Software Requirements

Pete Sawyer

THE CONTEXT OF SOFTWARE REQUIREMENTS

Software requirements concern the specification of software systems. Software requirements are always derived from some business problem such as processing passport applications, improving automotive safety systems, or adding features to cell phones. Sometimes, the solution will be implemented entirely in software. At other times, software will be only one of the technologies that need to be integrated into the solution. Projects may develop new software products or they may be concerned with evolving existing or legacy systems. In all cases, the software will be embedded in an operational context so it will have interfaces to human users, business process elements, or other software or hardware systems (Figure 1).

The consequence of this is that derivation of the software requirements can very rarely be isolated from the underlying business problem. In most cases, the software requirements have to be developed with full awareness of the real-world context. Hence, software requirements are not really a discrete area of software engineering. Rather, they are part of the systems engineering process known as requirements engineering (RE). RE deals with all aspects of problem understanding and solution specification.

Transforming a requirement into software is a complex process that consumes many resources. The deeper into the process, the more design and implementation strategies become committed to satisfying the requirement. The effect of this is to make the costs of rectifying errors in the requirements increase dramatically as development proceeds. An effective RE process that minimizes the occurrence of requirements errors is, therefore, critical to the success of any development project.

The RE process is the responsibility of the *requirements engineer.* Requirements analyst, systems analyst, and business analyst are some common alternative names for requirements engineer. The term *business analyst* is particularly revealing because it correctly implies that the requirements engineer's role is not purely a technical one. The requirements engineer must be able to understand the domain of the business problem and translate the problem that derives from this domain into a specification for a software solution. Hence, the requirements engineer bridges the business and technical domains. This clearly calls for a combination of skills. Software engineers performing the role of requirements engineer (requirements engineers do not always have a software background) need to supplement their technical skills and knowledge with business domain understanding and the "people" skills necessary to draw information from customers, users, and other stakeholders.

Requirements and Constraints

A requirement defines a property or capability that must be exhibited by a system in order for it to solve the business problem for which it was conceived. Thousands of requirements, of different kinds and at different levels of abstraction, may be needed to specify a system that addresses a challenging business problem.

The classic way to categorize requirements is according to whether they are functional or nonfunctional. A functional requirement describes some function that the software must perform. For example, a requirement for a burglar alarm system to activate the siren when a sensor is tripped is a functional requirement. It defines an action to be performed by the burglar alarm on the occurrence of some stimulus, whether an event in the environment or a request by a user.

Nonfunctional requirements describe qualities of a system. The most important class of nonfunctional requirements[1] address how well the system operates within its environment. Essentially, they specify the quality of delivery of the functional requirements and are sometimes called extrafunctional requirements because of this. Common types of nonfunctional requirements include reliability, availability, security, safety, usability, and performance requirements.

Some requirements are emergent properties. These are dependent on a wide range of factors, some of which are hard to analyze and control. Requirements engineers and developers feel most comfortable when a requirement for a system property

[1]Another class of nonfunctional requirements address nonoperational issues such as maintainability.

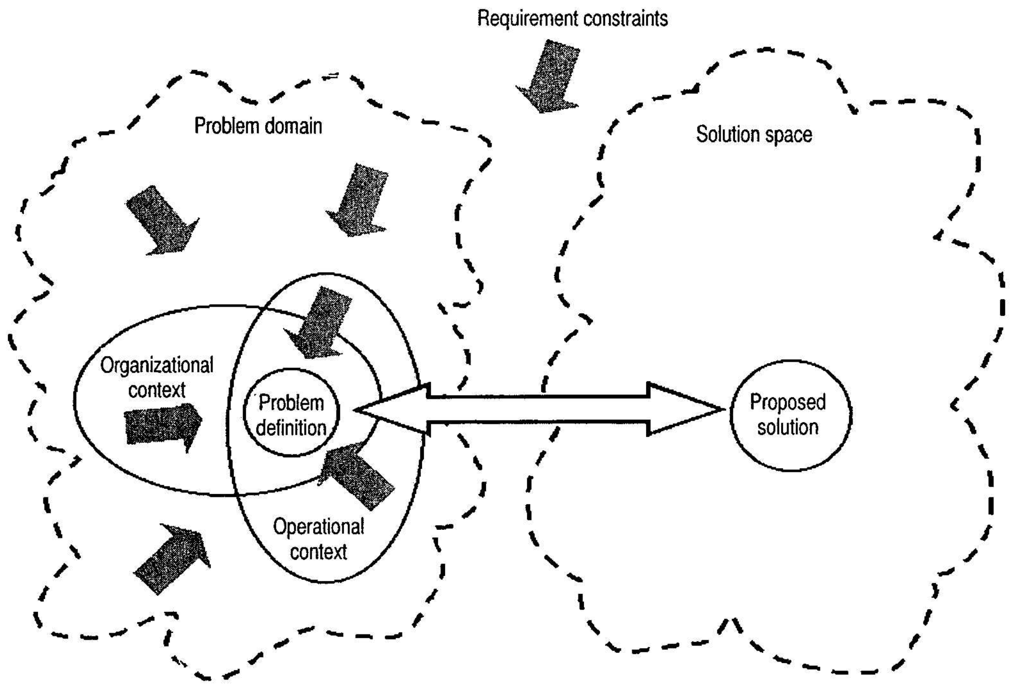

Figure 1. Requirements, constraints, problems, and solutions in RE.

can confidently be allocated to a particular system component that assumes responsibility for satisfying the requirement. Emergent properties are the antithesis of this—satisfying a requirement depends upon how the whole system operates within its environment. For example, the performance of a distributed system may be an emergent property because it is dependent upon the design of the system but also on factors such as demand and network traffic.

Requirements are usually specified at several points on a spectrum that ranges from those with a business focus to those with a technical focus. The technically focused requirements exist only to make it possible to satisfy the business-focused requirements. For convenience, we will treat these as levels, where the highest level requirements are those with a business focus.

At the highest level are the goals of the system that set out in very broad strategic terms what is needed to solve some business problem. Identifying these is what typically motivates a development project. The next level of requirements define what must be observable (by the people or other systems in the environment) in a black-box system that would solve the business problem. These are often called the user requirements. This term is a little unsatisfactory, however, because users (the people who will interact directly with the system) are not the only sources of requirements. Stakeholder requirements is a better term because it is inclusive of users and others who impose important requirements, such as managers or external regulators. Requirements may come from nonhuman sources such as the environment or the application domain (Figure 1). For this reason, the term *system requirements* is used throughout this chapter to describe required properties observable by users, other stakeholders, and other systems in the system's environment.

One of the main tasks of requirements analysis is to elaborate the system requirements to discover more about the implications of satisfying them. This involves deriving new, lower-level requirements (*derived requirements*) that focus more on detailed technical issues. (Confusingly, technically focused requirements are sometimes called system requirements, although not in this chapter.)

Software requirements are the requirements that a software system or component has the task of satisfying. In a domestic burglar alarm system, only a subset of the requirements would be software requirements because other requirements would specify the properties of hardware components such as sensors and sirens. A PC image manipulation application, by contrast, would run in a software environment and involve no hardware components. Even here, not all the requirements may be software requirements, because there may also be requirements for an installation guide and a user manual.

Constraints are like negative requirements. They act to limit the set of possible solutions to the business problem. For example, noise legislation may act as a constraint that limits the length of time that a burglar alarm is permitted to sound its siren. Many constraints are technical and concern, for example, memory, processor speed, and bandwidth. Others are related to the problem domain. For example, on-board train control systems are constrained by the high levels of electromagnetic radiation in the trackside and on-board environments.

REQUIREMENTS ENGINEERING PROCESS

A requirements engineering (RE) process must transform a business problem into a specification of the properties of a system that will provide an appropriate solution to the problem. Arriving at a solution specification requires the application of a systematic and rigorous process in order to understand the problem and the impact of a range of possible solutions.

The RE process is broadly composed of the set of activities that is reflected in the structure of the following sections of this chapter. The properties that the software must exhibit have to be *elicited*. The elicited requirements need to be subjected to *analysis* in order to establish a set of requirements that is correct, complete, and feasible. The set of requirements that emerge from the analysis activity need to be recorded in a *specification* document that communicates the requirements to the people who will use them to develop the software. The documented requirements need to be *validated* to ensure that the software that they specify will meet the needs of the people from whom the requirements were elicited (Figure 2). As development proceeds, the requirements need to be *managed* so that changes are controlled.

RE is never a strictly linear process. For example, analysis of the requirements is invariably closely coupled with their elicitation so that the requirements engineer and the stakeholders can work closely to develop a consistent understanding.

The IEEE publishes three standards that directly address the RE process and have evolved to make the handling of requirements rigorous and controllable:

- IEEE Std 1362-1998, *Guide for Information Technology—System Definition—Concept of Operations (ConOps) Document* [IEEE 98a]
- IEEE Std 1233-1998, *Guide for Developing System Requirements Specifications* [IEEE 98b]
- IEEE Std 830-1998, *Recommended Practice for Software Requirements Specifications* [IEEE 98c]

The process implicit in this set of standards begins with scoping the system. This involves understanding the underlying problem that the system is to address, identifying the goals of the system, and outlining how it will operate in its environment. This is the concept of operations or *ConOps*. This is followed by a process in which the system requirements are elicited from their sources, analyzed, and validated. The product of this is a *system requirements specification* document that defines the requirements for the overall system. For each software component, further analysis of the allocated requirements is used to derive the requirements that fully specify the software. These are documented in a *software requirements specification* (SRS). The SRS forms the definitive set of requirements that the component must satisfy and is sufficiently detailed to allow development to commence. The development of a system architecture from the system requirements specification is an implicit prerequisite for the identification of the software (and other) components.

This process has evolved to deal with large systems engineering projects, typically comprising both hardware and software. Such systems are often developed using a supply chain of subcontractors responsible for delivering system components and who are managed in a hierarchical relationship with, at its root, a main contractor. The complexity of such projects means that very serious risks are posed by requirements change (e.g., the emergence of new, erroneous, or poorly understood requirements) late in the development process. This requires an RE process that seeks above all to minimize the risk of unexpected requirements change. The process achieves this by ensuring that the problem and its solution are rigorously analyzed and docu-

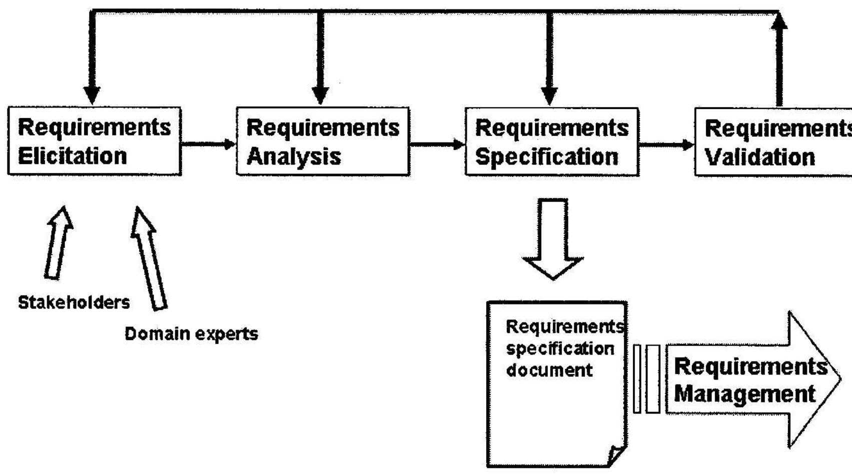

Figure 2. Generic model of the RE process.

mented before serious development begins. Some requirements change will still occur, but the extent and impact of the change should be controllable.

In some domains, it is impossible to identify all the requirements that will serve to define a product for its expected service life. If the product's environment is volatile, the product's requirements will also be volatile. When software products are developed to compete in a competitive open market, for example, the requirements are likely to evolve with the market. Other pressures may also impact on the RE process. In particular, capturing market share sometimes means that meeting release date deadlines is traded off against rigor in the RE process. This trade-off is a delicate one that can easily result in short-term expediency storing up problems for later in the product life cycle if, for example, requirements are poorly managed. Despite this caveat, small projects to develop noncritical software undertaken by companies with relatively simple organizational and management structures may be able to cope with some requirements change, provided that the likelihood and extent of change is predicted and acknowledged. Such projects need to have mechanisms in place to enable them to adapt gracefully when requirements change occurs.

One increasingly common response to coping with change is to design the development process for frequent releases of the software when development proceeds in relatively small increments. The RE process needs to be synchronized with this by being enacted in increments (Figure 3). Here, requirements to be implemented in each release are selected from a pool of requirements held over from the previous release, and also newly emerged requirements.

In the incremental RE process depicted in Figure 3, RE proceeds in parallel with other development phases. When release *n* of the software is being implemented, development of release *n* + 1 is prepared by selecting the requirements that it is to satisfy. It is clear that in this process, RE is not simply a front-end activity that ceases when the requirements have been documented and design commences. Instead, it is a continuous process that lasts the whole product life cycle. In fact, this is true of all RE processes. Even where the process is optimized to minimize requirements change, some reworking of requirements is inevitable after design and coding has commenced. The RE process normally consumes most effort early in a project, but effort needs to be allocated to requirements management and coping with change following sign-off of the requirements specification.

Only two very broad types of RE processes have been outlined above. In practice, there are many different models of the development process and each needs an RE process that is tailored to it. Despite the variety of RE processes and the range of

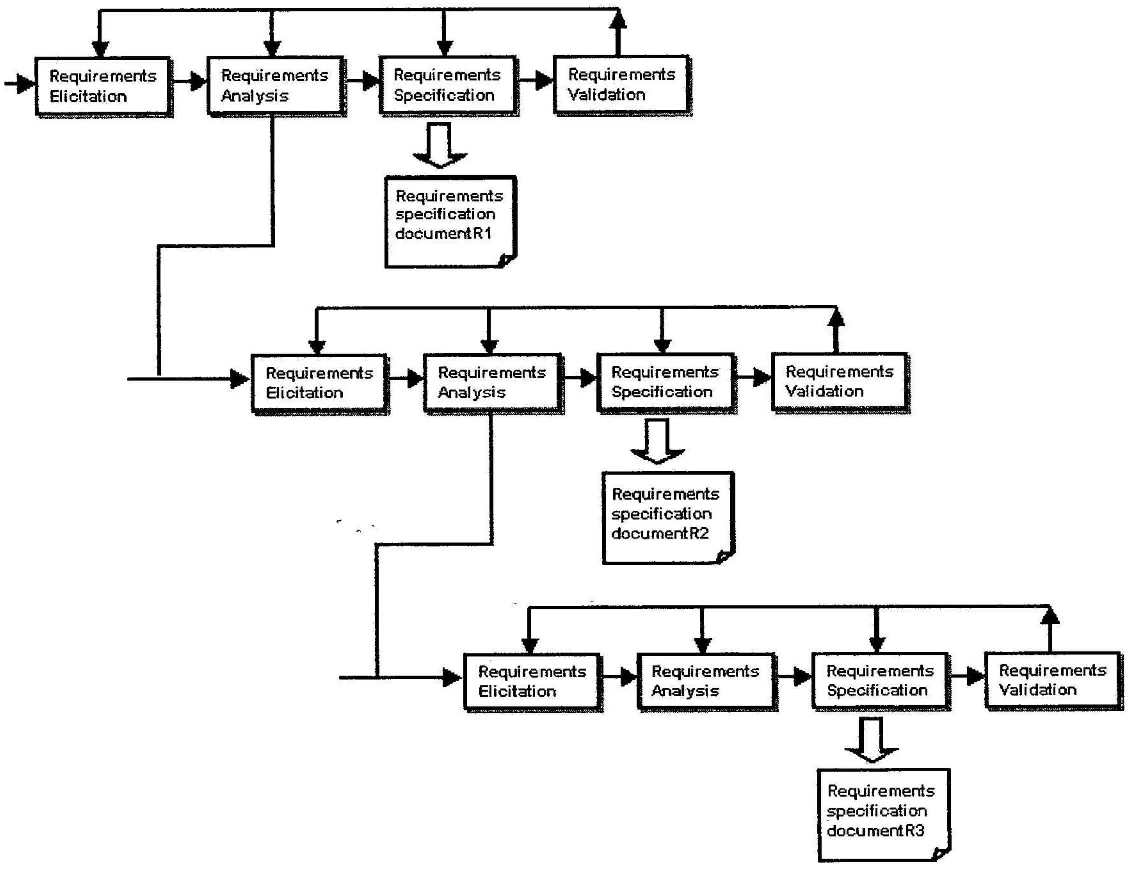

Figure 3. An incremental RE process.

practices used for elicitation, analysis, specification, and management, (see for example [Robertson 99, Sommerville 97, Wiegers 03, Young 01]) the essential goal of an RE process always applies. This is to derive and specify an appropriate solution to a business problem. To achieve this, requirements have to be discovered, understood, recorded, checked, communicated, and managed.

REQUIREMENTS ELICITATION

Requirements elicitation is the process of discovering the requirements. The requirements engineer needs to identify the sources of the requirements, collect information about the problem from these sources, and synthesize requirements from the information. As always, this is not a linear process but one that requires iteration as information is collected, clarified, corrected, and reformulated.

However, it is not simply about the requirements engineer passively learning about what the systems stakeholders require. The requirements engineer needs to dig beneath the stakeholders' accounts of their concerns, needs, and desires to uncover the underlying business problem that gives rise them. The requirements engineer learns about the problem but the stakeholders also need to learn about what can be achieved and the relationship between their requirements and those of other stakeholders.

The key point is that "elicitation"—the name routinely applied to this activity—is misleading. Requirements are not simply elicited fully formed from stakeholders. They have to be *discovered,* whether their sources are human, documentary, or the environmental. Another way to think of this is that requirements that will be documented in the specification are synthesized from the elicited requirements information.

Systems are normally commissioned to address some strategic goal or goals and these are the first things that have to be discovered. The requirements subsequently discovered must contribute to meeting these goals. When the project goals are known, the scope of the project can be defined. This may include management or process issues that are outside the RE process. An important element of the scope, however, is defining the system boundary. This is dealt with under requirements analysis. From the elicitation point of view, an explicit definition of the project scope helps ensure that elicitation of the requirements is focused on issues that are relevant.

Requirements Sources

A large system will have many different stakeholders and in most cases these will be the primary sources of the requirements. Indeed, any project that does not engage with the people who will actually pay for and use the proposed system is bound to fail.

Identifying the stakeholders is a crucial first step. The type of project is a factor in this. In a project developing a custom system for a commissioning customer, it will often be easier to identify representative stakeholders than if the project is developing a product to sell in a market. Stakeholders need to be classified in order to ensure that no significant sources of requirements are overlooked. The role of each stakeholder needs to be understood and representatives of each type of stakeholder must be selected to work with the requirements engineer. Most stakeholders are busy people, so it is vital to find people who are motivated to act as *product champions* [Wiegers 03] and to negotiate access to them.

Stakeholders have viewpoints. These are partial views of the problem domain that are colored by the stakeholders' own roles and experience. Inevitably, this means that their requirements may not be consistent with those of other stakeholders. Recognizing the scope and limitation of stakeholders' viewpoints is important. It can help the requirements engineer impose a separation of concerns on the set of requirements and so help resolve inconsistencies and apply priorities.

Stakeholders are not the only sources of requirements. Embedded software may have no direct users, for example. However, the system in which the software is embedded will have users and other stakeholders, and the embedded software requirements will be derived from these.

Key requirements and constraints often come from the application domain (burglar alarm systems may be subject to noise legislation, for example) and from the business rules that exist in the organizational environment (for defining usage privileges, for example). Key requirements may, therefore, be hidden in documents, interface specifications, and the experience of domain experts.

Domain expertise plays a crucial role in successful requirements elicitation. It is needed to identify the key requirements and constraints that are derived from the application domain. Although stakeholders have domain expertise, they are not an ideal choice to play the role of domain expert. This is partly because of their viewpoints, but also because they often find it hard to articulate the tacit knowledge that often underpins how the domain operates. The domain expert's role is to help make this tacit knowledge explicit.

The requirements engineers may have sufficient domain expertise if they have previous experience of the application domain. However, in an unfamiliar, complex domain, it may be infeasible for the requirements engineer to acquire domain expertise during the course of the project, even with training. In this case, domain expertise may have to be brought in from elsewhere to play an active role in the RE process.

Elicitation Techniques

Once the requirements sources have been identified. The process of collecting knowledge about the problem—the raw material from which requirements will be synthesized—can begin. This starts with understanding the problem. Only once the problem makes sense will the elicited requirements start to make sense. A key factor in understanding the problem is understanding the stakeholders' roles in the problem domain. In most cases, this means understanding their jobs.

Many jobs involve subtle facets and the interaction of people with different roles and expertise. Failure to recognize this can result in a system that fails to do its job. However, capturing the subtleties can be difficult because people often find it hard to articulate all the nuances of their job. Because of this, simply asking a user, "what do you need from the system?" will not work. The requirements engineer first needs to understand the problem and then find an effective way to get the stakeholders to analyze what they need. The requirements engineer will then be better able to reconcile any inconsistencies and make an assessment of what is needed and is feasible. In order to do this, the requirements engineer needs to find a means to tease out the information in a way that gives stakeholders some context within which to describe their role.

For sociotechnical systems, users' stories or scenarios can provide a valuable tool. Here, the requirements engineer gets users to identify their principal tasks. Each task is described as a sequence of events, noting the preconditions, postconditions, communications with colleagues, and other events that comprise the task. Use cases mesh well with scenarios. Here, use cases are simply top-level user tasks within the problem domain that are described in terms of scenarios.

Using scenarios helps the users articulate their tasks by getting them to describe the constituent activities. They help the requirements engineer by getting the users to focus on the essential characteristics of their tasks. The requirements engineer can better understand how jobs are performed and how they are organized within the work environment. Scenarios provide cues that allow the requirements engineer to ask for clarification at key points ("what happens if *<resource x>* is unavailable when you need it?").

Once the problem is understood, use cases can be used again to envision ways of interacting with the proposed system. Here, the use cases represent the top-level services offered by the system. Scenarios are then developed by the stakeholders and the requirements engineer for each use case to explore how it should operate, the interactions necessary to make it operate, and the exceptions to the normal sequence of events that have to be handled. The requirements implicit in the use case scenarios must then be made explicit.

Elicitation can proceed as a series of interviews between the requirements engineer and individual stakeholders but it is often useful to get the stakeholders together in one place. This is particularly useful once the requirements engineer has progressed beyond understanding the problem and has begun to elicit the requirements of the proposed system. Stakeholders' restricted viewpoints may lead them to propose requirements that are hard to reconcile with those of other stakeholders. For example, there may simply be too few resources available in the project to satisfy everyone's requirements. If run skilfully, elicitation workshops can provide an efficient way of resolving such problems by exposing stakeholders to others' ideas and concerns and by fostering a sense of collective responsibility for their resolution. Care must be taken to ensure fairness and counter phenomena such as "groupthink," however.

The use of scenarios and workshops does not preclude the use of other techniques and may not be the best techniques for every project. Some stakeholders may be unwilling or unable to participate in workshops or may find scenarios an awkward mechanism for articulating their requirements. Senior managers, for example may not interact directly with a system and so may find it hard to talk about the system in terms of scenarios. Traditional, individual interviews, in which the requirements engineer gets the stakeholder to state directly the properties and qualities required of the proposed system, may be necessary. Even here, however, the requirements engineer needs to understand (at least) the rationale for each requirement and the importance that the stakeholder attaches to it. Observation of users in their working environment may also provide revealing insights about how the work is actually done. For example, the physical layout of a control room may influence working practices in ways that can only be appreciated when directly observed.

The techniques discussed above emphasize the role of system users. Sometimes, however, access to users may be hard. When a product is being developed for a market rather than for an identified customer, it may be hard to identify representative users. At the very least, a user's champion must be identified. Competing products and emerging trends and technologies are also likely to be important sources of requirements.

Special attention must be given to discovering nonfunctional requirements. One important subset of nonfunctional requirements is that concerned with dependability. For systems that need high degrees of dependability, considerable resources need to be devoted to the discovery and analysis of the dependability requirements. For example, in safety-critical domains, a hazard analysis is normally used to identify the hazards and derive safety requirements to counter them. In systems with serious security implications, software security experts are needed to assess the security vulnerabilities and derive requirements.

Specialist requirements practices of the kind adopted for dependable systems is expensive and is economically impractical for most other systems. Nonfunctional requirements may nevertheless be very important for these systems. Stakeholders and domain experts will usually have an appreciation of the critical quality aspects of the problem domain. However, it is the requirements engineer who must recognize the required quality characteristics of the proposed solution. For example, manual business processes are often fault-tolerant because of the natural adaptability of the people who perform them. Automation can result in reduced dependability unless the appropriate reliability, availability, or other nonfunctional requirements are recognized and specified. A deep understanding of the implications of the proposed system is needed if the requirements engineer is to recognize such requirements.

REQUIREMENTS ANALYSIS

Requirements analysis is about understanding the problem and synthesizing a set of requirements that specify the best solution. Analysis is needed to help deepen understanding of the problem and what is required, and to detect and resolve problems such as inconsistencies and incompatibility with the requirements.

Elicitation and analysis are closely coupled activities. Requirements information has to be analyzed as it is elicited in order to understand, for example, what question to ask next. Among the questions to ask are *why* questions—why is it done this way? Deriving the best solution may involve challenging stakeholders' perceptions and ways of doing things.

Ultimately, the analysis activity should yield a *baseline* set of requirements. This is the set of requirements that the system (or this version of the system) will implement. Resource limitations often mean that the requirements baseline specifies a subset of the features requested by the stakeholders. The requirements engineer and the stakeholders have to agree on what is included and what is left out. Inconsistencies and conflicting requirements must be negotiated away and traded off. In the end, the requirements in the baseline should be necessary and sufficient. This means that the baseline should not include any requirements that do not contribute to the goals of the project, but it must be complete—there must be no requirements missing that are necessary to specify how the problem is to be solved.

The System Boundary

The first analysis activity takes place when the system goals are defined (see requirements elicitation)—the project needs to be scoped. The scope of the project may include things to do with, for example, the development process or procurement policies. However, the scope must also include a definition of the *system boundary.* This is concerned specifically with identifying which elements of the problem are to be addressed by the proposed system.

The system boundary should ensure that the proposed system focuses on satisfying the project goals. Outside the system boundary are the things in the system's environment that may impose requirements on or constrain the system. Within the system boundary are all the aspects of the problem for which the proposed system will provide a solution.

For example, in a project to develop a domestic burglar alarm system, all the functions concerned with protecting the house from intruders will be inside the system boundary. Outside the system boundary will be other systems in the alarm's environment with an interface to the alarm. For example, the alarm might be connected via a communications link to a security service. The functions to permit the security service to monitor the alarm would be inside the system boundary, so requirements for these need to be discovered and specified in the system specification. However, specifying the security service's reaction to a tripped alarm would be out of scope and so outside the boundary. There are a number of ways to depict the system boundary. One way is to use a use case diagram. Figure 4 depicts the system boundary for a burglar alarm system using a use case.

Defining the system boundary involves envisioning how the proposed system will operate in its environment. This is one of the main advantages of explicitly developing a concept of operations [IEEE 98a], which involves elicitation of the project goals, analysis of possible solutions, and definition of the system boundary. The concept of operations helps verify the project's viability and ensures that project participants understand what they are trying to achieve before commencing the main elicitation and analysis activities.

Requirements Modeling

In any RE project, there will be a lot to understand. To help make sense of complex information, engineers use models. Requirements engineers construct models of the problem so that they can discover suitable solution requirements. Models are also used to help describe the proposed system to help communicate the requirements to the developers.

There are many different modeling notations. In general terms, the same notations are useful for both problem and solution modeling. For example, an object model (or a class diagram in UML [Rumbaugh 99] terms) may be constructed to understand

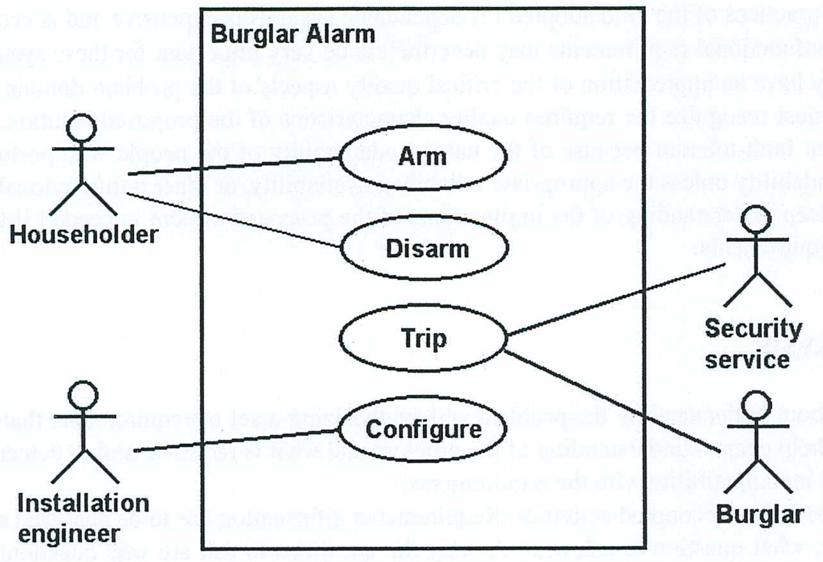

Figure 4. Use case defining the system boundary for a domestic burglar alarm.

the structure of a business activity just as an object model may be used to define the structure of a system that automates the activity.

Models help explore the requirements by providing alternative depictions of the problem to that provided by verbal or textual descriptions. Constructing a model helps the requirements engineer to make sense of the information as they transform it, using the rules of the chosen modeling notation. Graphical models help engineers visualize complex system properties through the use of appropriate abstractions. For example, a UML sequence diagram makes explicit the sequence in which events occur and the interactions between people or system components, in order to perform some task.

Although such models encourage rigor in the analysis, they do not enforce it. Formal specification notations such as Z [Davies 96] or CSP [Schneider 99] (or even UML's OCL) do enforce analytical rigor. Formal models are tractable to formal reasoning and enable some properties of system models to be proven.

Sometimes, the dynamic behavior of systems cannot be adequately analyzed using static models. In these circumstances, it may be necessary to conduct simulations. If uncertainty about systems' dynamic behavior affects their feasibility, then simulations must be conducted early in the requirements process.

Derived Requirements

The cost and technical implications of system requirements are often unclear and this makes them hard to assess and validate. To make the implications clear, it is usually helpful to elaborate the system requirements by deriving new requirements that focus on finer-grained properties of the proposed system. Modeling helps uncover the detail necessary for synthesis of the derived requirements.

In the burglar alarm example, the use case "arm" responds to the system requirement that the householders must be able to arm the alarm when they exit the premises. If the control panel is located inside the premises, the requirements engineer needs to derive the additional requirements that are an implication of this. In this example, it leads to a derived requirement that there must be a time delay between the user instigating the arming process and the alarm becoming armed. In a use case model, such a derived requirement would naturally emerge from writing the arm scenario(s):

Arm primary scenario
Precondition: Alarm is unarmed

- Householder initiates arming procedure
- Timer countdown commences
- Householder exits premises
- Timer countdown finishes
- Arming procedure completes

Postcondition: Alarm is armed

Of course, this is a very simple example and the synthesis of derived requirements is seldom so easy.

The organizational or operational environments may also yield derived requirements. Knowledge of the domestic (operational) environment may derive a requirement that the alarm must be protected against being armed accidentally by, for example, the family dog brushing against the control panel.

Understanding the implications of system requirements is one motivation for deriving more detailed requirements. The other reason is to add specificity for the developers. The requirements elicited from the stakeholders will typically be expressed in terms of the application domain. If the application domain is life insurance, for example, then the requirements are likely to be couched in terms of policies, investment vehicles, and so on. Every domain has its own culture and jargon and software developers may not understand these well enough to interpret the requirements correctly.

Essentially, the requirements need to be translated from being domain-centric to being software-centric; from abstract to technical. For example, knowledge about the cost and practicality of mechanical locks might lead to a requirement to use a PIN for arming the alarm being derived from the requirement to guard against errant family pets. This in turn might stimulate analysis to derive the length of the PIN required to provide adequate security without being an unreasonable burden on the householder's memory.

The derivation of requirements is not confined to functional requirements. High-level expressions of nonfunctional requirements need to be explored and have detail added too. They need to be quantified or transformed into a set of equivalent functional requirements. For example, a requirement that a system be reliable is hard to make rational design decisions for and impossible to verify. The requirements engineer needs to chose a suitable metric (e.g., mean time to failure) and specify how the system must score against this metric. Selection of a suitable value for a nonfunctional requirement against the selected metric may have a profound effect on the cost of satisfying the requirement and the requirements engineer needs to understand the practical limits to achieving high levels of, for example, reliability.

Requirements are not always derived top-down because some requirements emerge from the environment or the domain rather than as a consequence of properties requested by the stakeholders. For example, noise legislation may place a legal limit on the length of time that a burglar alarm siren can sound. Hence, a requirement that the siren must not sound for longer than this period when the alarm is tripped must be included in the specification. Sometimes, requirements patterns emerge in which an experienced requirements engineer recognizes a problem and is able to identify a known solution without having to iteratively derive and analyze successively more detailed requirements. The process of deriving requirements is driven largely by the requirements engineers' experience.

The derivation process should stop when the requirements are sufficiently specific for the requirements engineer to be confident that the requirements are fully understood, and for the developers to commence solution design. Getting too detailed risks leads to making premature design decisions and constraining the design unnecessarily. Inevitably, requirements derivation *does* involve the narrowing of design choice but this narrowing can be justified if the requirements engineer is the person best placed to make the choice. This was illustrated above by the derivation of a requirement that specifies the use of a PIN to guard the alarm, and another to specify the length of a PIN. Both these requirements constrain the design. However, the decision on the use of PINs needs to be informed by several factors that the requirements engineer is better able to assess than the developer with his or her exclusively technical focus.

There is an important housekeeping activity related to requirements derivation that is a the cornerstone of requirements management. Derived requirements must have the requirements from which they are derived recorded. Over the course of the project, a complete set of derivation relationships must be built up that permits the relationships between system and derived requirements to be traced (see requirements management).

Requirements Attributes

It is insufficient merely to record the statements of need that express the requirements. Requirements have a number of attributes that should be assigned values in order to ease their management. The analysis stage is where most of these attribute values are assigned. These attributes may include:

- **Identifier.** Every requirement must be assigned a unique identifier that allows it to be unambiguously referenced.
- **Source.** The source of the requirement may be, for example, a stakeholder from whom it was elicited or a higher-level requirement from which it was derived.
- **Date.** When the requirement was formulated.
- **Rationale.** The rationale explains the purpose of the requirement. This helps in subsequent analysis, particularly if the requirement is challenged or needs to be reworked at a later stage.

- **Type.** This attribute records, for example, whether the requirement is functional or nonfunctional; whether it is a user interface requirement, a safety requirement, and so on.
- **Priority.** When there are many requirements and a limited budget, choices must me made about what can be implemented and what has to be left out. The requirements engineer needs to impose some view of priorities on the set of requirements. IEEE Std 830-1998, for example, recommends classifying requirements as essential, conditional, or optional.
- **Stability.** Even if the requirements have been analyzed rigorously, uncertainty may remain about some requirements if, for example, the system's business environment is dynamic. The fact that uncertainty surrounds a requirement should be recorded so that its likely volatility is made explicit and appropriate risk containment measures can be taken.
- **Verification procedure.** This attribute defines how to verify that the requirement has been satisfied once the software has been implemented. If the verification procedure is defined at the same time as the requirement, it acts as a useful reality check on the requirements (see requirements validation below).
- **Status.** The requirement's status records its current position in the life cycle of the requirement (see requirements management below).

Requirement Trade-offs

The stakeholders need to approve the requirements baseline by accepting that some requirements will make the cut but others will not. Analysis will sometimes reveal that requirements from different stakeholders are valid but mutually inconsistent. Even more frequently, there will be insufficient resources available to the project to satisfy all the requirements. The stakeholders must be made aware of these conflicts and be explicitly involved in making the necessary trade-offs.

Agreeing on requirements' priorities helps the trade-off process but it is seldom possible to implement all the high-priority requirements because the cost of implementation has to be balanced against priority. In large projects, systematic analytical methods for informing the necessary trade-offs are needed (see, e.g., [Karlsson 97]). A good discussion of requirements priorities can be found in [Wiegers 03]. However, requirements priorities may sometimes be regarded merely as indicators of their proponents' willingness to negotiate.

Achieving agreement is often easier if stakeholders are aware of each others' concerns. If the stakeholders have been involved in elicitation workshops, this awareness may already exist and ease the process of forming a consensus. If not, workshops specifically aimed at achieving this consensus and deriving an acceptable baseline may be necessary.

SOFTWARE REQUIREMENTS SPECIFICATION

The software requirements specification documents the software requirements. Projects may use up to three kinds of "specification" document at different stages of the RE process. These broadly correspond to the IEEE standards reviewed above, which are designed to cope with large, complex projects. A concept of operations document sets out the project vision and scope, a system specification defines the requirements for the whole system, and a software requirements specification (SRS) specifies the requirements for a software component or subsystem.

Each document has a different purpose and each addresses successively finer levels of detail. The system specification must demonstrate how the envisaged system will address the problem for which it was conceived. It must be readable by the system stakeholders to enable them to validate the requirements and approve them (sign them off) as the basis for subsequent development. The SRS, by contrast, is primarily a technical document aimed at developers. Although its focus is on specifying the software, some motivation and context about the root business problem is always needed. Smaller projects may use a single specification document but it must still be capable of being validated by stakeholders and of communicating the requirements to developers in sufficient detail.

To serve its purpose, an SRS must have a number of qualities, some of which derive from the quality of analysis and some of which are housekeeping issues. Its primary job is to specify the system or software completely and unambiguously. The developer should not have to infer the requirements engineer's intentions or the stakeholders' needs.

In most specifications, requirements are written in natural languages such as English to make them understandable by developers and stakeholders. However, care is needed to write requirements in ways that are precise, concise, and with only one sensible interpretation. It is good practice to write requirements as single sentences, with the minimum number of conjunctions and using modal verbs consistently. Sometimes, restricted subsets of the natural language are imposed to restrict the vocabulary. The following requirement illustrates several of these characteristics:

Arm011	On completion of the arming sequence, there shall be a time delay equal to the escape period before the alarm enters the armed state.

Several features are illustrated by this example:

- The requirement's interpretation requires the terms *arming sequence, escape period* and *armed state* to be defined in a glossary.
- It is good practice to adopt a standard convention for using modal verbs. For example, the modal verb *shall* is normally used to denote an essential requirement.
- *Arm011* is the requirement's unique identifier (see requirements analysis). The mnemonics chosen for requirement IDs often reflect their derivation.

Despite the expressiveness of natural language, some features required of the system may be difficult to describe precisely. Formal notations such as Z and CSP (mentioned above in requirements analysis) provide precise descriptions (in addition to enabling reasoning and proof) because they have formal semantics. In general, however, the software industry has remained sceptical of the benefits of formal notations. Despite this, there are pockets of success, particularly in critical domains where the perceived additional costs can be offset against the real benefits that the additional rigor of formal specification can bring. Perhaps the most successful notations are those developed to address the particular needs of an application domain, and in which training, tools, and standardization efforts have acted to further cement the notations' success. A well-known example of this is SDL [Mitschele-Thiel 01], which is widely used in telecommunications domains.

In general, the requirements engineer should seek the most precise form of description that is consistent with readability and the overall coherence of the document (an SRS that used many different specification languages would be incoherent). This may mean supplementing textual requirements with other formal or graphical descriptions. Most SRSs, for example, include a number of key system models to supplement the textual requirements and help interpret them.

Many SRSs allocate a separate section for nonfunctional requirements. Requirements analysis must select suitable metrics for nonfunctional requirements and these should be used in the SRS. There is no point, for example, in specifying that a burglar alarm must be "reliable" or "very reliable." These vague, unquantified terms are of little help to the developer and make it impossible to verify the implementation against the specification. Instead, a suitable metric has to be selected such as the alarm's probability of failure on demand.

In many cases, functional requirements are derived from a nonfunctional requirement. For example, a quantified reliability requirement may appear in a system specification, whereas the corresponding SRS may instead specify requirements for a redundant architecture that will satisfy the reliability requirement.

As the formal record of the system specification, an SRS must be a structured document. It must record not only the requirements and associated system models, but should also include (as a minimum) contextual and supporting information; descriptions of constraints, conventions, and assumptions; and definitions of specialist terms. A development organization should adopt or adapt a standard SRS format for use in all its development projects to maintain continuity of experience and quality.

REQUIREMENTS VALIDATION

Requirements validation can be crudely characterized as ensuring correctness. The set of requirements specified in the requirements specification must accurately reflect what is needed to solve the underlying business problem, subject to the various constraints that act upon them. This concerns not just the correctness of individual requirements, but the correctness, completeness, and consistency of the specification as a whole.

The requirements must also conform to appropriate standards, guidelines, and conventions in order to ensure the readability, maintainability, consistency, and other important qualities of a specification document. Strictly speaking, this is about verification rather than validation since it concerns the requirements' conformance to a specification in the form of the standards, guidelines, and so on. Nevertheless, it is important since, for example, poorly documented requirements are hard to validate. IEEE Std 830-1998 provides a definitive guide to requirement qualities.

Validation is always applied to the requirements specification document or a final draft of it. However, a more informal kind of validation of the correctness of the requirements is typically performed during elicitation and analysis. Clearly, it is silly to defer all validation until late in the process.

Once it has been validated and any necessary changes made, the requirements specification is "signed off" on. Since the effect of this is to issue the document as the basis for design and implementation, both the document and the requirements it contains must be subject to formal change and version control from this point (see below). Not all projects are as cleanly organized as this implies, and sometimes development begins before the specification is signed off on (a formal signing off may not occur either). For example, an in-house project may be free of the contractual commitment implicit in a signing off. Similarly, the implementation of certain requirements may be safely anticipated without waiting for a formal commitment from the project sponsor. However, it is highly desirable to have a formal point in the process at which the requirements are frozen[2] and agreement sought from all the stakeholders. It is absolutely essential that change and version control be applied. The absence of a formal signing off considerably complicates these requirements management tasks.

In most cases, the requirements are validated statically. In some cases in which complex dynamic behavior is specified, the requirements may need to be validated dynamically using prototypes or simulations. These are usually costly, however, and a well-run project will anticipate the need to perform this kind of validation well before issue of the draft specification document.

Requirements reviews are a mechanism for validating requirements that should be applied in all cases. These may take the form of preparatory work in which the reviewers have access to the specification document, followed by a plenary meeting in which the reviewers and the requirements engineer examine key aspects of the requirements. The composition of the review panel is important and should include both development and stakeholder representatives.

Their task can be made easier by including checklists of things to look for (see, e.g., [Sommerville 97]). This can be particularly useful for checking the requirements specification's compliance with standards or conventions and for generic quality problems, such as weak or ambiguous requirements. A number of tools exist that can help check compliance of the SRS for quality guidelines. For example, NASA's ARM tool measures an SRS against NASA standards using a set of quality metrics [Rosenburg 98].

At some stage, the developed system will have to be verified for compliance with the requirements. This is not strictly an issue of requirements validation. However, writing tests cases against the requirements is a good way to validate their correctness and clarity. If it proves excessively hard to plan how a requirement is to be verified, then there is likely something wrong with the requirement. Although some types of nonfunctional requirements, such as those concerning security or reliability, are inherently hard to verify, they must be verifiable if they are to serve any useful purpose.

REQUIREMENTS MANAGEMENT

Weigers [Wiegers 03] defines requirements management as comprising four tasks: change control, version control, requirements tracing, and status tracking. A fundamental prerequisite for each of these is that requirements be uniquely identified (see software requirements specification above).

Requirements management is a crucial but often neglected task. Fortunately, significant improvements in industry requirements management practice were stimulated by the issue of the Capability Maturity Model for Software (SW-CMM [Paulk 93]) in the early 1990s. The SW-CMM identified requirements management as a Level-2 Key Process Area. This effectively meant that all development organizations that wished to demonstrate minimally effective project control and management had to adopt formal requirements management practices. This in turn stimulated a market for requirements management tools. These ease the housekeeping activities that make up a large part of requirements management.

Change Control

As noted earlier in this chapter, change (e.g., changes to existing requirements, the emergence of new requirements, changes to requirements' priorities) will occur, even after the SRS has been signed off on [Hutchings 95]. However, it is crucial that change is not permitted to occur without control. Requirements creep is a well-known phenomenon of software projects. Here, uncontrolled or ad-hoc changes to the requirements make project plans impossible to manage, inevitably resulting in systems that are late and over budget. If they ever reach service, they prove thoroughly unsatisfactory.

A request to change a requirement needs to be carefully assessed to determine the positive and negative effects of the proposed change. The outcome of this assessment may be a decision to accept or reject the change or perhaps defer it until a subsequent release of the software. Change control requires that there be a formal process in place to ensure that all the necessary information is available in order to make an informed assessment of the costs and benefits, and that the people who will form the panel to assess the change request are identified.

The factors that have to be considered by the assessment panel include how the change will impact on project schedule and costs. For example, the effects of a change to a system requirement will propagate through all levels of derived requirements and into the design and implementation components charged with implementing them. If implementation is already underway or (worse) completed, the change will be very costly. Of course, the cost implications of *not* approving the change must also be considered.

[2]This does not necessarily mean that subsequent changes to the requirements are prohibited. Rather, it means that from this point, change requests must go through a formal approval process.

Approved changes must be prioritized and the development schedule adjusted to accommodate them. This may necessitate delay to the delivery date or failure to implement other, less critical, requirements. Negotiation with key stakeholders and reference to the priority and rationale (see requirements analysis above) attributes of the requirements affected by a proposed change are crucial.

Version Control

Requirements changes should be recorded. This should include the details of the change, the date of approval, and the rationale for the change, including the rationale for the decision to approve the change. These notes should form attributes of all affected requirements so that an explicit record of the evolution of the requirements can be maintained for contractual and project postmortem purposes.

Changes need to be communicated to the development team and this requires the issuing of new versions of the requirements specification. A numbering system for document versions is essential. Several requirements management tools integrate a requirements database with fields for recording requirements and their attributes into document management tools. These can automatically generate new document versions with associated change histories.

Requirements Tracing

To enable change control and status tracking, requirements must be traced. As a minimum, this means that the derivation relationships between requirements should be recorded. Conceptually, the traced requirements form an acyclic graph. This is depicted in Figure 5, which shows the derivation relationships between requirements and the allocation of requirements to architectural components.

Requirements must be traceable in either direction. Forward tracing is necessary to assess the impact of a requirement down through its derived requirements and into the components to which they are allocated. In Figure 5, for example, a change to the system requirement at the bottom of the diagram propagates into two architectural components. The impact of this change needs to be assessed for each of the elided derived requirements and the affected components. Backward tracing is needed for cases in which, for example, a change is proposed (perhaps for pragmatic reasons) in a derived requirement or component. In this case, the impact has to be assessed based on the system requirements, their stakeholders, and, ultimately, on the system's ability to adequately solve the business problem.

Links between requirements and design are only a subset of the trace links that should be maintained. It is also necessary to trace, for example, requirements to their sources and from requirements to their verification plans [Gotel 95].

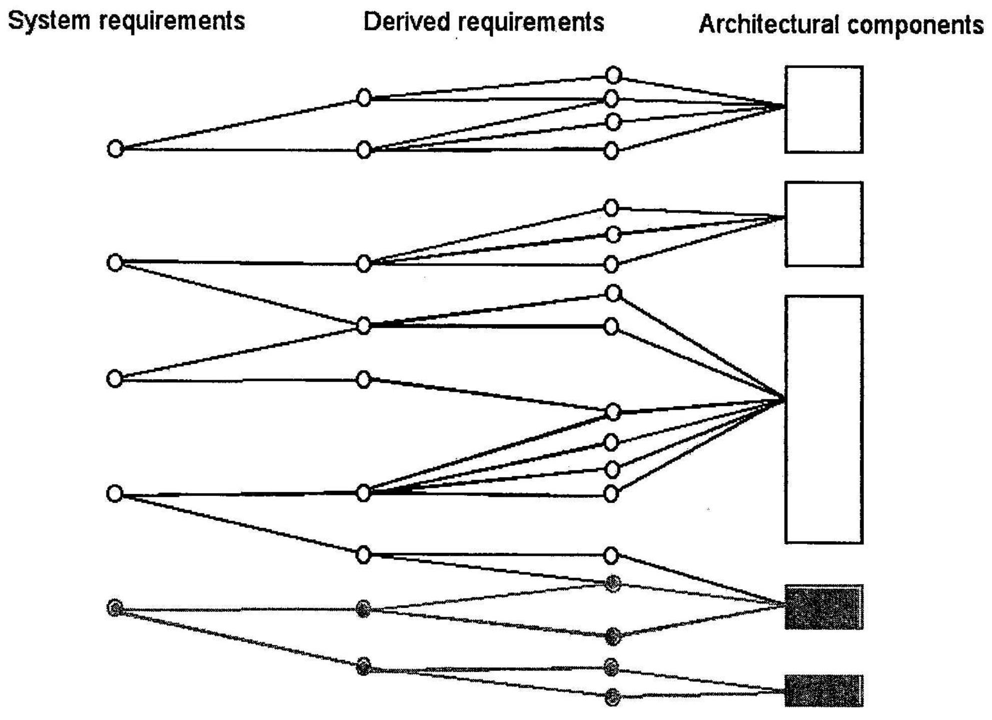

Figure 5. Requirements trace.

All requirements management tools maintain the relationships between requirements and allow the trace graph to be traced in either direction. In the absence of a tool, tables or matrices can be used. Maintaining these is an administrative overhead but tracing is absolutely fundamental to good requirements management.

Status Tracking

Status tracking concerns maintaining information on the processing and implementation of requirements. For example, requirements can be classified according to whether they are pending, approved, rejected, deferred (to a later release), validated, completed, or cancelled. Change requests pass through an analogous set of states. Managing a record of requirements' status is important for tracking project progress and for ensuring that decisions regarding the implementation and scheduling of requirements are carried out.

SUMMARY

Software requirements come from real business problems. How software requirements are derived from business problems is the problem addressed by RE, which is the subject of this chapter. RE has spawned many good textbooks and thousands of research papers in recent years but, inevitably, this chapter has provided only a skim of the topic. They key point, however, is that RE is a crucial part of any systems or software engineering process. It is the point at which the foundations are laid for a successful project. If neglected or done badly, it will ensure that the project subsides into a mire of misunderstood needs and uncontrolled changes.

RE is really part of systems rather than software engineering. However, the pervasiveness of software in the modern world and the ever-increasing demands made on software-intensive systems means that software is the focal point of critical investment by companies, governments and individuals. This has stimulated a great deal of research on RE by the systems and software engineering communities since the1970s.

This chapter gives a snapshot of the broad orthodoxy that has emerged during that time. New techniques, methods and tools constantly emerge and even the orthodoxy itself comes under fairly constant challenge. Proponents of agile development methods, such as eXtreme Programming (XP [Beck 00]), for example, argue that close interaction between developers and users, allied to small software increments, makes traditional requirements specification documents inappropriate. XP is geared to projects in which change is expected. It copes with change by scheduling delivery in small increments and developing the specification for each increment as needed rather than attempting to specify a complete system at the outset of the project. The requirements for each increment take the form of use cases and associated test cases. These form the increment's specification but they are not embedded within a formally structured document. XP accepts that, eventually, the accumulation of changes to the software will overwhelm the architecture, and require the software to be refactored.

There is an active and as yet unresolved debate about the efficacy of agile methods. They are a relatively new approach and, despite evidence that they work in circumstances where the requirements are volatile, there is still little experience of them in projects in which, for whatever reason, the requirements must be rigorously defined at the outset or where systems cannot be delivered incrementally. In most cases, therefore, an orthodox RE process is needed to derive and manage the SRS as the definitive record of the software specification.

The need to elicit, analyze, specify, validate, and manage requirements are axiomatic for any successful project but the practices that implement these and the way in which these are integrated into a software development process are constantly being examined and explored. It is essential, therefore, that software professionals and their managers maintain awareness of the fundamental precepts of RE and of good RE practice.

REFERENCES

[Beck 00] Beck, K.: *Extreme Programming Explained: Embrace Change,* Addison-Wesley, 2000.

[Davies 96] Davies, J., and Woodcock, J.: *Using Z: Specification, Refinement and Proof,* Prentice-Hall, 1996.

[Gotel 95] Gotel, O., and Finkelstein, A.: "Contribution Structures," in *Proceedings of 2nd IEEE International Symposium on Requirements Engineering (RE '95),* York, UK, 1995.

[Hutchings 95] Hutchings, A., and Knox, S.: "Creating Products Customers Demand," *Communications of the ACM, 38* (5). 1995.

[IEEE 98a] *IEEE Std 1362-1998, IEEE Guide for Information Technology—System Definition—Concept of Operations (ConOps) Document,* IEEE Computer Society Press, 1998.

[IEEE 98b] *IEEE Std 1233-1998, IEEE Guide for Developing System Requirements Specifications,* IEEE Computer Society Press, 1998.

[IEEE 98c] *IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements,* IEEE Computer Society Press, 1998.

[Karlsson 97] Karlsson J., and Ryan, K.: "A Cost-Value Approach for Prioritizing Requirements," *IEEE Software, 14* (5), September 1997.

[Mitschele-Thiel 01] Mitschele-Thiel, A.: *Systems Engineering with SDL: Developing Performance-Critical Communication Systems,* Wiley, 2001.

[Paulk 93] Paulk, M., Curtis, W., Chrissis, M. B., and Weber, C.: *Capability Maturity Model, Version 1.1,* IEEE Computer Society Press, 1993.

[Roberston 99] Roberston, S., and Robertson, J.: *Mastering the Requirements Process,* Addison-Wesley, 1999.

[Rosenburg 98] Rosenburg, L., Hammer, T., Huffman, L.: "Requirements, Testing and Metrics," in *Proceedings of 15th Annual Pacific Nothwest Software Quality Conference,* Utah, 1998.

[Rumbaugh 99] Rumbaugh, J., Jacobson, J., and Booch, G.: *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.

[Schneider 99] Schneider, S.: Concurrent and Real-time Systems: The CSP Approach, Wiley, 1999.

[Sommerville 97] Sommerville, I., and Sawyer, P.: *Requirements Engineering—A Good Practice Guide,* Wiley, 1997.

[Wiegers 03] Wiegers, K.: *Software Requirements,* Microsoft Press, 2003.

[Young 01] Young, R.: *Effective Requirements Practices,* Addison-Wesley, 2001.