Software Requirements

Jane Cleland-Huang

ABSTRACT

This paper describes the activities and work products that contribute to the specification and validation of the software requirements of a system. Although requirements practices are closely related to specific software development life cycle models, the general activities described in this paper are common to most process models. The activities of elicitation, analysis, specification, validation, and requirements management are discussed and recommended practices in each of those areas are highlighted. Characteristics of a quality requirements specification are also described.

1. INTRODUCTION

The process of eliciting, analyzing, validating, and managing requirements, often referred to as "requirements engineering" plays a critical role in the success of software development projects. Despite ongoing technological advances, an unsatisfactory number of projects continue to be delivered late and overbudget, or fail to provide all of the functionality needed by the stakeholders [1, 2]. The often-quoted Standish report [2] identified requirements related problems as a leading cause of failure, and, conversely, well-implemented requirements practices were seen as major success factors.

Unfortunately, there is no general consensus in either the literature or in practice for either a common requirements terminology or for a consistent requirements process. Actual requirements practices vary broadly from organization to organization, according to the culture of the organization, its maturity in implementing software engineering processes, and the domain in which the software is being developed. In too many cases, little effort is expended on the requirements process, which can result in construction of a product that does not meet its stakeholders' needs, in costly redesign efforts, and in projects that are abandoned despite large investment losses. IEEE standards such as IEEE Std 830-1998 provide guidelines for recommended practices. Similarly, the *Guide to the Software Engineering Body of Knowledge* (SWEBOK) [5], which was developed through extensive contributions from both software engineering practitioners and educators, provides another valuable resource for those who wish to establish or improve their current requirements practices.

Although many life-cycle models emphasize the requirements phase as an up-front activity, it is actually an iterative one that continues throughout the entire lifetime of the project [3, 4]. Requirements drive not only the initial design and validation of the system, but support ongoing activities such as change management, regression test selection, and compliance monitoring. The requirements process can be described by the five primary disciplines of requirements elicitation, analysis, specification, validation, and management. Although there is overlap between these activities, they are described here as individual phases for pedagogical purposes.

- **Elicitation** is concerned with proactively working with stakeholders to discover their needs, identify and negotiate potential conflicts, and establish a clear scope and boundaries for the project.
- **Analysis** involves gaining a deeper understanding of the product and its interactions, identifying requirements with global impact in order to define the high-level architectural design, allocating requirements to architectural components, and, finally, identifying additional conflicts that emerge through considering architectural implementations and negotiating agreements between stakeholders.
- **Specification** involves the production of a series of documents that capture the system and software requirements in order to support their systematic review, evaluation, and approval.
- **Management** of requirements is an ongoing activity that starts from the moment the first requirement is elicited and ends only when the system is finally decommissioned. Requirements management includes software configuration management, traceability, impact analysis, and version control.
- **Validation** occurs throughout the other four activities. It involves ensuring that the product meets stakeholders' requirements through activities such as formal and informal reviews and, for more complex or critical systems, through the use of formal verification techniques.

Activities across these disciplines are tightly coupled and, therefore, there is significant overlap and iteration between them. Figure 1 illustrates the iteration that occurs between various requirements activities and depicts the ongoing progress toward a validated requirements specification.

Although the related literature uses the term "requirements engineering" to broadly describe the requirements process this is not a term frequently found in industry. As the requirements process, is just one aspect of software engineering, it does not in itself result in the delivery of a fully engineered product. In this paper we therefore follow the practice adopted in the *Guide to Software Engineering Body of Knowledge* (SWEBOK), in which the term "requirements engineer" is replaced by the general term "software engineer" [5]. Additional terms such as elicitor, analyst, and specifier that depict specific roles of the requirements process are also used.

The following section provides a more formal definition of a requirement and then the remaining sections discuss each of the specific activities of elicitation, analysis, specification, validation, and management of requirements.

2. DEFINING A REQUIREMENT

A requirement is simply a property of the system or a constraint placed either upon the product itself or upon the process by which the system is created [4, 6, 7]. More formally, IEEE Std 610.12-1990 defines a requirement as

> (1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation of a condition or capability as in (1) or (2). [8]

As illustrated in Figure 2, product requirements can be either functional or nonfunctional and describe properties of the actual system that is to be delivered. A functional requirement (FR) describes what the system needs to do, such as a requirement for an online banking portal specifying that "The system shall display the current customer balance." In contrast, a "nonfunctional" requirement (NFR) describes a constraint upon the solution space, capturing a broad spectrum of systemic qualities such as reliability, portability, maintainability, usability, safety, and security [6, 9, 10]. Because many NFRs can actually be refined into functional requirements, many people prefer to call them "quality" requirements, "ilities," or even "systemic" re-

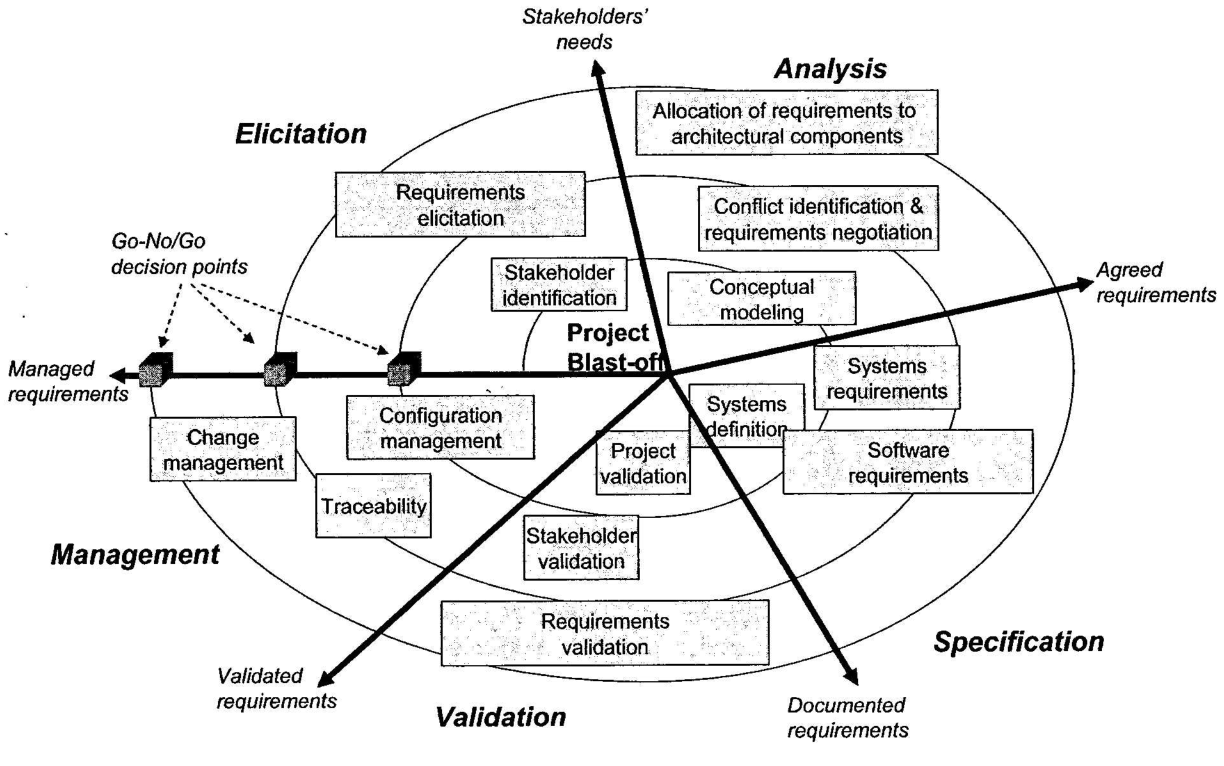

Figure 1. The requirements process.

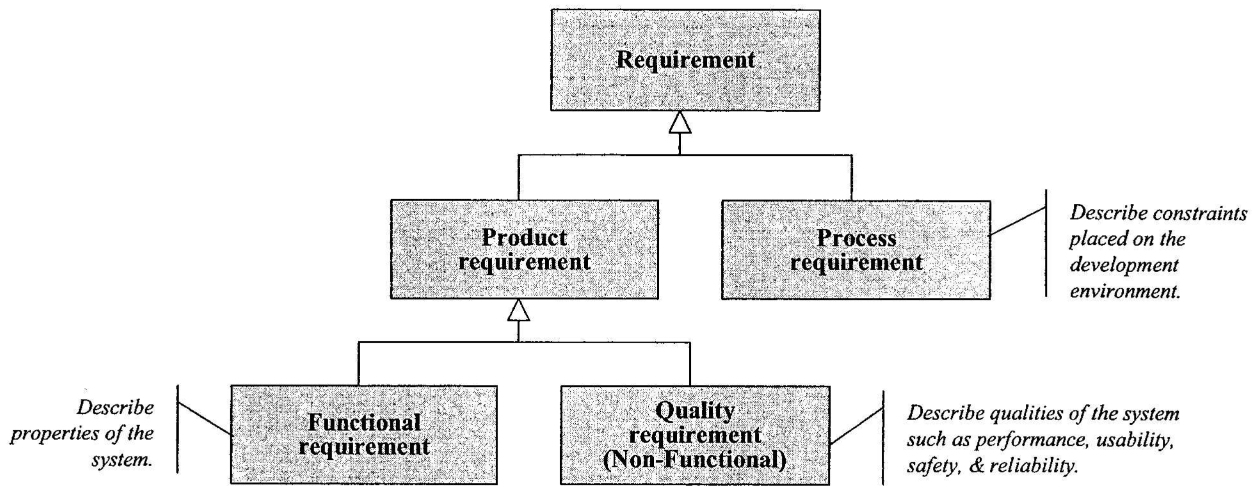
Requirement
Product requirement
Process requirement
Describe constraints placed on the development environment.
Describe properties of the system.
Functional requirement
Quality requirement (Non-Functional)
Describe qualities of the system such as performance, usability, safety, & reliability.

Figure 2. Types of requirements.

quirements. Often, NFRs can not be directly implemented as stand-alone functions, but are realized through the careful implementation of other requirements on which they depend. For example, a requirement stating that a specific query must return its results in less than three seconds is only realizable once the architecture and much of the system functionality has been implemented.

Process requirements specify constraints placed upon the development process. For example "The system shall be developed to run on the J2EE platform," or "Model Checking shall be used to formally validate the correctness of the security protocol." Process requirements can be defined either in a statement of work (SOW) or in a special section of the software requirements specification [9].

3. ELICITATION

Requirements elicitation focuses on gathering knowledge about the needs of the stakeholders by helping them to understand and articulate their problems and, where possible, by describing their vision of what they would like the new system to do. As such, it is a process of discovery, and represents one of the more critical aspects of the requirements task [11].

There are several dimensions to requirements elicitation [4]. These include understanding the problem and its domain, identifying clear business objectives for the project, and, finally, understanding the needs and constraints of system stakeholders.

3.1. Understanding the Problem and Its Domain

The first step in the elicitation process occurs at the very start of the project during an activity sometimes referred to as "project blastoff" [12]. During this phase of the project, the problem domain is explored in order to understand the context in which the proposed software application will execute. The task can be simplified by breaking down the domain into subdomains [13]. As illustrated in Figure 3, a preliminary breakdown of an emergency dispatch center identifies subdomains such as GPS tracking, emergency services such as police and fire services, and call dispatch. Robertson refers to subdomains as "adjacent systems," because they are the systems (whether manual or automated) with which the proposed application will interact [9, 12].

The requirements elicitor works with an initial group of stakeholders to identify the subdomains of the problem. Once these subdomains have been identified, an additional set of stakeholders or "subject matter experts" (SMEs) are selected to explore each one more fully. As a word of caution, it is imperative to find the right set of stakeholders. If an entire group of stakeholders are accidentally or deliberately omitted from the process, their needs may not be adequately considered, which may ultimately lead to their failure to support the project. Alexander and Robertson identify stakeholder roles through the use of an "onion model" [14]. The center of the onion represents the product to be developed, whereas the outer layers represent progressively more distant types of stakeholders. These include direct operators of the system, functional beneficiaries, and, finally, stakeholders such as political or financial beneficiaries. Taking a systematic approach to stakeholder identification reduces the likelihood of missing critical perspectives of the problem and related stakeholders, which, in turn, could have a catastrophic effect on the success of the project [15, 16]. Furthermore, successful elicitation is facilitated if the selected stake-

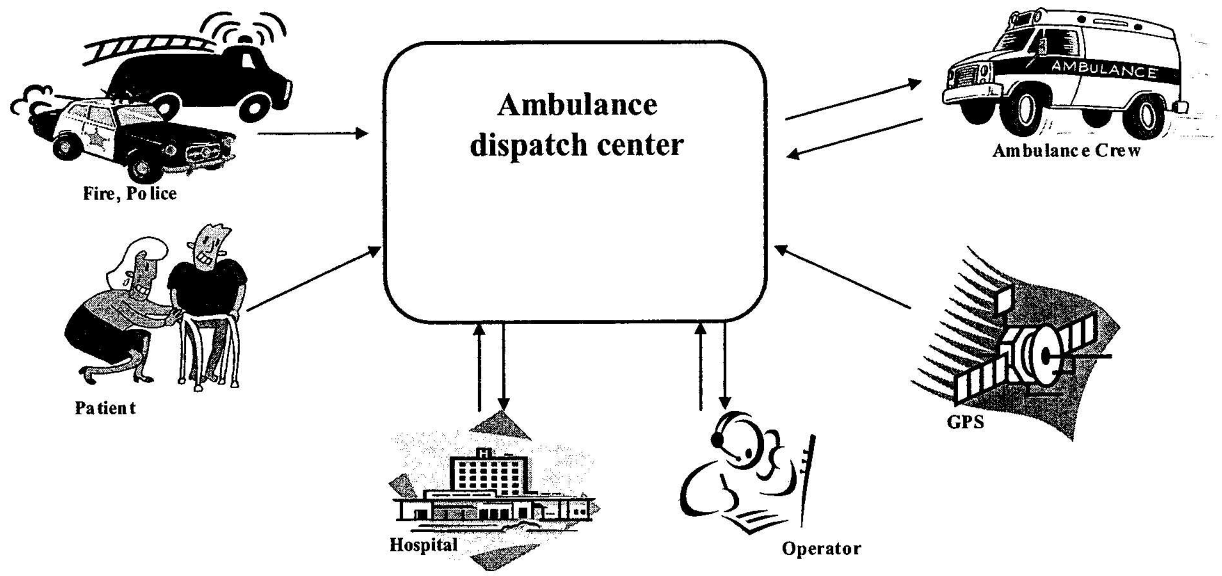

Figure 3. Defining the context of the work.

holders are representative of a specific group of people, empowered to make decisions for that group, able to work collaboratively with other stakeholders, and knowledgeable in the subject matter they represent.

Once stakeholders have been identified, the requirements elicitor must explore the problem that is to be solved. In addition to relying upon stakeholder knowledge, the software engineer should ideally acquire domain knowledge in order to more fully understand the needs of the stakeholders, whether those needs are articulated or not [9]. This is especially important if the product is being developed in a domain for which the typical users are not highly computer literate. It is the elicitor's job to initially steer stakeholders away from offering premature solutions until the problem space is well defined [14]. The new product should then be modeled within the context of its adjacent systems, showing the information flows that occur between them [12]. These work flows clearly define the boundaries of the new product and identify events that will trigger responses and that must be further explored during the elicitation process.

3.2. Making the Business Case

Current trends demonstrate that organizations are no longer willing to invest in IT projects unless those projects return clear value to the business [17, 18]. Prior to committing to a project, the customer and business stakeholders should perform a business analysis [18] to more fully understand the costs, risks, and anticipated benefits from the project.

During this phase, the high-level requirements are also examined for risks. The results of the risk and benefits analysis provide the basis for determining whether the project should proceed or not. We should point out, however, that this decision does not necessarily need to be an "all or nothing" decision. For example, in an extensive case study examining the catastrophic failure of the London Ambulance System in the late 1990s, Finkelstein observed that similar systems of smaller scope had been successfully and incrementally delivered to other regions in England [19]. One of the identified causes of failure had been the sheer scope of the project.

There are many risks that can impact the success of a project. From a requirements perspective, these include a lack of clear purpose for the product, insufficient stakeholder involvement, lack of agreement between stakeholders, rapidly changing requirements, goldplating (adding additional and unnecessary features), poor change management, and lack of analysis of the requirements [2]. At this stage, risks can be mitigated through awareness of these potential causes of failure, the definition of a clear problem statement and rationale with commitments from all major stakeholders, and a proactive risk-mitigation plan that includes processes to control change.

As many of the major documented cases of project failure trace their problems back to failure to create a clear and agreed upon statement of work during this early stage, a project should clearly *not* proceed without this agreement in place.

3.3. Elicitation Techniques

Once the boundaries of the work have been defined and stakeholders have agreed to proceed, the hard work of understanding users' needs can start in earnest. It is helpful to break down the problem into smaller and more understandable units such as use cases [9, 11, 20] or specific workflows [21]. Elicitation then focuses upon understanding the users' needs in respect to these smaller units of work.

The role of the elicitor is to learn the needs of the users and to communicate these needs effectively to the developers. There are many different elicitation techniques, and a general consensus exists that there is no single method that is universally the best one. The correct approach is dependent upon the nature of the system to be developed and the background and experience of the stakeholders [5]. Some of the more typical methods used include collaborative sessions, interviewing, ethnography, questionnaires, checklists, role playing, modeling, and prototyping [22].

Collaborative sessions come in all shapes and sizes and are primarily useful for brainstorming and problem-solving activities. For example, a joint application design (JAD) session can be useful for bringing a small group of stakeholders together to form the initial goals and requirements of a system. It is a useful technique for setting initial goals. Collaborative methods are also useful for identifying and negotiating conflicts that might exist between requirements. These methods are discussed in greater detail in the section on requirements analysis.

Interviewing techniques are one of the simplest yet most effective methods of requirements elicitation. Interviews can either be structured around a specific set of questions or be open-ended, with the intention of gathering as much useful information as possible. In most cases, both techniques are used in a single interview. Structured interviews have the advantage that all interviewees are asked the same questions and that critical questions are not inadvertently forgotten. In an unstructured interview, the interviewer may ask a few leading questions but then allow the interview to develop in a less rigid fashion. This approach may unearth entirely new areas of discussion that had previously been overlooked. As both methods have their own advantages, it is often beneficial to combine both techniques in a single interview. Interviews are often conducted in stages, so that responses from the first round can be used to generate a deeper set of more focused questions for the second round. It is often useful to target the second round of interviews to stakeholders with specific responsibilities or interests related to the more targeted questions.

Questionnaires can also be useful if it is possible to formulate a very specific set of questions. This usually is only possible when the problem is quite well defined up front. Questionnaires tend to be used more frequently in the form of market research surveys when developing a product for an external client, or to elicit a general response from a targeted group of stakeholders such as the users of an existing system.

Ethnography involves observing the way users interact with an existing system. This is particularly useful when users are unable to fully articulate their needs, or are too busy to attend other types of elicitation meetings. Studying how a user currently performs a task, and noting problems and possible areas of improvement, can lead to identifying real user requirements that might otherwise have been missed. It can be particularly enlightening to observe shortcuts and work-arounds that may have been developed by power users, because these often offer insights into their real needs.

Prototyping is a useful technique for taking an early set of user requirements and rapidly building a "system" that can be used to elicit additional requirements. There are various types of prototypes. Low-fidelity models are built with pen and paper, index cards, Post-it notes, and so on, and are exceptionally useful because for very little cost you can obtain useful feedback from the user. They have the added advantage that the user feels comfortable making suggestions because the prototype does not look like a final product. Higher-fidelity prototypes that utilize rapid development techniques to deliver a semifunctioning product to the user can also be useful for eliciting feedback. Through interacting with something that looks like the final product, the user often identifies additional requirements or discovers areas in which the product does not do what they had intended it to do.

Documentation can provide significant insights into possible requirements. These come in a variety of shapes and sizes such as problem reports, memos, user manuals from existing systems, existing designs and specifications, reports output from existing systems, documentation from competitors' products, and even previously written contracts [15].

Modeling can also be used during the elicitation process primarily as a means of communicating back to the user the specifiers' understanding of their needs. A broad range of methods such as data flow diagrams (DFD), statecharts, use cases, and sequence diagrams are available. As the primary purpose of modeling at this stage is to support the thought process and to serve as a communication aid between users and elicitors, the selected models must be easily understood by the stakeholders. A model is useful during elicitation *if* it helps the elicitor to figure out which questions to ask, or if it brings hidden requirements to the surface [12]. In general, formal models are not that useful during the elicitation process [23], primarily because they are typically not well understood by stakeholders.

Roleplaying or use of surrogate techniques can be used to explore stakeholders needs when those stakeholders are unavailable. This is particularly useful, for example, if you are developing a product that will be mass marketed and you do not know who the actual users will be.

Checklists of NFRs can be used to help stakeholders identify the nonfunctional needs of the system. It is often much easier to think about what the system needs to do than to identify its critical qualities such as performance, usability, and security, and so on, therefore, a checklist is a useful tool for triggering discussions in this area. As many postmortem analyses of large system failures have identified NFRs as the primary cause of the failure, it is imperative to consider NFRs during the elicitation process.

3.4. Conflict Identification and Negotiation

The requirements elicitor is also responsible for identifying inconsistencies and unresolved issues in the gathered requirements. There are two primary sources of conflicts that must be dealt with. First, stakeholders may have conflicting ideas concerning the functionality of the new system. These types of issues can be minimized through clearly defining the scope of each stakeholder group, and can be resolved through identifying and negotiating solutions to conflicts as they occur. Having an empowered project manager or "champion" who can guide the project through sometimes muddied waters can be very beneficial for resolving these types of conflicting needs. Some conflicts, such as those related to nonfunctional requirements such as performance, cost, and security, may not be unearthed until the requirements analysis phase, when candidate architectural solutions are considered. The objective, however, is to reveal conflicts as early as possible so that they can be resolved in a timely manner and accommodated within the requirements specification. In most projects in which requirements are written textually rather than formally, conflicts are identified through a qualitative review of the requirements. It is not an overstatement to point out that numerous projects are cancelled simply because stakeholders cannot reach agreement about what the system should do.

4. REQUIREMENTS ANALYSIS

During the requirements analysis phase, the emphasis is on gaining an understanding of the product to be developed through requirements classification and conceptual modeling. During this stage, it is important to classify requirements according to priority and scope, consider candidate architectures, allocate requirements to components, evaluate the impact of the architecture upon the requirements, and identify and negotiate architecturally related tradeoffs.

4.1. Conceptual Modeling

The purpose of modeling changes during requirements analysis. Whereas earlier models were primarily used to elicit further requirements, now they are used to gain a deeper understanding of the requirements. There are several types of models that are useful, including data and control flows, state models, event traces, object models, and user interactions. Each of these models is applicable in different situations. For example, it is useful to model real-time systems using control flow and state models, and dataflow diagrams are very useful for representing the flow of data between external entities and systems in business oriented systems. Selection of a specific modeling notation is dependent upon many factors, including the nature of the problem domain, the expertise of the software engineer performing the modeling, process requirements established by the customer, and availability of supporting tools. There is no clear evidence that any particular modeling notation is generally superior to all others; however, there is an advantage to using a widely accepted industry standard such as UML, simply because it is well known and understood by a broader range of stakeholders [5]. IEEE defines two standard notations for conceptual modeling. These are IEEE Std 1320.1, IDEF0 [24] for functional modeling, and IEEE Std 1320.2, IDEF1 X97 [25] for information modeling.

4.2. Architectural Design and Requirements Allocation

The analysis phase is tightly interwoven with the high-level architectural design of the system [26]. During this stage, requirements are categorized in order to differentiate between process and product requirements and also to identify requirements that may assert more influence upon the architectural design. Special attention needs to be paid to nonfunctional requirements, as many of these have a global impact upon the system and, therefore, exert a strong influence on architectural decisions.

Architectural quality is measured by its ability to fulfill the stated requirements. There are several techniques such as the Architectural Trade-off Assessment Method (ATAM) [27, 28], that can be used to assess this fit. ATAM evaluates the ability of an architecture to fulfill the requirements. Similarly, the NFR framework [29] provides a framework for reasoning about trade-offs between requirements and for assessing the impact of various implementation decisions upon the NFRs. Evaluating architectural quality during the elicitation and analysis process provides insights into conflicts, trade-offs, and missing requirements, and ultimately leads to the development of a higher-quality set of requirements.

Once a high-level architecture is defined, requirements can be allocated to components, thereby triggering a further round of elicitation and analysis.

5. REQUIREMENTS SPECIFICATION

The requirements specification is a document that describes the system to be developed in a format that can be reviewed, evaluated, and approved in a systematic way [5]. For large and complex systems in which the software component is just one of many parts, three distinct documents, depicted in Figure 4, are typically needed. These are the systems definition document, systems requirements document, and the software requirements document [6]. In contrast, software-intensive applications with no major hardware components are normally described fully in the software requirements specification.

The systems definition document, which is commonly called the user requirements document or the concept of operations (ConOps) is often written using domain terminology and defines the high-level system requirements from the domain perspective. It provides background information about the general objectives of the system, lists any constraints and assumptions, identifies critical nonfunctional requirements, and generally depicts the system context in which the proposed system will operate. Conceptual models of the domain normally depict interactions with adjacent systems, specifically identifying any events that the system must respond to. Primary data stores can also be identified.

The second document, the systems requirement specification, is typically only used in systems with substantial nonsoftware components such as an embedded airplane system. Development of this document, which is actually a systems engineering activity, enables the separation of systems and software specifications. Typically, the software requirements are derived from the systems requirement specification and must specify the interfaces between hardware and software. IEEE Standard 1233 defines the process for developing system requirements [30].

The third document, the software requirements specification (SRS), defines what the software component of the product is expected to do, and, where necessary, explicitly states what it should not do. It describes functional requirements in terms of all the inputs and outputs to the system and the functionality that must be provided to transform those inputs into the outputs. It additionally describes the nonfunctional requirements that have been negotiated and agreed upon by the stakeholders. A supporting document known as the requirements definition document provides clear definitions of all the terms used in the specification. The SRS is normally written in natural language but complex or critical requirements may be more formally specified [5].

The SRS is used to identify risks, estimate cost and schedule, drive the design and implementation of the system, and act as a contractual agreement to support eventual customer acceptance of the product. The SRS is created as a hierarchical document, including an introduction, overall description of the product describing constraints, assumptions, and dependencies, and a section in which specific requirements are described. Typically, requirements are organized by type, such as external interfaces, functional requirements, performance requirements, design constraints, and other quality requirements [9]. Many organizations adopt standard templates for organizing the SRS and structuring requirements attributes. These templates are useful because they clearly define the sections of the SRS and the structure of each individual requirement.

Because of the criticality of the SRS, various standards such as IEEE Std 830-1998, IEEE Std. 1233, and IEEE Std 1362-1998 define the structure and requirements of the document.

5.1. Qualities of an Individual Requirement

To minimize errors that are introduced during the requirements phase, each requirement should be written to exhibit the following qualities: [31, 32, 33, 34]

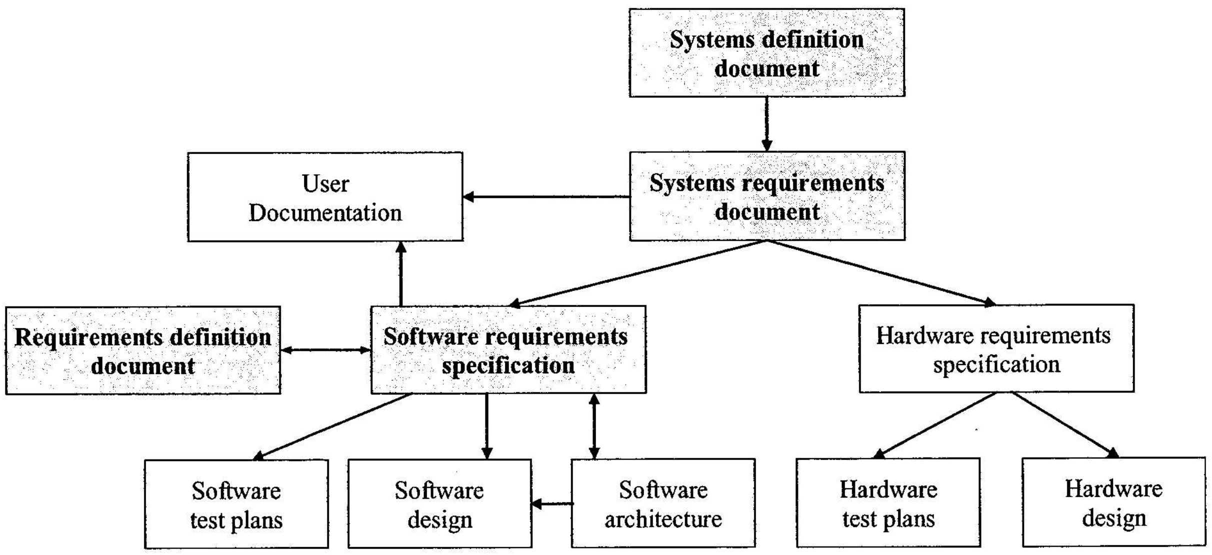
Systems definition document
User Documentation
Systems requirements document
Requirements definition document
Software requirements specification
Hardware requirements specification
Software test plans
Software design
Software architecture
Hardware test plans
Hardware design

Figure 4. Requirements documents.

- **Concise.** A requirement should describe a single property of the desired system and should include no information beyond that necessary to describe the intended property. It should be stated in clear, simple, and understandable terms. Whenever a precondition or constraint is applicable to a single requirement, it can be attached as a constraint on that requirement. However, as frequently occurs, a group of requirements may share a set of constraints. In this case, the constraints should be stated at a higher and shared level of the requirements hierarchy.
- **Correct.** A requirement should accurately describe the intended property of the system, with no information missing that is needed to define or implement the system.
- **Nonambiguous.** A requirement should be stated clearly and understandably, in order to avoid ambiguous interpretations. Although common language usage sometimes encourages ambiguity, this can be reduced to a minimal level by making all terms in the requirement explicit and through use of a project glossary to clearly define terms.
- **Feasible.** A requirement should be feasible from technical, financial, and managerial perspectives.
- **Verifiable.** A requirement should be written in such a way as to provide a clear and testable acceptance criterion. For example, it is not sufficient to state that a "query must return a fast response time." Instead, the requirement should be written in a form such as the "query must return a response within 1 second 90% of the time, and within 3 seconds 99.9% of the time." Even requirements stated in such clear terms may be hard to verify and may need to be tested through simulations, runtime measurements, or, in the case of usability requirements, through structured usability studies. To remove any ambiguity, the verification method may also need to be agreed on contractually in the requirements document.

In addition to these qualities, it is often useful to attach attributes to each requirement in order to manage them more effectively through tracking priorities and current status.

One additional point to keep in mind when writing requirements is to ensure that the requirements state the needs of the user and do not contain unnecessary design constraints. For example, a requirement for a computerized stopwatch might say that the "timer shall be reset by the user," but unless the customer has explicitly introduced the button as a constraint on the design, the requirement should not state that the "user shall click on the reset button." This would prematurely assume that the reset option will utilize a clickable button, and prematurely introduces a design element that can limit creativity and place unnecessary constraints on the finished product.

The writing style of a requirement is also important. Although there is some variation in standards between organizations, a general guideline is that all requirements describing a mandatory system property use the words "shall" or "must." Words such as "will" are generally reserved to depict events that will happen in the future and are not used to describe properties of the system. Certainly words such as "ought to," "should," "would," "might," and "may" do not belong in a requirement because they immediately introduce the idea of an optional feature and therefore do not result in contractually binding requirements.

5.2. Qualities of the Set of Requirements

In addition to these individual characteristics, there are a further set of qualities that must be applied to the requirements as a whole. The requirements should be:

- **Realistic.** The requirements should represent realistic goals at both the product and project levels.
- **Concise.** The requirements should concisely describe the system that is to be developed. An excessive number of requirements create greater opportunity for inconsistencies and errors.
- **Complete.** The requirements should collectively describe the entire system to be implemented, with no information missing.
- **Consistent.** Inconsistencies between requirements lead to conflicts that prevent all of the requirements from being implemented successfully. Inconsistencies should be identified and conflicts negotiated.

6. VALIDATION

Validation falls under the general heading of V&V or verification and validation. Again, this is an area in which there seems to be some ambiguity about the meaning of the individual terms. IEEE Sandard 1012-1998 defines requirements validation as the process of evaluating an implemented system to determine whether it conforms to the specified requirements [35]. However, this definition does not take into account the fact that the specified requirements may fall short of capturing the real needs of the stakeholders. The SWEBOK defines validation as the process of ensuring that the engineer has understood the requirements correctly, in other words, "Have we got the right requirements?" On the other hand, verification is defined as the process of ensuring that the requirements documents conform to specified standards. Verification addresses the question of "Have we got the requirements right?" [5]. Perhaps rather wisely, many organizations simply include all the activities aimed at ensuring that the software will function as required under the single umbrella of V&V.

Validation practices should be built into every stage of the requirements process in order to ensure a quality product. Studies [36] have shown that errors introduced during the requirements stage are the most costly to repair because of their far-reaching implications for the system. Furthermore, as depicted in Table 1, it is generally accepted that the relative cost to repair a software error progressively increases at later stages of the lifecycle model, thereby underlining the importance of early V&V activities. Typical methods include reviews, prototypes, models, and acceptance tests [37].

Reviews are conducted by stakeholders with the intent of finding errors, conflicts, incorrect assumptions, ambiguities, and missing requirements. Formal inspections and reviews have been shown to be effective in removing errors early in the process, thereby reducing the cost and effort that would have been involved in fixing downstream problems [37, 38]. It is important to have customer and user representatives as well as developers involved in the review process so that all perspectives can be considered. Reviews are useful at all major milestones in the delivery of the requirements documents, including completion of the system definition document, system requirements document, SRS, and prior to all major baselines. All reviews should result in a list of identified problems and a set of agreed upon actions. As reviews require significant time commitments, they can be costly to conduct, and it can be beneficial to perform prereview activities to identify and handle obvious errors in advance. Furthermore, review documents should be distributed and read in advance of a meeting so that all members can arrive well prepared.

Prototyping is useful for validating the software engineer's interpretation of the users' needs. Stakeholders provide more useful feedback when interacting with a prototype than when they simply read an SRS. In fact, requirements developed with the help of a prototype tend to be less volatile than those developed without one.

Model validation is used to verify the correctness of the system. Conceptual models can be formally or informally validated, either by statically analyzing the model or, in the case of formal specifications, by applying formal reasoning to analyze the properties of the system. In critical systems, it has been found that the activity of formally modeling the system can in itself serve to identify errors such as ambiguities and conflicts; however, the cost of creating formal models can normally only be justified for high-assurance systems.

Acceptance tests are used to validate that the completed product fulfills the requirements of the system. All requirements, including nonfunctional ones, must, therefore, be specified in a way in which they can be validated.

7. REQUIREMENTS MANAGEMENT

Almost every software product continues to change and evolve throughout its lifetime. If change is not managed well, the quality of the product will deteriorate and future changes will become increasingly difficult to accommodate. Change management is concerned with carefully controlling changes to the requirements, both during the development process and following the product's deployment. Change management is supported through requirements traceability, managing the current status of all requirements, and through placing requirements under configuration control. Measuring the volatility of the requirements in a project can provide useful insights into the overall requirements process.

Requirements traceability is defined as "the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases)" [39]. A trace defines a relationship between two artifacts. For example, a vertical trace between a lower-level requirement and a higher-level one could define a "refines" relationship, whereas a trace from an executable method to a requirement could define an "implements" relationship. Typical traceability techniques include matrices, hyperlinks, or traceability tools embedded into requirements management tools [39, 40].

Table 1. Cost to repair software errors at various stages

Stage	Relative Repair Cost
Requirements	1–2
Design	5
Coding	10
Unit test	20
System test	50
Maintenance	200

When a change is proposed, the traceability infrastructure provides the ability to trace back to the rationale behind impacted requirements so that current decisions can be informed ones, and to trace forward to artifacts such as design documents, code, and test cases in order to more completely understand how to implement the change and to identify and mitigate its possible side effects. Unnecessary traces lead to a maintenance nightmare, whereas too little traceability provides inadequate support for the change process [41]. Therefore, links should be carefully established to provide necessary support for change analysis activities.

Change requests should be managed systematically. Many requirements management packages now also incorporate "request for change" (RFC) features. Once a RFC has been created, an impact analysis is performed and the change is prioritized and assessed for in terms of its benefits, cost, and effort. Any change that is approved should go through the same rigorous analysis and quality assurance process as the initial requirements.

Requirements attributes are an important part of the change management process. Each requirement is assigned a unique identifier for tracking purposes, and auxiliary attributes are used to record information such as change dates, rationales, and current status.

8. CONCLUSIONS

In this paper, we have emphasized a more traditional approach to requirements engineering in which the requirements process involves elicitation, analysis, specification, validation, and management. Recently, there has been a trend toward adopting more agile development methods [42]. Among other things, the agile philosophy has challenged the accepted wisdom that the cost of change increases over time and has adopted a more flexible approach that embraces the changing requirements of the customer throughout the development process. Agile methods minimize the importance of an up-front requirements phase, instead focusing upon delivering executable code to the customer as early as possible. Although agile methods are gaining in popularity, Boehm and Turner point out [43] that they are more suited to smaller, volatile, and noncritical projects. The more mainstream agile methods targeted at larger, more complex systems, or those developed in distributed environments, adopt many of the requirements practices described in this paper.

For readers interested in learning more about software requirements, there are numerous books available, several of which have been referenced in this paper [3, 4, 7, 11, 12, 15], that provide more detailed discussions on a variety of related topics.

REFERENCES

1. Jones, C., *Patterns of Software Systems Failure and Success,* International Thompson Computer Press, Boston, 1996.
2. The Standish Group, *Chaos Report,* 1995. Available online at: http://www.standishgroup.com/visitor/chaos.htm.
3. Young, R. R., *Effective Requirements Practices,* Addison-Wesley, 2001.
4. Kotonya, G. and I. Sommerville, *Requirements Engineering: Processes and Techniques,* Wiley, 2000.
5. *Guide to the Software Engineering Body of Knowledge* (SWEBOK). Available online at http://www.swebok.org
6. Thayer, R. H. and M. Dorfman, *Software Requirements Engineering* (2nd ed). IEEE Computer Society Press, 1997.
7. Sommerville, I. *Software Engineering* (6th ed.), Addison-Wesley, pp. 63–97, 97–147, 2001.
8. IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology.*
9. Thayer, R. H., "Software Requirements Engineering Concepts," in *Software Engineering, Part 1: The Development Process,* 2nd ed., Edition, R. H. Thayer, ed., IEEE Computer Society Press, Los Amitos, CA, 2002.
10. Sommerville, I. and P. Sawyer, "Viewpoints: Principles, Problems, and a Practical Approach to Requirements Engineering," *Annals of Software Engineering,* vol. 3, N. Mead, ed., 1997.
11. Sutcliffe, A. *User-Centred Requirements Engineering,* Springer-Verlag, 2002.
12. Robertson, S. and Roberston, J., *Mastering the Requirements Process,* Addison-Wesley, 1999.
13. Jackson, M., Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices, Addison-Wesley, 1995.
14. Alexander, I. and Robertson, S., "Understanding Project Sociology by Modeling Stakeholders", *IEEE Software, 21,* 1, Jan/Feb, 2004, pp. 23–27.
15. Alexander, I. and Stevens, R., *Writing Better Requirements,* Addison-Wesley, 2002.
16. In, H. and Boehm, B., "Using WinWin Quality Requirements Management Tools: A Case Study," *Annals of Software Eng.,* vol. 11, 2001, pp. 141–174.
17. M. Denne and J. Cleland-Huang, *Software by Numbers: Low-Risk, High-Return Development,* Prentice-Hall, 2003.
18. B. Boehm, "Value-based Software Engineering," *ACM SIGSOFT Software Engineering Notes, 28,* 2, Mar. 2003.

19. Finkelstein, A. and Dowell, J. "A Comedy of Errors: the London Ambulance Service Case Study," in *8th International Workshop on Software Specification & Design (IWSSD-8)*. IEEE Computer Society Press, 1996, pp. 2–4.
20. Cockburn, A., *Writing Effective Use Cases,* Addison-Wesley, 2000.
21. Potts, C., Takahashi, K., et al. "Inquiry-based Requirements Analysis," *IEEE Software, 11,* 2, 1994, pp. 21–32.
22. Hickey, A. M. and Davis, A. M., "Elicitation Technique Selection: How Do Experts Do It?," in *11th IEEE International Requirements Engineering Conference,* Sept. 8–12, 2003 pp. 169–180.
23. Maiden, N., and G. Ruggs, "ACRE: Selecting Methods for Requirements Acquisition," *Software Engineering J, 11,* 5 (May 1996), pp. 183–192.
24. IEEE Std 1320.1-1998, *IEEE Standard for Functional Modeling Language—Syntax and Semantics for IDEF0,* IEEE, New York, 1998.
25. IEEE Std 1320.2-1998, *IEEE Standard for Conceptual Modeling Language—Syntax and Semantics for IDEF1X$_{97}$*, IEEE, New York, 1998.
26. Nuseibeh, B., "Weaving Together Requirements and Architecture", *IEEE Computer, 34,* 3, Mar. 2001, pp. 115–117.
27. Dobrica, L. and Niemela, E., "A Survey on Software Architecture Analysis Methods," *IEEE Transactions on Software Engineering,* July 2002, pp. 638–653
28. Kazman, R., Klein, M., and Clements, P., "ATAM: Method for Architecture Evaluation," Software Engineering Institute Technical Report, CMU/SEI-2000-TR-004, August 2000.
29. Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J., *Non-Functional Requirements in Software Engineering,* The Kluwer International Series in Software Engineering, Volume 5, Kluwer Academic Press, 1999.
30. IEEE Std 1233, 1998 Edition, *IEEE Guide for Developing System Requirements Specifications*, IEEE, New York, 1998.
31. Boehm, B. W. "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software, 1,* 1, Jan. 1984, pp. 75–88.
32. Young, R. *Effective Requirements Practices,* Addison-Wesley, 2001.
33. IEEE Std 830-1998, *IEEE Recommended Practice for Software Requirements Documentation,* IEEE, New York, 1998.
34. Firesmith, D., "Modern Requirements Specification" *Journal of Object Technology, 2,* 2, 2003, pp. 53–64.
35. IEEE Std 1012, 1998 Edition, *IEEE Standard for Software Verification and Validation,* IEEE, New York, 1998.
36. Boehm, B., *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, NJ, 1981.
37. Pressman, R. *Software Engineering: A Practitioners Approach,* 4th ed. McGraw-Hill, 1996.
38. Freedman, D. P., and Weinberg, G. M., *Handbook of Walkthroughs, Inspections, and Technical Reviews,* Dorset House, New York, 1990.
39. Gotel, O. and Finkelstein, A., "An Analysis of the Requirements Traceability Problem," in *1st International Conference on Requirements Engineering,* 1994, pp. 94–101.
40. Ramesh, B., and Jarke, M., "Toward Reference Models for Requirements Traceability," *IEEE Transactions on Software Engineering, 27,* 1, Jan. 2001, pp. 58–92.
41. Cleland-Huang, J., Zemont, G., and Lukasik, W., "Heterogeneous Solutions for Improving the ROI of Requirements Traceability," in *IEEE International Requirements Engineering Conference,* Kyoto, Japan, Sept. 8–10, 2004, pp. 230–239.
42. Beck, K., *Extreme Programming: Embrace Change,* Addison-Wesley, 1999.
43. Boehm, B., and Turner, R., *Balancing Agility and Discipline: A Guide for the Perplexed,* Addison-Wesley.