007). The extreme in extreme programming does

not indicate “daredevil programming” but instead refers to taking existing best practices to the extreme (Beck 1999). The development process is a collaborative one between the customer and the developer where functionality is described as a series of stories (similar to use cases) and where each release chooses the set of stories that are viewed as the most important. Releases are developed using test-first development and pair-programming.

The goal of XP is to center the development process on coding and to try to develop releases that are as simple as possible and to plan on refactoring later if necessary. The danger of this is the difficulty of knowing where short-cuts were made that may need to be re-examined in later releases. Documenting the rationale for the decisions made in earlier iterations can be used to detect where alternatives were chosen in the interest of expediency that may require change as requirements are added or refined. The value of this is demonstrated by the Software Engineering Using RATionale (SEURAT) system (Burge and Brown 2006) where non-functional requirement priorities can be modified and used to detect where earlier choices should be reconsidered. A rationale-based support system such as SEURAT can be used during XP to detect candidates for refactoring.

10.3.3 Other Models

10.3.3.1 Rapid Application Development

The goal of Rapid Application Development (RAD) is to build software products more quickly, and with higher quality, than can be done using more traditional software life-cycle approaches (Martin 1991). This is accomplished by taking advantage of Computer-Aided Software Engineering (CASE) tools and fourth-generation language tools. RAD is an approach that can be used to build data-intensive business applications (Sommerville 2007) by exploiting commonalities between these systems: forms needed for data input and display, database access, commonly used office applications such as word processors and spreadsheets, and report generation. Many RAD projects are a form of COTS-based development projects because they link together existing Commercial Off-the-Shelf (COTS) applications to provide the required functionality (Sommerville 2007). RAD is often confused with rapid prototyping but the key difference is that rapid application development is intended to build the final system while a prototype is typically built to gain a better understanding of system requirements or available technology.

The success of a RAD development effort hinges on the selection of the tools, products, and COTS applications used in its construction. There may need to be compromises made to adjust system requirements so that they can be supported by these tools and components. Capturing rationale for the choices made and alternatives considered assists the selection process by making the reasons for selection and any tradeoffs made explicit. The rationale, and the alternatives captured in it, is also useful if subsequent versions of the system need to reconsider these decisions. RAD systems run the risk of dependence on third-party software where the vendor may go out of business, stop supporting the product, or raise licensing fees. These vendor changes may necessitate a change in the system to avoid problems.

10.3.3.2 Component-Based Software Engineering

The Component-Based Software Engineering (CBSE) development process builds software products out of reusable components. The goal is to make software engineering more like other engineering disciplines where parts are ordered from a catalog and configured using well-defined interfaces in order to create a new product. CBSE relies on the availability of components and on being able to adapt requirements, when necessary, to work with these components. CBSE is not strictly a process or a life-cycle. The components can be developed and used within any of the life-cycle models described here.

Rationale can be used during CBSE by both component providers and consumers. For component providers, the component rationale can describe both functional and nonfunctional capabilities of the component. For component consumers, the rationale can be used to find a component that best matches the functional and nonfunctional requirements of the system under development.

10.3.3.3 Open-Source Software Development

Open-source software development involves multiple software developers working together over the Internet to build software systems where the code is freely available to all. This has resulted in a number of successful software projects including the Linux operating system (www.linux.org), the Apache web server (www.apache.org), and Mozilla project products (www.mozilla.org) such as the Firefox browser and the Bugzilla bug-tracking system. There have also been open-source projects with corporate support, such as IBM's Eclipse development framework (www.eclipse.org). The unifying attribute of these systems that has made

them successful is that they are all systems that the developers want to be able to use themselves. Successful projects result from developers solving problems that they are excited about (Raymond 2001).

Since open-source development is a highly collaborative process where developers can come and go from the project at will, the capture and use of rationale could play a significant role in the success of these efforts. Successful open-source projects such as Apache and Mozilla make heavy use of version control systems, such as CVS, and bug tracking (Mockus et al. 2002). These systems capture the reasons behind software changes that could be included in their rationale. Capturing the intent behind the software modifications can be used to help guide the developers as the system evolves.

10.3.3.4 Model-Driven Development

Models have been used to assist with software development for many years. The simplest definition of model-driven development (MDD) is to built a model of a system that is then transformed into the system itself (Mellor et al. 2003). A more specific view is to develop domain models for application areas and use those to develop system architectures (Boehm 2006a). Models used in MDD can be developed using UML (France et al. 2006) or domain-specific modeling languages (DSMLs) that define relationships between domain concepts along with semantics and constraints (Schmidt 2006).

The usefulness of these models would be increased if they were developed with rationale attached. This would assist in selecting the appropriate model for the problem that the system is solving and could also help to determine when tailoring the model would be appropriate or not.

10.3.3.5 Service-Oriented Development

In service-oriented development applications are built using stand-alone services that can be executed on distributed computers (Sommerville 2007). Services are accessed via a service registry which is used to find applicable services. When a service is found by an application, the application is then bound to that service. A key aspect of service-oriented development is the ability to perform "ultra-late-binding" where the service is located and bound dynamically (Turner et al. 2003). Web services are an example of the service oriented development paradigm.

The uses of rationale in service oriented development are similar to those in CBSE: the rationale can be used as part of the selection criteria used when discovering service providers. For example, the Web services

stack framework proposed in (Turner et al. 2003) includes a non-functional description level that provides a non-functional description of a service. These protocols would then provide the rationale for selecting the service.

10.4. Software Process Improvement

As described earlier, the quality of software products is related to the quality of the software process. In this section, we describe two process improvement initiatives: the CMM and CMMI process improvement framework and the Personal Software Process.

10.4.1 CMM

The Software Engineering Institute (SEI) developed the Capability Maturity Model (CMM) (Paulk et al. 1993) to define software maturity levels. These levels are initial, repeatable, defined, managed, and optimizing. At the initial level, the process is undefined and unpredictable. At the repeatable level there are policies and procedures in place for the software process. Companies working at the defined level have documented and standardized procedures that work across the organization. At the managed level metrics are collected to assess the quality of the software process and at the optimizing level this information is fed back into the process to improve it.

The Capability Maturity Model has been replaced with Capability Maturity Model Integration (CMMI) (CMMI Team 2006). The CMMI integrates the software CMM with the Systems Engineering Capability Model (SECM) (EIA 1998) and the Integrated Product Development Capability Maturity Model (IPD-CMM) (SEI 1997). The CMMI has two representations—a staged model that assesses the organizations process at one of five discrete levels (similar to the CMM) and a continuous model where different process areas within an organization can be ranked at different capability levels. The capability levels are incomplete, performed, managed, defined, quantitatively managed, and optimizing. There are 24 process areas defined within the CMMI. Examples are project planning, requirements management, and configuration management.

Rationale capture and use is related to the CMMI Decision Analysis and Resolution process area. This process consists of defining a "formal evaluation process" for evaluating decision alternatives. This process includes identifying the alternatives, determining the evaluation criteria, selecting and using the evaluation method, and selecting the alternatives

based on the criteria (CMMI Team 2006). The evaluation process used on a project should determine which categories of decision will require formal evaluation (such as high-risk decisions) and how the evaluation will be performed and documented.

10.4.2 Personal Software Process

The Personal Software Process (PSP) (Humphrey 1995) arose from applying the CMM to small software projects. The CMM focuses on improving the process of software development organizations and the PSP extends that focus to improving the process of individual software engineers. The PSP follows the principles that each developer needs to base their process on data that they collect on their own performance, the developers need to follow a defined and measured process, developers need to be responsible for the quality of their work, and that defects should be avoided if possible, fixed as soon as they are detected, and that the right way to do the job will be the fastest and cheapest (Humphrey 2000).

The PSP follows a process improvement cycle where individual developers capture metrics on their job performance: time spent and defects introduced and removed. These metrics are then used to improve their performance. The PSP provides detailed forms and scripts to use during the development process.

The Team Software Process (TSP) (McAndrews 2000) extends the PSP to developing software in teams. The TSP addresses four causes of project failure: lack of training in planning, development, and quality practices; the focus on schedule rather than quality; the lack of a formal team-building process; and unrealistic project plans damaging motivation. The TSP defines how Level 5 of the CMM can be put into practice.

Neither the PSP nor TSP calls for the recording of rationale as part of the process. The success of these approaches, however, indicates that emphasizing quality over schedule concerns leads to more successful projects. The addition of rationale to the collected data would add to this success by providing additional insight into the development process that can then be used to tune these processes during future development. It is clear from the results of PSP/TSP projects that spending time up front to collect data ends up improving the process and not having the detrimental effect on schedule that is so often feared.

10.5. Summary and Conclusions

The incentive behind the defining, modeling, and monitoring of the software lifecycle is to increase quality and decrease costs. Software process models have evolved from sequential models towards more iterative ones in order to be more responsive to changes in software requirements. The importance of a defined and monitored software process has been highlighted by process improvement efforts such as the CMMI and the PSP.

The capture and use of rationale should be an integral part of any development process. The usual software artifacts produced during development only describe what was done and not why. Knowing the information behind the decisions can provide much-needed insight when these decisions are the basis of future ones. The reasons for making decisions that are captured in the rationale are often nonfunctional requirements that affect overall software quality. The rationale can provide a way to evaluate that quality and support quality improvement.

Much of the opposition to the capture and use of rationale has been the view that it is difficult and time consuming to collect. This argument can be used against most forms of documentation but it is rare to find anyone who does not believe that documenting software will not save money in the long run. As software processes become more rigorous, the cost of collecting rationale will continue to become less of an issue compared to the savings provided by the defect reduction and requirement conformance provided by the improved processes.