# Chapter 4

R. Representation

# Writing the Requirements Document

## Requirements Agreement and Analysis

Throughout the elicitation activity and especially before finalizing the SRS (systems requirements specifications), raw requirements should be analyzed for problems and these problems reconciled. Requirements analysis is the activity of analyzing requirements for problems. Problematic requirements include those that are:

- Confusing
- Extraneous
- Duplicated
- Conflicting
- Missing

We make these concepts more rigorous in Chapter 5.

Requirements analysis is usually informal; for example, customers and stakeholder representatives are shown elicited requirements in a format that they can understand to ensure that the requirements engineer's interpretation of all customers' desiderata are correct. Formal methods provide more rigorous techniques for requirements analysis.

Requirements agreement is the process of reconciling differences in the same requirement derived from different sources. Requirements agreement is generally an informal process, although a systematic process should be used. Use cases and user stories are often effective tools for this purpose (Hull, Jackson, and Dick 2011). Screen mockups are another useful technique, especially when used in

conjunction with functional prototypes (Ricca et al. 2010). Others suggest using scenarios, storyboards, paper prototypes, scripted concept generators, and functional prototypes for requirements agreements (Sutcliffe, Thew, and Jarvis 2011).

As part of the customer agreement process, Hull, Jackson, and Dick (2011) suggest asking the following questions:

- Is the requirement complete?
- Is the requirement clear?
- Is the requirement implementable?
- Is the qualification plan clear and acceptable?

Note in Chapter 5 how these questions relate to the desirable qualities of a requirements specification, and in Chapters 5 and 6 we look at more precise ways to achieve these qualities throughout the requirements engineering life cycle.

In some cases, particularly when the application domain is complex or unfamiliar to the requirements engineer, it is helpful to develop a context diagram to describe the system boundaries with respect to its environment. The context diagram can assist in requirements agreement and analysis by showcasing ambiguous, complex, and missing requirements.

Context diagrams are informal and there are no standard techniques for representing them, however, use case, onion skin, or block diagrams are often used. For example, the simple block diagram shown in Figure B.1 in Appendix B provides the context for the wet well pumping system.

## Requirements Representation Approaches

Various techniques can be used to describe functionality in any system, including use cases, user stories, natural languages, formal languages, stylized (restricted natural) languages, and a variety of formal, semi-formal, and informal diagrams. Selecting the right technique for a single project or organizationwide use is a significant challenge, and in a sense, a theme of this book. Eberlein and Jiang (2011) provide a focused discussion of selecting various requirements techniques including specification approaches. When making such selections, they suggest considering the following factors:

- Technical issues such as the maturity and effectiveness of the technique being considered
- Complexity of the techniques
- Social and cultural issues such as organization opposition to change
- Level of education and knowledge of developers
- Certain characteristics of the software project such as time and cost constraints, complexity of the project, and the structure and competence of the software team



Generally, there are three approaches to requirements representation: formal, informal, and semi-formal. Requirements specifications can strictly adhere to one or another of these approaches, but usually they contain elements of at least two of these approaches (informal and one other). Formal representations have a rigorous mathematical basis, and we explore these further in Chapter 6. But even formal requirements specifications documents will have elements of informal or semi-formal specification.

Informal requirements representation techniques cannot be completely transliterated into a rigorous mathematical notation. Informal techniques include natural language (i.e., human languages), flowcharts, ad hoc diagrams, and most of the elements that you may be used to seeing in systems requirements specifications. In fact, all SRS documents will have some informal elements. We can state this fact with confidence because even the most formal requirements specification documents have to use natural language, even if it is just to introduce a formal element. Therefore, we focus our attention here mostly on requirements representation using informal techniques.

Finally, semi-formal representation techniques include those that, although appearing informal, have at least a partial formal basis. For example, many of the diagrams in the Unified Modeling Language (UML) or Systems Modeling Language (SysML) family of metamodeling languages including the Use Case Diagram are informal or can be formalized. UML and SysML are generally considered as semi-formal modeling techniques; however, they can be made entirely formal with the addition of appropriate formalisms (Laplante 2006).

User stories were discussed in Chapter 3, and we briefly study one semi-formal technique, the Use Case Diagram, in this chapter. Formal requirements modeling is discussed in Chapter 6. As it is the predominant approach used in industry, and the most problematic, the majority of this chapter focuses on informal modeling using natural language.

The 2003 Neill and Laplante and the 2008 Marinelli studies were intended to discover the prevalence of various specification techniques that were used (Neill and Laplante 2003). Many of the findings with respect to elicitation techniques have been reported already but not with respect to representation approaches. In both surveys the majority of users reported that requirements were expressed in terms of natural language (Figure 4.1).

Clearly informal representations still dominated industry practices then, and it is discouraging that there was little change in notation used between 2003 and 2008, particular that there was not an increase in utilization of semi-formal and formal methods. The 2008 survey also showed that there was no discernible relationship between specification notation chosen and software development life-cycle (SDLC) methodology utilized.

The 2003 study also yielded information about the number of requirements in an SRS document (Figure 4.2). Here it is interesting to see that a relatively large number of requirements specifications (about 20%) were reported to contain

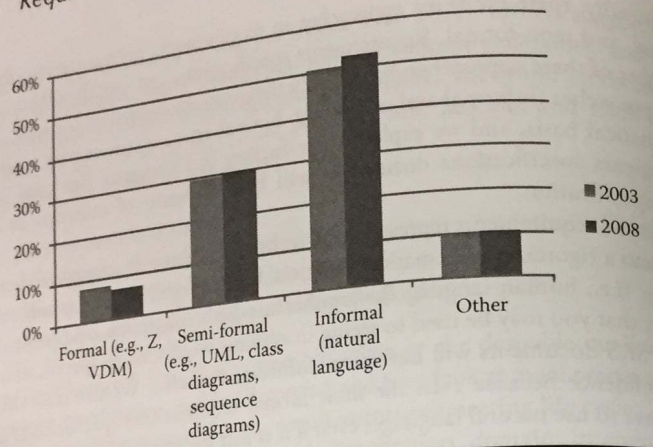


**Figure 4.1 Requirement specification notation prevalence in 2003 and 2008 industry surveys. (Adapted from C.J. Neill and P.A. Laplante, *Software*, 20(6): 40–45, 2003, and V. Marinelli, *An Analysis of Current Trends in Requirements Engineering Practice*, Master of Software Engineering, Professional Paper, Penn State University, Great Valley School of Graduate Professional Studies, Malvern, PA, 2008.)**

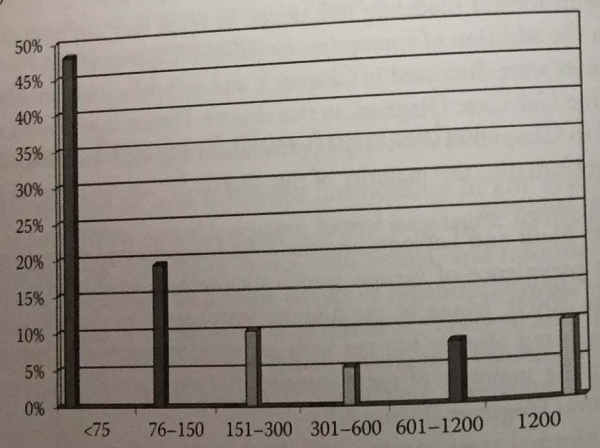


**Figure 4.2 Reported number of requirements specifications. (Reprinted from C.J. Neill and P.A. Laplante, *Software*, 20(6): 40–45, 2003. With permission.)**

between 76 and 150 individual requirements. Only a small number of systems (about 10%) were reported to be "large," having greater than 1,200 individual requirements (Neill and Laplante 2003).

Finally, in the 2003 survey we get some data on the number of pages in a requirements specification document (Figure 4.3). Interestingly we see a relatively



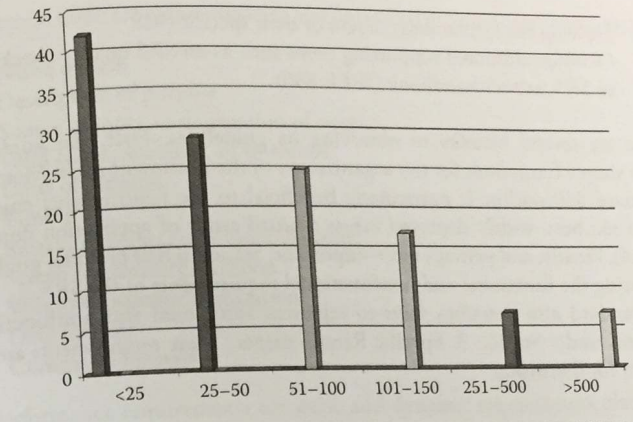


**Figure 4.3 Reported number of pages of requirements specifications. (Reprinted from C.J. Neill and P.A. Laplante, *Software*, 20(6): 40–45, 2003. With permission.)**

high proportion (around 30%) of small (2,550-page) SRS documents, likely corresponding to "small" development projects. These numbers are consistent with those found in a study of 56 NASA requirements specifications documents developed in the 1990s through the early 2000s. For the projects studied, the SRS document sizes ranged from about 140 to 28,000 lines of text, with a median of 2,200 and an average of 4,700 lines of text (Wilson, Rosenberg, and Hyatt 1997). Assuming 60 lines of text per page, these figures convert to a range of 2 to 470 pages, a median of 37 and an average of 78 pages. These numbers are quite consistent with those shown in Figure 4.3.

The 2008 survey did not ask about the number of requirements or requirements document size. Since 2003 no other comprehensive surveys of requirements size have been published.

## IEEE Standard 830–1998

The IEEE Standard 830–1998, Recommended Practice for Software Requirements Specifications, describes recommended approaches for the specification of software requirements. The standard is based on a model that produces a document that helps:

- Software customers to accurately describe what they wish to obtain
- Software suppliers to understand exactly what the customer wants
- Individuals to accomplish the following goals:
  - Develop a standard software requirements specification outline for their own organizations

- Develop the format and content of their specific SRS
- Develop additional supporting items such as an SRS quality checklist or an SRS writer's handbook (IEEE 830)

There are several benefits to observing its guidelines. First, the guidelines provide a simple framework for the organization of the document itself. Moreover, the Standard 830 outline is particularly beneficial to the requirements engineer because it has been widely deployed across a broad range of application domains (Figure 4.4). Finally, and perhaps more important, Standard 830 provides guidance for organizing the functional and nonfunctional requirements of the SRS.

The standard also describes ways to represent functional and nonfunctional requirements under Section 3, Specific Requirements. These requirements are the next subject for discussion.

## IEEE Standard 830 Recommendations on Representing Nonfunctional Requirements

Standard 830 defines several types of nonfunctional requirements involving:

- External interfaces
- Performance
- Logical database considerations
- Design constraints
- Software system attribute requirements



**Figure 4.4 Table of contents for an SRS document as recommended by IEEE Standard 830.**



External interface requirements can be organized in a number of ways, including:

- Name of item
- Description of purpose
- Source of input or destination of output
- Valid range, accuracy, or tolerance
- Units of measure
- Timing
- Relationships to other inputs/outputs
- Screen formats/organization
- Window formats/organization
- Data formats
- Command formats

Performance requirements are static and dynamic requirements placed on the software or on human interaction with the software as a whole. Typical performance requirements might include the number of simultaneous users to be supported, the numbers of transactions and tasks, and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

Logical database requirements are types of information used by various functions such as:

- Frequency of use
- Accessing capabilities
- Data entities and their relationships
- Integrity constraints
- Data retention requirements

Design constraint requirements are related to standards compliance and hardware limitations. We show how to organize constraints in the requirements statements shortly.

Finally, software system attributes can include reliability, availability, security, maintainability, portability, and just about any other "ility" you can imagine.

## IEEE Standard 830 Recommendations on Representing Functional Requirements

The functional requirements should capture all system inputs and the exact sequence of operations and responses (outputs) to normal and abnormal situations for every input possibility. Functional requirements may use case-by-case descriptions or other general forms of description (e.g., using universal quantification, use cases, user stories). However, be wary of the dangers of using universal quantification in requirements as described in Chapter 1. Similarly, related to "for all" requirements are those using the words "never," "always," "none," and "each." Because these

can be formally equated to universal quantification through negation, you are also advised to be wary of these kinds of requirements (Laplante 2009).

The IEEE Standard 830–1998 is not prescriptive in terms of how to organize specific functional requirements; instead, a menu of organizational options is offered. Specific functional requirements can be organized by:

- Functional mode (e.g., "navigation," "combat," "diagnostic")
- User class (e.g., "user," "supervisor," "diagnostic")
- Object (by defining classes/objects, attributes, functions/methods, and messages)
- Feature (describes what the system provides to the user)
- Stimulus (e.g., sensor 1, sensor 2, actuator 1, and so on)
- Functional hierarchy (e.g., using structured analysis)

Or a combination of these techniques can be used within one SRS document.

For example, the smart home description given in Appendix A is a feature-driven description of functionality. One clue that it is feature driven is the mantra "the system shall" for many of the requirements. The wet well pumping system specification in Appendix B is also largely described by functionality.

Alternatively, consider a system organized by functional mode, the NASA WIRE (Wide-Field Infrared Explorer) System (1996). This system was part of the "Small Explorer" program. The system describes a submillimeter-wave astronomy satellite incorporating standardized space-to-ground communications. Information was to be transmitted to the ground and commands received from the ground, according to the Consultative Committee for Space Data Systems (CCSDS) and Goddard Space Flight Center standards (NASA 1996).

The flight software requirements were organized as follows:

- System management
- Command management
- Telemetry management
- Payload management
- Health and safety management
- Software management
- Performance requirements

Reviewing the document under the System Management mode we see the operating system functionality described as follows:

### Operating System

001 The operating system shall provide a common set of mechanisms necessary to support real-time systems such as multitasking support, CPU scheduling, basic communication, and memory management.
  001.1 The operating system shall provide multitasking capabilities.



  001.2 The operating system shall provide event-driven, priority-based scheduling.
    001.2.1 Task execution shall be controlled by a task's priority and the availability of resources required for its execution.
  001.3 The operating system shall provide support for intertask communication and synchronization.
  001.4 The operating system shall provide real-time clock support.
  001.5 The operating system software shall provide task-level context switching for the 80387 math coprocessor.

Also under system functionality is Command Validation, defined as follows:

### Command Validation

211 The flight software shall perform CCSDS command structure validation.
  211.1 The flight software shall implement CCSDS Command Operations Procedure number 1 (COP-1) to validate that CCSDS transfer frames were received correctly and in order.
  211.2 The flight software shall support a fixed-size frame-acceptance and reporting mechanism (FARM) sliding window of 127 and a fixed FARM negative edge of 63.
  211.3 The flight software shall telemeter status and discard the real-time command packet if any of the following errors occur:
    - Checksum fails validation prior to being issued.
    - An invalid length is detected.
    - An invalid Application ID is detected.
  211.4 The flight software shall generate and maintain a Command Link Control Word (CLCW). Each time an update is made to the CLCW, a CLCW packet is formatted and routed for possible downlink (NASA 1996).

It is also very common in software-based systems to take an object-oriented approach to describe the system behavior. This is particularly the case when the software is expected to be built using a pure object-oriented language such as Java.

Object-oriented representations involve highly abstract system components called objects and their encapsulated attributes and behavior. The differences between traditional "structured" descriptions of systems and object-oriented descriptions of systems are summarized in Table 4.1.

When dealing with requirements organized in an object-oriented fashion, it is very typical to utilize user stories (especially in conjunction with agile software development methodologies, which we discuss in Chapter 7) or use cases and Use Case Diagrams to describe behavior.

**Table 4.1 Structured versus Object-Oriented Representation**

| | *Structured* | *Object-Oriented* |
|---|---|---|
| System components | Functions | Objects |
| Data and control specification | Separated through internal decomposition | Encapsulated within objects |
| Characteristics | Hierarchical structure | Inheritance<br>Relationship of objects |
| | Functional description of system | Behavioral description of system |
| | Encapsulation of knowledge within functions | Encapsulation of knowledge within objects |

## *ISO/IEC Standard 25030*

ISO/IEC Standard 25030 is designed to be complementary to IEEE Standards 830 and 1223.* ISO 25030 is intended to take a "process view" rather than a product view, with further emphasis on measurable quality requirements. The standard template for 25030, which is similar to that specified by 830, is shown in Figure 4.5.

In particular, Section 6.3, System Boundaries, is a suitable place for a context diagram, whereas Section 6.4 is intended to provide system quality requirements based on measurable targets. This practice is in keeping with IEEE 830's "measurable" characteristics of good requirements. Attribute metrics also assist in comparing values or computing relevant statistics that can lead to system improvement over time (Glinz 2008).

For example, consider a hypothetical requirement 3.4.2 in the baggage handling system:

> 3.4.2 Each baggage scanner unit shall process, on average, 10 pieces of luggage per minute.

A suggested format for the measurable targets based on one suggested in Glinz (2008) is given in Figure 4.6. Standard 25030 can be used as a template for organizing the SRS document. Alternatively, if using the IEEE 830 or another standard template, Figure 4.6 can also be used to structure metrics and acceptance criteria for requirements, whatever the overall format being used for the SRS.

* IEEE Std. Guide for Developing Systems Requirements Specifications, IEEE, 2002. This standard is a software agnostic or systems-oriented version of Standard 830.





**Figure 4.5 Table of contents for requirements specification from ISO/IEC 25030.**

- *Attribute:* Average time that a scanner unit needs to scan a piece of luggage.
- *Scale:* Seconds (type: ratio scale).
- *Procedure:* Measure time required to scan a package for forbidden contents; take the average over 1,000 pieces of various types.
- *Planned value:* 50% less than reference value.
- *Lowest acceptable value:* 30% less than reference value.
- *Reference value:* Average time needed by competing or similar product to scan a piece of luggage.

**Figure 4.6 Sample requirements attribute metrics for baggage handling system.**

Standard 25030 is one of five collections of standards known as the "software product quality requirements and evaluation series of standards" (with the acronym "SQUARE"). The other four standards refer to the quality management (ISO/IEC 25000), quality model (ISO/IEC 25010), quality measurement (ISO/IEC 25020), and quality evaluation (ISO/IEC 25040; Boegh 2008).

## Use Cases*

Use cases have already been noted as a way for more sophisticated customers and stakeholders to describe their desiderata. Use cases are an essential element of many SRS documents and are described graphically using any of several techniques. Use cases depict the interactions between the system and the environment around the system, in particular, human users and other systems. They can be used to model the behavior of pure software or hybrid hardware/software systems.

Use cases describe scenarios of operation of the system from the designer's (as opposed to customers') perspective. Use cases are typically represented using a Use Case Diagram, which depicts the interactions of the system with its external environment. In a Use Case Diagram, the box represents the system itself. The stick figures represent "actors" that designate external entities that interact with the system. The actors can be humans, other systems, or device inputs. Internal ellipses represent each activity of use for each of the actors (use cases). The solid lines associate actors with each use. Figure 4.7 shows a Use Case Diagram for the baggage inspection system.

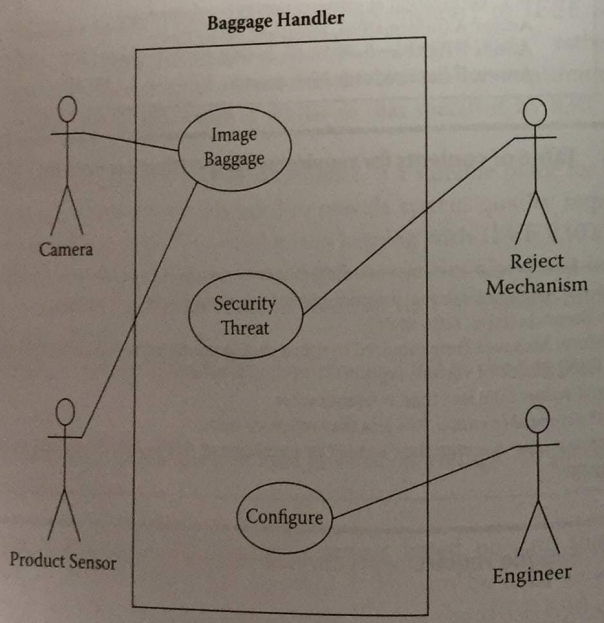


**Figure 4.7 Use Case Diagram of baggage inspection system.**

* This discussion is adapted from one found in Laplante (2006), with permission.



Three uses are shown: capturing an image of the baggage ("image baggage"), the detection of a security threat (in which case the bag is rejected from the conveyor for offline processing), and then configuration by the systems engineer. Notice that the imaging camera, product sensor, and reject mechanism are represented by a humanlike stick figure. This is typical: the stick figure represents a system "actor" whether human or not.

Appendix B, Figure B.4, provides another example: a Use Case Diagram for the wet well.

Each use case is a form of documentation that describes scenarios of operation of the system under consideration as well as pre- and postconditions and exceptions. In an iterative development life cycle these use cases will become increasingly refined and detailed as the analysis and design workflows progress. Interaction diagrams are then created to describe the behaviors defined by each use case. In the first iteration these diagrams depict the system as a "black box," but once domain modeling has been completed the black box is transformed into a collaboration of objects as shown later.

If well developed, sometimes the use cases can be used to form a pattern language, and these patterns and the derived design elements can be reused in related systems (Issa and Al-Ali 2010). Using patterns when specifying requirements ensures a greater level of consistency and can reduce errors in automated measurement of certain requirements properties.

## Requirements Document

The system (or software) requirements specification document is the official statement of what is required of the system developers. Hay (2003) likens the SRS to the score for a great opera: without a score there is chaos, but the finished score ensures that the activities and contributions of the various performers in the opera are coordinated.

It is also important to remember that, under those circumstances where there is a customer–vendor relationship between the sponsor and builders of the system, the SRS is a contract and is therefore enforceable under civil contract law (or criminal law if certain types of fraud or negligence can be demonstrated).

### *Users of a Requirements Document*

There are several "users" of the requirements document, each with a unique perspective, needs, and concerns. Typical users of the document include:

- Customers
- Managers
- Developers

- Test engineers
- Maintenance engineers
- Stakeholders

Customers specify the requirements and are supposed to review them to ensure that they meet their needs. Customers also specify changes to the requirements. Because customers are involved, and they are likely not engineers, SRS documents should be accessible to the lay person (formal methodologies excepted).

Managers at all levels will use the requirements document to plan a bid for the system and to plan for managing the system development process. Managers, therefore, are looking for strong indicators of cost and time-to-complete in the SRS.

And of course, developers use the requirements specification document to understand what system is to be developed. At the same time, test engineers use the requirements to develop validation tests for the system. Later, maintenance engineers will use the requirements to help understand the system and the relationship between its parts so that the system can be upgraded or fixed. Other stakeholders who will use the SRS include all of the direct and indirect beneficiaries (or adversaries) of the system in question, as well as lawyers, judges, plaintiffs, juries, district attorneys, arbiters, mediators, and so on, who will view the SRS as a legal document in the event of disputes.

## Requirements Document Requirements

What is the correct format for an SRS? There are many documentation formats, and none is better than any other (except, of course, in the case of a hard-to-read, badly organized, and imprecise document).

The IEEE 830 standard provides a general format for a requirements specification document. Most requirements management software will produce documents in customizable formats. But the "right" format all depends on what the sponsor, situation, customer, application domain, your employer, and so forth demand.

That the SRS document should be easy to change is evident for the many reasons we have discussed thus far. Furthermore, because the SRS document serves as a reference tool for maintenance, it should record forethought about the life cycle of the system, that is, to predict changes.

In terms of general organization, writing approach, and discourse, best practices include:

- Using consistent modeling approaches and techniques throughout the specification, for example, a top-down decomposition, structured, or object-oriented approaches
- Separating operational specification from descriptive behavior
- Using consistent levels of abstraction within models and conformance between levels of refinement across models



- Modeling nonfunctional requirements as a part of the specification models, in particular timing properties
- Omitting hardware and software assignments in the specification (another aspect of design rather than specification)

Following these rules will always lead to a better SRS document. Finally, the IEEE Standard 830 describes certain desirable qualities for requirements specification documents and these are discussed in Chapter 5.

### Preferred Writing Style

Engineers (of all types) have acquired an unfair reputation for poor communication skills, particularly writing. In any case, it should be clear now that requirements documents should be very well written. It is out of scope to offer recommendations on writing here, but there are excellent texts on technical writing that should be studied, such as Laplante (2011). You are urged to improve your writing through practice, study (of writing techniques), and through reading; yes, read "well-written" SRS documents to learn from them. You should also read literature, poetry, and news, as much can be learned about economy and clarity of presentation from these writings. Some have even suggested screenwriting as an appropriate paradigm for writing requirements documents (or for user stories and use cases; Norden 2007).

In any case approach the requirements document like any writing: be prepared to write and rewrite, again and again. Have the requirements document reviewed by several other stakeholders (and possibly a nonstakeholder who writes well).

### Text Structure Analysis

Metrics can be helpful in policing basic writing features. Most word-processing tools calculate average word, sentence, and paragraph length, and these are valuable to collect because they can highlight writing that is too simplistic or too difficult to understand. One important feature of writing that is not computed by standard word processors, however, is the numbering structure depth.

Numbering structure depth is a metric that counts the numbered statements at each level of the source document. For example, first-level requirements, numbered 1.0, 2.0, 3.0, and so forth, are expected to be very high-level (abstract) requirements. Second-level requirements numbered 1.1, 1.2, 1.3, …, 2.1, 2.2, 2.3, …, 3.1, 3.2, and so on are subordinate requirements at a lower level of detail. Even more detailed requirements will be found at the third level, numbered as 1.1.1, 1.1.2, and so on. A specification can continue to fourth or even fifth-level requirements, but normally, third or fourth levels of detail should be sufficient. In any case, the counts of requirements at level 1, 2, 3, and so on provide an indication of the document's organization, consistency, and level of detail.

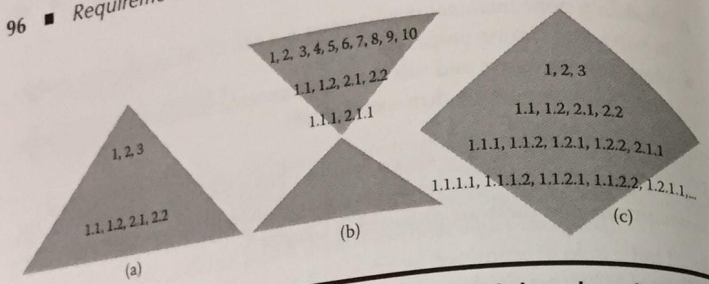


**Figure 4.8 (a) Pyramid, (b) hourglass, and (c) diamond-shaped configurations for requirements text structure.**

A thoughtful and well-organized SRS document should have a consistent level of detail, and if you were to list the requirements at each level, the resultant shape should look like a pyramid in that there should be a few numbered statements at level 1 and each lower level should have increasingly more numbered statements than the level above it (Figure 4.8a).

On the other hand, requirements documents whose requirements counts at each level resemble an hour-glass shape (Figure 4.8b) are usually those that contain a large amount of introductory and administrative information. Finally, diamond-shaped documents, represented by a pyramid followed by decreasing statement counts at lower levels (Figure 4.8c) indicate an inconsistent level of detail representation (Hammer et al. 1998).

The NASA ARM tool introduced in Chapter 5 computes text structure for a given SRS document in an appropriate format.

### *Requirement Format*

Each requirement should be in a form that is clear, concise, and consistent in the use of language. Using consistent language assists users of the requirements documents (Hull, Jackson, and Dick 2011) and renders analysis of the SRS document by software tools much easier.

A simplified standard requirement form is:

> The [noun phrase] shall (not) [verb phrase]

where [noun phrase] describes the main subject of the requirement, the shall (or shall not) statement indicates a required or prohibited behavior and [verb phrase] indicates the actions of the requirement.

Consider these two requirements for the baggage handling system:

- The system shall reject untagged baggage.
- The system shall not reject damaged baggage.



We assume that the terms "baggage," "reject," "tagged," and "damage" have been defined somewhere in the SRS document.

It is not uncommon for requirements to be written in nonstandard form. For example, the first requirement shown could be rewritten as:

> Untagged baggage shall be rejected by the system.

Both versions of this requirement are equivalent, but representing requirements in the standard form is desirable for the reasons already noted.

It is also not uncommon to omit the article in requirements, so the two example requirements would be:

- System shall reject untagged baggage.
- System shall not reject damaged baggage.

There is nothing wrong with this formulation of requirements.

It is important to identify requirements in some way for ease of reference, usually with hierarchical numbering. Hence the standard form for a requirement extends to the following:

> [identifier] The [noun phrase] shall (not) [verb phrase]

Returning to the previous two requirements, and inventing some identifier numbers, we have the following two examples:

> 1.2.1 The system shall reject untagged baggage.
> 1.2.2 The system shall not reject damaged baggage.

Finally, it is desirable to place measurable constraints on performance for functional requirements whenever possible, enriching the standard requirement form to:

> [identifier] The [noun phrase] shall (not) [verb phrase] [constraint phrase]

For the baggage handling system requirement number 1.2.1, a realistic constraint might be incorporated to yield the following requirement:

> 1.2.1 The system shall reject untagged baggage within 5 seconds of identification.

This requirement can be augmented with other constraint attributes in the format shown in Figure 4.6. There are numerous variations of the "final" requirements standard form depicted, for example, those given by Hull et al. (2011). These variations, however, can be shown to be reducible to the format shown above.

**Table 4.2 Imperative Words and Phrases**

| *Imperative* | *Most Common Use* |
|---|---|
| are applicable | To include, by reference, standards, or other documentation as an addition to the requirements being specified |
| is required to | As an imperative in specifications statements written in the passive voice |
| must | To establish performance requirements or constraints |
| responsible for | In requirements documents that are written for systems whose architectures are predefined |
| shall | To dictate the provision of a functional capability |
| should | Not frequently used as an imperative in requirement specification statements |
| will | To cite things that the operational or development environment are to provide to the capability being specified |

*Source:* Wilson et al. 1997.

### *Use of Imperatives*

"Shall" is a command word or imperative that is frequently used in requirements specifications. Other command words and phrases that are frequently used include "should," "must," "will," and "responsible for," and there are more. Wilson, Rosenberg, and Hyatt (1996) describe the subtle differences in these imperatives, and these are summarized in Table 4.2.

It is considered good practice to use "shall" as the imperative when requirements are mandatory and to omit the use of weak imperatives such as "should" for optional requirements. When requirements are ranked for importance, as they should be, a requirement in standard form with a low rank essentially can be deemed an optional requirement.

## Behavioral Specifications

In some cases the requirements engineer may be asked to reverse engineer requirements for an existing system when the requirements do not exist, are incomplete, out of date, or incorrect. It may also be necessary to generate requirements for open source software (software that is free for use or redistribution under the terms of a license) for the purposes of generating test specifications. In these cases a form of SRS, the behavioral specification, needs to be generated.

The behavioral specification is identical in all aspects to the requirements specification, except that the former represents the engineers' best understanding of what the users intended the system to do, and the latter represents the users' best understanding of what the system should do. The behavioral specification has an additional layer of inference and therefore can be expected to be even less complete than a requirements specification.



Fortunately, there is an approach to generating the behavioral specification. The technique involves assembling a collection of as many artifacts as possible that could explain the system's intent. Then, these artifacts are used to reconstruct the system's intended behavior as it is best understood. The foregoing description is adapted from the original paper by Elcock and Laplante (2006).

Artifacts that may be used to derive a behavioral specification include (but should not be limited to):

- Any existing requirements specification, even if it is out of date, incomplete, or known to be incorrect
- User manuals and help information (available when running the program)
- Release notes
- Bug reports and support requests
- Application forums
- Relevant versions of the software under consideration

Some of these artifacts may have to be scavenged from various sources, such as customer files, e-mails, open source community repositories, archives, and so forth. A brief description of how these artifacts are used in generating the specification is given below.

Starting with user manuals, statements about the behavior of an application in response to user input can be directly related to the behavioral specification. User manuals are particularly well suited for this purpose inasmuch as describing the response of software to user stimulus is germane to their existence. Help information, such as support websites and application help menus, are also rich in information that can either be used directly or abstracted to define behavioral requirements.

Next, release notes can be consulted. Release notes are typically of limited benefit to the process of developing test cases because they tend to focus on describing which features are implemented in a given release. There is usually no description of how those supported features should function. However, release notes are particularly important in resolving the conflict that arises when an application does not respond as expected for features that were partially implemented in or removed from future implementation.

Defect reports can also be a great source for extracting behavioral responses because they are typically written by users, they identify troublesome areas of the software, and they often clarify a developer's intention for the particular behavior. Defect reports, at least for open source systems, are readily found in open repositories such as Bugzilla. Although it is true that in some cases bug

reports contain too much implementation detail to be useful for extracting behavioral responses, they can be discarded if behavior cannot be extrapolated from the details.

In many ways, the content of support requests is similar to bug reports in that they identify unexpected behavior. Unlike bug reports, however, support requests can sometimes be helpful in identifying features that are not fully implemented as well as providing information that illuminates the expectations of users. It is important to investigate both because, in addition to providing the insights mentioned, they may also aid in rationalizing unexpected behavior.

For many open source projects, and some closed source projects, there are web-based forums associated with the project. Within these forums, various amounts of behavioral information can be extracted. Ranging from useless to relevant, open-discussion postings need to be carefully filtered and applied only when other development artifacts are lacking. As with other artifacts, these postings can also be used to clarify behavior.

Finally, in the absence of any other artifacts, the software being tested could itself be an input to developing structural (glass-box) test cases. Assuming that this approach is necessary, the reality is that it will really be the tester's characterization of correct behavior that will largely prevail in defining the test cases.

Once the discovery process has concluded, the behavioral specification can be written. The format of the behavioral specification is identical to that for requirements specification, and all of the IEEE 830 rules should be applied (Elcock and Laplante 2006).

## Best Practices and Recommendations

Writing effective requirements specifications can be very difficult even for trivial systems because of the many challenges that we have noted. Some of the more common dangers in writing poor SRS documents include:

- Mixing of operational and descriptive specifications
- Combining low-level hardware functionality and high-level systems and software functionality in the same functional level
- Omission of timing information

Other bad practices arise from failing to use language that can be verified. For example, consider this set of "requirements":

- The system shall be completely reliable.
- The system shall be modular.
- The system will be fast.
- Errors shall be less than 99%.



What is wrong with these? The problem is they are completely vague and immeasurable and therefore their satisfaction cannot be demonstrated. For example, what does "completely reliable" mean? Any arbitrary person will have a different meaning for reliability for a given system. Modularity (in software) has a specific meaning, but how is it measurable? What does "fast" mean? Fast as a train? Faster than a speeding bullet? This requirement is just so vague. And finally, "Errors shall be less than 99%," is a recipe for a lawsuit: 99% of what? over what period of time? and so forth.

For the above set of requirements, improved versions are:

- Response times for all level-one actions will be less than 100 ms.
- The cyclomatic complexity of each module shall be in the range of 10 to 40.
- 95% of the transactions shall be processed in less than 1 s.
- Mean time before first failure shall be 100 hours of continuous operation.

But even this set is imperfect, because there may be some important details missing; we really can't know what, if anything, is missing outside the context of the rest of the SRS document. Using Glintz's attribute template, shown in Figure 4.6, is a good solution to this problem.

Some final recommendations for the writing of specification documents are:

- Invent and use a standard format and use it for all requirements.
- Use language in a consistent way.
- Use "shall" for mandatory requirements.
- Use "should" for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of technical language unless it is warranted.

Dick (2010) also suggests avoiding the following when writing the requirements document:

- Rambling phrases
- Hedging clauses, such as "if necessary"
- Conjunctions such as "and," "or," and "but," because these can lead to multiple requirements in a single statement
- Speculative terms such as "perhaps" and "sometimes"
- Vague terms such as "user friendly" and "generally" because these cannot be verified

But we are not done yet with our treatment of writing the requirements specification. Chapter 5 is devoted to perfecting the writing of specific requirements.

## Exercises

4.1 Under what circumstances is it appropriate to represent an SRS using informal techniques only?

4.2 What can the behavioral specification provide that a requirements document cannot?

4.3 If the customer requests that future growth and enhancement ideas be kept, where can these ideas be placed?

4.4 What are some items to be included under "data retention" in the SRS?

4.5 Here are some more examples of vague and ambiguous requirements that have actually appeared in real requirements specifications. Discuss why they are vague, incomplete, or ambiguous. Provide improved versions of these requirements (make necessary assumptions).

4.5.1 The tool will allow for expedited data entry of important data fields needed for subsequent reports.

4.5.2 The system will provide an effective means for identifying and eliminating undesirable failure modes and/or performance degradation below acceptable limits.

4.5.3 The database creates an automated incident report that immediately alerts the necessary individuals.

4.5.4 The engineer shall manage the system activity, including the database.

4.5.5 The report will consist of data, in sufficient quantity and detail, to meet the requirements.

4.5.6 The data provided will allow for a sound determination of the events and conditions that precipitated the failure.

4.5.7 The documented analysis report will include, as appropriate, investigation findings, engineering analysis, and laboratory analysis.

4.6 In Section 9.4 of the SRS in Appendix A, which requirements are suitable for representation using the "measurable targets" in the format shown in Figure 4.6?

4.7 Many of the requirements in Appendices A and B can be improved in various ways. Select 10 requirements listed and rewrite them in an improved form. Discuss why your rewritten requirements are superior using the vocabulary of IEEE 830.

4.8 Draw a context diagram for the baggage handling system. Make whatever assumptions you need.

4.9 Draw a context diagram for the pet store point of sale system. Make whatever assumptions you would need.

*4.10 Research different techniques for creating context diagrams and prepare a report highlighting the strengths and weaknesses of each.

* This exercise is suitable for a small research assignment.



*4.11 The requirements specifications in Appendices A and B use imperatives inconsistently. Using a textual analysis tool such as ARM or via manual analysis, assess the use of imperatives in these documents and suggest improvements.

## References


Boegh, J. (2008). A new standard for quality requirements, *Computer*, 25(2): 57–63.

Dick, J. (2010). Requirements engineering: Principles and practice. In *Encyclopedia of Software Engineering*, P. Laplante (Ed.). Boca Raton, FL: Taylor & Francis. Published online, 949–961.

Eberlein, A. and Jiang, L. (2011). Requirements engineering: Technique selection. In *Encyclopedia of Software Engineering*. P. Laplante (Ed.), Boca Raton, FL: Taylor & Francis. Published online, 962–978.

Elcock, A. and Laplante, P.A. (2006). Testing without requirements, *Innovations in Systems and Software Engineering: A NASA Journal*, 2: 137–145.

Glinz, M. (2008). A risk-based, value-oriented approach to quality requirements, *Computer*, 25(2): 34–41.

Hammer, T.F., Huffman, L.L., Rosenberg, L.H., Wilson, W., and Hyatt, L. (1998). Doing requirements right the first time, *CROSSTALK The Journal of Defense Software Engineering*, 20–25.

Hay, D. (2003). *Requirements Analysis: From Business Views to Architecture*. Upper Saddle River, NJ: Prentice Hall PTR.

Hull, E., Jackson, K., and Dick, J. (2011). Requirements engineering in the problem domain. In *Requirements Engineering*, London: Springer-Verlag, 93–14.

IEEE Standard 830-1998 (1998). *Recommend Practice for Software Requirements Specifications*, IEEE Standards Press, Piscataway, NJ.

Issa, A.A. and Al-Ali, A. (2010). Use case patterns driven requirements engineering. In *Proceedings of the Second International Conference on Computer Research and Development*, 307–313.

Laplante, P.A. (2011). *Technical Writing: A Practical Guide for Engineers and Scientists*. Boca Raton, FL: CRC Press.

Laplante, P.A. (2006). *What Every Engineer Needs to Know About Software Engineering*. Boca Raton, FL: CRC/Taylor & Francis.

Marinelli, V. (2008). *An Analysis of Current Trends in Requirements Engineering Practice*, Master of Software Engineering Professional Paper, Penn State University, Great Valley School of Graduate Professional Studies, Malvern, PA.

NASA WIRE (Wide-Field Infrared Explorer) Software Requirements Specification (1996). http://science.nasa.gov/missions/wire/, last accessed January 23, 2013.

Neill, C.J. and Laplante, P.A. (2003). Requirements engineering: The state of the practice, *Software*, 20(6): 40–45.

Norden, B. (2007). Screenwriting for requirements engineers, *Software*, 26–27.


* This exercise is suitable for a small research assignment.