Karl Wiegers (1999)
Software Requirements
Microsoft Press
ISBN 0-7356-0631-5

Found March 3, 2001 at:
http://mspress.microsoft.com/prod/books/sampchap/3215/

Chapter 9: Documenting the Requirements

Requirements development has a final product: a documented agreement between the customers and the development group about the product to be built. This agreement spans the triad of business requirements, user requirements, and software functional requirements. As we saw earlier, the vision and scope document contains the business requirements, while the user requirements are captured in use-case documents. You must also document both the functional requirements derived from use cases and the product's nonfunctional requirements, including quality attributes and external interface requirements. Unless you write these requirements in an organized and readable fashion, and have key project stakeholders review and approve them, people will not be sure what they are agreeing to.

You can document software requirements in three ways:

- Textual documents that use well-structured and carefully written natural language
- Graphical models that illustrate transformational processes, system states and changes between them, data relationships, logic flows, or object classes and their relationships
- Formal specifications that define requirements by using mathematically precise formal logic languages

While formal specifications provide the greatest rigor and precision, few software developers, and virtually no customers, are familiar with formal notation. Despite its many shortcomings, structured natural language remains the most practical way of documenting requirements for most software projects. A text-based software requirements specification that contains the functional and nonfunctional requirements will meet the needs of most projects. Graphical analysis models augment the SRS by providing alternative views of the requirements.

This chapter addresses the purpose and structure of the SRS, including a suggested document template. Guidelines for writing functional requirements are also presented, along with several examples of imperfect requirement statements and suggestions for improving them. Graphical modeling techniques for representing the requirements are the topic of Chapter 10. Formal specification methods are not addressed further in this book; see Alan Davis's *Software Requirements: Objects, Functions, and States* (1993) for references to further discussions of formal specification methods.

The Software Requirements Specification

The software requirements specification is also known as the functional specification, requirements agreement, and system specification. The SRS precisely states the functions and capabilities that a software system must provide and the constraints that it must respect. The SRS is the basis for all subsequent project planning, design, and coding, as well as the foundation for system testing and user documentation. The SRS should describe as completely

as possible the intended external, user-visible behaviors of the system. It should not contain design, construction, testing, or project management details other than known design and implementation constraints. Several audiences use the SRS for various purposes:

- Customers and the marketing department rely on the SRS to know what product they can expect to be delivered.
- Project managers base their plans and estimates of schedule, effort, and resources on the product description contained in the SRS.
- The software development team relies on the SRS to understand what it is to build.
- The testing group uses the product behavior descriptions contained in the SRS to derive test plans, cases, and procedures.
- Software maintenance and support staff refer to the SRS to understand what each part of the product is supposed to do.
- The publications group writes user documentation, such as manuals and help screens, based on both the SRS and the user interface design.
- Training personnel can use both the SRS and the user documentation to help them develop educational materials.

As the ultimate repository for the product requirements, the SRS must be comprehensive: *all* requirements should be included. Developers and customers should make no assumptions. If any desired functional or nonfunctional requirement is not identified in the SRS, it isn't part of the agreement and no one should expect it to appear in the product.

Nothing says you have to write the product's entire SRS before you begin design and construction. You can also approach requirements specification iteratively or incrementally, depending on a number of factors: whether you can identify all of the requirements at the outset, whether the same people who write the SRS will build the product, the number of planned releases, and so forth. However, every project must have a baselined agreement for each set of requirements that will be implemented. *Baselining* is the process of transitioning an SRS under development into one that has been reviewed and approved. Changes in a baselined SRS should be made through the project's defined change control process.

All participants must work from the same set of approved requirements to avoid unnecessary rework and miscommunication. I know of one project that suddenly experienced a flood of bug reports from the testers. It turned out they had been testing against an older version of the SRS, and what they thought were bugs really were features. Much of their testing effort was wasted, because the testers were looking for the wrong system behaviors.

Chapter 1 presented several characteristics of high-quality requirements documents: they must be complete, consistent, modifiable, and traceable. Structure and write the SRS so that users and other readers can understand it (Sommerville and Sawyer 1997). Keep the following readability suggestions in mind:

- Number sections, subsections, and individual requirements consistently.
- Leave text ragged on the right margin, rather than being fully justified.
- Make liberal use of white space.
- Use visual emphasis (such as **bold**, underline, *italics*, and `different fonts`) consistently and judiciously.
- Create a table of contents and an index to help readers find the information they need.
- Number and label all figures and tables and refer to them by number.
- Use your word processor's cross-reference facility, rather than hard-coded page or section numbers, to refer to other items or locations within the document.

Labeling Requirements

To satisfy the SRS quality criteria of traceability and modifiability, every software requirement must be uniquely identified. This allows you to refer to specific requirements in a change request, modification history, cross-reference, or requirements traceability matrix. Because simple and bulleted lists are not adequate for this purpose, I will describe several different requirements-labeling methods with their advantages and shortcomings. Select whichever technique makes the most sense for your situation.

Sequence number. The simplest approach is to give every requirement a unique sequence number, such as UR-2 or SRS13. Commercial requirements management tools assign such a number when a new requirement is added to the tool's database (most such tools also support hierarchical numbering). The prefix indicates the requirement type, such as UR for "user requirement." The numbers are not reused, so a deleted requirement is simply flagged as deleted in the database, and a new requirement gets the next available number. This simple numbering approach does not provide any logical or hierarchical grouping of related requirements, and the labels give you no clue as to what each requirement is about.

Hierarchical numbering. This is perhaps the most commonly used convention. If the functional requirements appear in section 3.2 of your SRS, they will all have labels such as 3.2.4.3. More digits in the label indicate a more detailed, lower-level requirement. This method is simple and compact, and your word processor can probably assign the numbers automatically. However, the labels can expand to many digits in even a medium-sized SRS, and they tell you nothing about the purpose of each requirement. If you have to insert a new requirement, the numbers of all following requirements in that section will be incremented. Delete or move a requirement, and the numbers below it in that section will be decremented. These changes can create havoc for any references to the requirements elsewhere in the system.

An improvement on the simple hierarchical numbering approach is to number the major sections of the requirements hierarchically and then identify individual functional requirements in each section with a short text code followed by a sequence number. For example, the SRS might contain "Section 3.2.5 – Editor Functions," and the requirements in that section could be labeled ED-1, ED-2, and so forth. This approach provides some hierarchy and organization while keeping the labels short, somewhat meaningful, and positionally independent. Inserting new requirement ED-9 between ED-1 and ED-2 does not force you to renumber the rest of the section.

Hierarchical textual tags. Consultant Tom Gilb suggests a text-based hierarchical tagging scheme for labeling individual requirements (Gilb 1988). Consider this requirement: "The system shall ask the user to confirm any request for printing more than ten copies." This requirement might be tagged PRINT.COPIES.CONFIRM, which indicates that it is part of the print function and is related to the issue of setting the number of copies to be printed. Hierarchical textual tags are structured, semantically meaningful, and unaffected by adding, deleting, or moving other requirements. Their primary drawback is that they are bulkier than the hierarchical numeric labels.

Dealing with Incompleteness

Sometimes you know that you lack a piece of information about a specific requirement. You might need to consult with a customer, check an interface description for another system, or define another requirement before you can resolve this uncertainty. Use the notation *TBD* ("to be determined") as a standard indicator to highlight these knowledge gaps in the SRS. This way, you can search the SRS for TBDs to identify the spots where you know clarification is needed. Document who will resolve each issue, how, and by when. Number each TBD and create a list of all TBDs to help you track each item to closure.

Resolve all TBDs before you proceed with construction of a set of requirements, because any uncertainties that remain increase the risk of making errors and having to rework the requirements. When the developer encounters a TBD or other ambiguity, he might not go back to the requirement's originator to clarify or resolve it. More likely, the developer will make his best guess, which won't always be correct. If you must proceed with construction while TBDs remain, defer those requirements' implementation if you can, or design those portions of the product to be easily modifiable when the open issues are resolved.

User Interfaces and the SRS

Incorporating user interface designs in the SRS has both drawbacks and benefits. On the minus side, screen images and user interface architectures are descriptions of solutions (designs), not of requirements. If you cannot baseline the SRS until the user interface design is completed, the requirements development process will take longer than it would otherwise. This can try the patience of managers, customers, or developers who are already concerned about the time being spent on requirements development. User interface layouts are not a substitute for defining the functional requirements. Don't expect developers to deduce the underlying functionality and data relationships from screen shots. Including user interface designs in the SRS also implies that developers must follow the requirements change process every time they want to alter a user interface element.

On the plus side, exploring potential user interfaces can help you refine the requirements and makes the user–system interactions more tangible to both the users and the developers. User interface displays can also assist with project planning and estimation. You can count graphical user interface (GUI) elements or calculate the number of function points[1] associated with each screen and then estimate the effort it will take to implement the screens, based on your previous experience with similar development activities.

A sensible balance is to include conceptual images—sketches—of selected user interface components in the SRS, without creating an expectation that the implementations must precisely follow those models. This enhances communication by representing the requirements in another fashion, but without overconstraining the developers and overloading your change management process. For example, a preliminary sketch of a complex dialog box will illustrate the intent behind a portion of the requirements, but a skilled designer might turn it into a tabbed dialog or employ another approach that enhances usability.

A Software Requirements Specification Template

Every software development organization should adopt a standard SRS template for its projects. Several recommended SRS templates are available (Davis 1993; Robertson and Robertson 1999). Dorfman and Thayer (1990) collected some 20 requirements standards and several examples from the National Bureau of Standards, the U.S. Department of Defense, NASA, and several British and Canadian sources. Many people use templates derived from IEEE Standard 830-1998, "IEEE Recommended Practice for Software Requirements Specifications" (IEEE 1998). This is a well-structured, flexible template that is suitable for many kinds of software projects.

Figure 9-1 illustrates an SRS template that was adapted and extended from the IEEE 830 standard. Modify this template to fit the needs and nature of your projects. If a particular section of the template doesn't apply to your project, leave the section heading in place but specify that it does not apply. This will prevent the reader from wondering whether something important was omitted inadvertently. Use this template to guide your thinking, and add any sections that are pertinent to your project. As with any software project document, include a table of contents and a revision history that lists the changes that were made to the SRS, including the date of the change, who made the change, and the reason.

1. Introduction
1.1 Purpose
1.2 Document Conventions
1.3 Intended Audience and Reading Suggestions
1.4 Product Scope
1.5 References

2. Overall description
2.1 Product Perspective
2.2 Product Functions
2.3 User Classes and Characteristics
2.4 Operating Environment
2.5 Design and Implementation Constraints
2.6 Assumptions and dependencies

3. External Interface Requirements
3.1 User interfaces
3.2 Hardware interfaces
3.3 Software interfaces
3.4 Communications interfaces

4. System Features
4.x System Feature X
4.x.1 Description and Priority
4.x.2 Stimulus/Response Sequences & Information flows
4.x.3 Functional Requirements

5. Other Nonfunctional Requirements
5.1 Performance requirements
5.2 Safety requirements
5.3 Security Requirements
5.4 Software Quality Attributes
5.5 Business Rules
5.6 User Documentation

6. Other Requirements

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: To-Be-Determined List

Figure 9-1 *Template for software requirements specification.*

The rest of this section describes the information you would include in each section of the template in Figure 9-1. You can incorporate sections by reference to other existing project

documents (such as a vision and scope document or an interface specification), rather than duplicating information in the SRS or bundling everything together into a single document. Don't follow the template dogmatically; do what makes sense for you.

1. Introduction

The introduction presents an overview of the SRS to help the reader understand how the document is organized and how to read and interpret it.

1.1 Purpose

Identify the product whose software requirements are specified in this document, including the revision or release number. If this SRS pertains to only part of the entire system, identify the portion or subsystem that it addresses.

1.2 Document Conventions

Describe any standards or typographical conventions that were followed when writing this SRS, including text styles, highlighting, or significant notations. For instance, state whether the priority shown for a high-level requirement is inherited by all of its detailed requirements, or whether every requirement statement has its own priority.

1.3 Intended Audience and Reading Suggestions

List the different readers to whom the SRS is directed, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of the SRS contains and how it is organized. Suggest a sequence for reading the document that is most appropriate for each type of reader.

1.4 Product Scope

Provide a short description of the software being specified and its purpose, including benefits and goals. Relate the software to corporate goals or business strategies. If a separate vision and scope document is available, refer to it rather than duplicating its contents here.

1.5 References

List any documents or other resources to which this SRS refers. These might include user interface style guides, contracts, standards, system requirements specifications, use-case documents, or the SRS for a related product. Provide enough information so that the reader can access each reference, including its title, author, version number, date, and source or location.

2. Overall Description

This section presents a high-level overview of the product being specified and the environment in which it will be used, the anticipated users of the product, and the known constraints, assumptions, and dependencies.

2.1 Product Perspective

Describe the context and origin of the product being specified in this SRS. State whether this product is the next member of a product family, the next release of a mature product, a

replacement for existing applications, or a new, self-contained product. If this SRS defines a component of a larger system, state how this software relates to the overall system and identify interfaces between the two.

2.2 Product Functions

Summarize the major functions the product must perform. Details will be provided in Section 4, so you only need a high-level summary here, such as a bulleted list. Organize the functions to make them understandable to any reader. A picture of the major groups of requirements and how they are related, such as a top-level data flow diagram or a class diagram, can be helpful.

2.3 User Classes and Characteristics

Identify the various user classes that you anticipate will use this product and describe their pertinent characteristics. (See Chapter 7.) Some requirements might pertain only to certain user classes. Distinguish the most important user classes for this product from those whom it is less critical to satisfy.

2.4 Operating Environment

Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and other software components or applications with which it must peacefully coexist.

2.5 Design and Implementation Constraints

Identify any issues that will restrict the options available to the developers and describe why they are constraints. Constraints might include the following:

- Specific technologies, tools, programming languages, and databases that must be used or avoided
- Required development conventions or standards (for instance, if the customer's organization will be maintaining the software, it might specify design notations and coding standards that a subcontractor must use)
- Corporate policies, government regulations, or industry standards
- Hardware limitations, such as timing requirements or memory restrictions
- Standard data interchange formats

2.6 Assumptions and Dependencies

List any assumed factors (as opposed to known facts) that could affect the requirements stated in the SRS. These could include commercial components that you plan to use, or issues around the development or operating environment. You might assume that the product will conform to a particular user interface design convention, whereas another SRS reader might assume something different. The project could be affected if these assumptions are incorrect, are not shared, or change.

Also, identify any dependencies the project has on external factors. For example, if you expect to integrate into the system some components that are being developed by another project, you are dependent upon that project to supply the correctly operating components on schedule. If these dependencies are already documented elsewhere, such as in the project plan, refer to those other documents here.

3. External Interface Requirements

Use this section to specify any requirements that ensure the new product will connect properly to external components. The context diagram shows the external interfaces at a high level of abstraction. Place detailed descriptions of the data and control components of the interfaces in the data dictionary. If different portions of the product have different external interfaces, incorporate an instance of this section within the detailed requirements for each such portion.

3.1 User Interfaces

State the software components for which a user interface is needed. Describe the logical characteristics of each user interface. The following are some characteristics you might include:

- GUI standards or product family style guides that are to be followed
- Screen layout or resolution constraints
- Standard buttons, functions, or navigation links that will appear on every screen (such as a help button)
- Shortcut keys
- Error message display standards

Document the user interface design details, such as specific dialog box layouts, in a separate user interface specification, not in the SRS.

3.2 Hardware Interfaces

Describe the characteristics of each interface between the software and hardware components of the system. This description might include the supported device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.

3.3 Software Interfaces

Describe the connections between this product and other external software components (identified by name and version), including databases, operating systems, tools, libraries, and integrated commercial components. Identify and describe the purpose of the data items or messages exchanged among the software components. Describe the services needed and the nature of the intercomponent communications. Identify data that will be shared across software components. If the data-sharing mechanism must be implemented in a specific way, such as a global data area in a multitasking operating system, specify this as an implementation constraint.

3.4 Communications Interfaces

Describe the requirements associated with any communication functions the product will use, including e-mail, Web browser, network communications standards or protocols, electronic forms, and so on. Define any pertinent message formatting. Specify communication security or encryption issues, data transfer rates, and synchronization mechanisms.

4. System Features

The template in Figure 9-1 shows the functional requirements organized by system features, the major services provided by the product. You might prefer to organize this section by use case, mode of operation, user class, object class, or functional hierarchy (IEEE 1998). You can

also use hierarchical combinations of these elements. Select an organizational approach that makes it easy for readers to understand the intended product.

4.x System Feature X

State the name of the feature in just a few words, such as "4.1 Spell Check and Spelling Dictionary Management." You will repeat subsections 4.x.1 through 4.x.3 for each system feature.

4.x.1 Description and Priority

Provide a short description of the feature and indicate whether it is of high, medium, or low priority. Alternatively, you could include specific priority component ratings, such as benefit, penalty, cost, and risk, each rated on a relative scale of 1 (low) to 9 (high). (See Chapter 13.)

4.x.2 Stimulus/Response Sequences

List the sequences of input stimuli (user actions, signals from external devices, or other triggers) and system responses that define the behaviors for this feature. These sequences will correspond to the dialogue elements associated with use cases, as was discussed in Chapter 8.

4.x.3 Functional Requirements

Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present for the user to carry out the services provided by the feature or to perform the task specified by a use case. Describe how the product should respond to anticipated error conditions or invalid input or actions. Uniquely identify each requirement, as described earlier in this chapter.

5. Other Nonfunctional Requirements

Use this section to list any nonfunctional requirements other than external interface requirements and constraints.

5.1 Performance Requirements

State any product performance requirements for various usage scenarios, and explain their rationale to help the developers make suitable design choices. Specify the number of concurrent users or operations to be supported, response times, and the timing relationships for real-time systems. You could also specify capacity requirements here, such as memory and disk space requirements or the maximum number of rows stored in database tables. Quantify the performance requirements as specifically as possible. You might need to state performance requirements for individual functional requirements or features, rather than collecting them all in one section. For example, "95% of catalog database queries shall be completed within 2 seconds on a 450-MHz Pentium II PC running Microsoft Windows 2000 with at least 50% of the system resources free."

5.2 Safety Requirements

Specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product. Define any safeguards or actions that must be taken, as well as potentially dangerous actions that must be prevented. Identify any safety certifications, policies, or regulations to which the product must conform. An example of a

safety requirement is: “An operation shall be terminated within 1 second if the measured tank pressure exceeds 95 percent of the specified maximum pressure.”

5.3 Security Requirements

Specify any requirements regarding security, integrity, or privacy issues that affect the use of the product and protection of the data used or created by the product. Define any user authentication or authorization requirements. Identify any security or privacy policies or certifications the product must satisfy. You might prefer to address these requirements through the quality attribute called integrity, which is described in Chapter 11. An example of a security requirement is: “Every user must change his initially assigned login password immediately after his first login. The initial password cannot be reused.”

5.4 Software Quality Attributes

Specify any additional product quality characteristics that will be important to either customers or developers. (See Chapter 11.) These characteristics should be specific, quantitative, and verifiable when possible. At the least, indicate the relative preferences for various attributes, such as ease of use over ease of learning, or portability over efficiency.

5.5 Business Rules

List any operating principles about the product, such as which individuals or roles can perform which functions under specific circumstances. These are not functional requirements in themselves, but they might imply certain functional requirements to enforce the rules. An example of a business rule is: “Only users having supervisory job codes can issue cash refunds of $100.00 or more.”

5.6 User Documentation

List the user documentation components that will be delivered along with the software, such as user manuals, online help, and tutorials. Identify any known user documentation delivery formats or standards.

6. Other Requirements

Define any other requirements that are not covered elsewhere in the SRS, such as internationalization requirements or legal requirements. You could also add sections on operations, administration, and maintenance to cover requirements for product installation, configuration, startup and shutdown, recovery and fault tolerance, and logging and monitoring operations. Add any new sections to the template that are pertinent to your project. If you don’t have to add any other requirements, omit this section.

Appendix A: Glossary

Define all the terms necessary for a reader to properly interpret the SRS, including acronyms and abbreviations. You might wish to build a separate glossary that spans multiple projects for the entire organization and include only terms that are specific to a single project in each SRS.

Appendix B: Analysis Models

This optional section includes, or refers to the existence and locations of, pertinent analysis models such as data flow diagrams, class diagrams, state -transition diagrams, or entity-relationship diagrams. (See Chapter 10.)

Appendix C: To-Be-Determined List

Compile a numbered list of the TBD (to be determined) references that remain in the SRS so that they can be tracked to closure.

Guidelines for Writing Requirements

There is no formulaic way to write excellent requirements; the best teacher is experience. Learning from the problems you have encountered in the past will teach you much. Many requirements documents can be improved by following effective technical-writing style guidelines and by employing user terminology rather than computer jargon (Kovitz 1999). Keep the following recommendations in mind as you document software requirements:

- Keep sentences and paragraphs short.
- Use the active voice.
- Write complete sentences that have proper grammar, spelling, and punctuation.
- Use terms consistently and as defined in the glossary.
- State requirements in a consistent fashion, such as "The system shall" or "The user shall," followed by an action verb, followed by the observable result. For example, "The stockroom manager subsystem shall display a list of all containers of the requested chemical that are currently in the chemical stockroom."
- To reduce ambiguity, avoid vague, subjective terms such as user-friendly, easy, simple, rapid, efficient, support, several, state-of-the-art, superior, acceptable, and robust. Find out what the customers really mean when they say "user-friendly" or "fast" or "robust" and state those expectations in the requirements.
- Avoid comparative words such as improve, maximize, minimize, and optimize. Quantify the degree of improvement that is needed, or state the maximum and minimum acceptable values of some parameter. Make sure you know what the customers mean when they say the new system should "process," "support," or "manage" something. Ambiguous language leads to unverifiable requirements.

Because requirements are written hierarchically, decompose an ambiguous top-level requirement into sufficient lower levels to clarify it and remove the ambiguity. Write requirements in enough detail so that if the requirement is satisfied, the customer's need will be met, but not with so much detail as to unnecessarily constrain the design. If you could satisfy a requirement in several ways and all are acceptable, the level of detail is probably adequate. However, if a designer who is reviewing the SRS is not clear on the customer's intent, you need to include additional detail to reduce the risk of reworking the product if a misunderstanding is subsequently revealed.

Requirements authors often struggle to find the right level of granularity. A helpful guideline is to write individually testable requirements. If you can think of a small number of related test cases to verify that a requirement was correctly implemented, it is probably at the right level of detail. If the tests you envision are numerous and diverse, perhaps several requirements were lumped together that need to be separated. Testable requirements have been suggested as a metric for software product size (Wilson 1995).

Write requirements at a consistent level of detail. I have seen requirement statements in the same SRS that varied widely in their scope. For example, "The keystroke combination Control-S shall be interpreted as File Save" and "The keystroke combination Control-P shall be interpreted as File Print" were split out as separate requirements. However, "The product shall

respond to editing directives entered by voice" describes an entire subsystem, not a single functional requirement.

Avoid long narrative paragraphs that contain multiple requirements. Conjunctions such as "and" and "or" in a requirement suggest that several requirements have been combined. Never use "and/or" or "et cetera" in a requirement statement.

Avoid stating requirements redundantly in the SRS. Although including the same requirement in multiple places might make the document easier to read, it also makes it harder to maintain. The multiple instances of the requirement all have to be updated at the same time, lest an inconsistency creep in. Cross-reference related items in the SRS to help keep them synchronized when making changes. Storing individual requirements just once in a requirements management tool or database alleviates the redundancy problem.

Think about the most effective way to represent each requirement. Consider a set of requirements that fit the following pattern: "The Text Editor shall be able to parse <*format*> documents that define <*jurisdiction*> laws." There are three possible values for <*format*> and four possible values for <*jurisdiction*>, for a total of 12 similar requirements. When you review a list of requirements that are this similar, it is hard to spot one that is missing, such as "The Text Editor shall be able to parse untagged documents that define international laws." Represent requirements that follow a pattern like this in a table, as illustrated in Table 9-1, to make sure you haven't missed any. The higher-level requirement could be stated as "ED-13. The Text Editor shall be able to parse documents in several formats that define laws in various jurisdictions, as shown in Table <xx>."

Table 9-1 Sample Tabular Format for Listing Requirement Numbers That Fit a Pattern

Jurisdiction	Tagged Format	Untagged Format	ASCII Format
Federal	ED-13.1	ED-13.2	ED-13.3
State	ED-13.4	ED-13.5	ED-13.6
Territorial	ED-13.7	ED-13.8	ED-13.9
International	ED-13.10	ED-13.11	ED-13.12

Sample Requirements, Before and After

Chapter 1 identified several characteristics of high-quality requirement statements: complete, correct, feasible, necessary, prioritized, unambiguous, and verifiable. Because requirements that don't exhibit these characteristics will cause confusion and rework down the road, you need to find and correct any problems as early as possible. Below are several requirements, adapted from real projects, that have some problems. Examine each statement in the context of these quality characteristics to see if you can spot the problems. I've presented my thoughts about what is wrong with each one and a suggested improvement. I have no doubt that another pass through my suggestions would make them even better, but your goal is not to write perfect requirements. It is to write requirements that are good enough to let your team proceed with design and construction at an acceptable level of risk.

Example 1

> *"The product shall provide status messages at regular intervals not less than every 60 seconds."*

This requirement seems to be incomplete: What are the status messages and under what conditions are they "provided" to the user? How long do they remain visible? What part of "the product" are we talking about? The timing interval is confusing, too. Is the interval between status messages supposed to be at least 60 seconds, so showing a new message every year is okay? If the intent is to have no more than 60 seconds elapse between messages, would one millisecond be too short? How consistent does the message display interval have to be? The word "every" just muddles the issue. Because of these problems, the requirement is not verifiable.

Here is one way we could rewrite the requirement to address those shortcomings (making some guesses about what was intended, which we must confirm with the customer):

1. The Background Task Manager (BTM) shall display status messages in a designated area of the user interface.
2. The messages shall be updated every 60 (plus or minus 10) seconds after background task processing begins and shall remain visible continuously.
3. If background task processing is progressing normally, the BTM shall display the percentage of the background task processing that has been completed.
4. The BTM shall display a "Done" message when the background task is completed.
5. The BTM shall display an error message if the background task has stalled.

I split this into multiple requirements because each will require separate test cases and to make each one individually traceable. If several requirements are strung together in a paragraph, it is easy to overlook one during construction or testing. Notice that the revised requirement does not specify precisely how the status messages will be displayed. That's a design issue, and if you specify it here, it becomes a design constraint placed on the developer. Prematurely constrained design options frustrate the programmers and can result in a suboptimal product design.

Example 2

"The product shall switch between displaying and hiding nonprinting characters instantaneously."

Computers cannot do anything instantaneously, so this requirement isn't feasible. It is also incomplete because it does not state the cause of the state switch. Is the software making the change on its own under certain conditions, or does the user take some action to initiate the change? Also, what is the scope of the display change within the document: selected text, the entire document, or what? There is an ambiguity problem too. Are "nonprinting" characters hidden text, or are they attribute tags or control characters of some kind? Because of these problems, this requirement cannot be verified.

This might be a better way to write the requirement: "The user shall be able to toggle between displaying and hiding all HTML markup tags in the document being edited with the activation of a specific triggering mechanism." Now it's clear that the nonprinting characters are HTML markup tags. The modified requirement indicates that the user triggers the display change, but it doesn't constrain the design because it doesn't define the precise mechanism used. When the designer selects an appropriate triggering mechanism (such as a hot key, a menu command, or voice input), you can write specific tests to verify whether the toggle operates correctly.

Example 3

"The parser shall produce an HTML markup error report that allows quick resolution of errors when used by HTML novices."

The word “quick” is ambiguous. The lack of definition of what goes into the error report indicates incompleteness. I’m not sure how you would verify this requirement. Find someone who calls herself an HTML novice and see if she can resolve errors quickly enough using the report? It is also not clear whether the HTML novice’s usage refers to the parser or the error report. And when is the report generated?

Let’s try this instead:

- After the HTML Parser has completely parsed a file, it shall produce an error report that contains the line number and text of any HTML errors found in the parsed file and a description of each error found.
- If no parsing errors are found, the error report shall not be produced.

Now we know when the error report is generated and what goes in it, but we’ve left it up to the designer to decide what the report should look like. We’ve also specified an exception condition: if there aren’t any errors, don’t generate a report.

Example 4

“Charge numbers should be validated online against the master corporate charge number list, if possible.”

I give up: What does “if possible” mean? If it’s technically feasible? If the master charge number list can be accessed online? If you aren’t sure whether a requested capability can be delivered, use TBD to indicate that the issue is unresolved. The requirement is incomplete because it doesn’t specify what happens if the validation passes or fails. Avoid imprecise words such as “should.” The customer either needs this functionality or he doesn’t. Some requirements specifications use subtle distinctions among keywords such as *shall, should,* and *may* as a way to indicate importance. I prefer to stick with *shall* or *will* as a clear statement of the requirement’s intent and to specify the priorities explicitly.

Here is an improved version of this requirement:

> “The system shall validate the charge number entered against the online master corporate charge number list. If the charge number is not found on the list, the system shall display an error message and the order shall not be accepted.”

A second related requirement might document the exception condition of the master corporate charge number list not being available at the time the validation was attempted.

Example 5

“The product shall not offer search and replace options that could have disastrous results.”

The notion of “disastrous results” is certainly subject to interpretation. Making an unintended global change while editing a document could be disastrous if the user does not detect the error or has no way to correct it. You should also be judicious in the use of inverse requirements, which describe things the system will not do. The underlying concern seems to pertain to protecting the file contents from inadvertent damage. Perhaps the real requirements are for a multilevel undo capability and confirmation of global changes or other actions that might result in data loss.

The Data Dictionary

Long ago, I worked on a project in which, on several occasions, the three programmers inadvertently used different variable names, lengths, and validations for the same data item. This caused confusion about what the real data definition was, truncation of data when it was stored in a variable that was too small, and maintenance headaches. We suffered from the lack of a data dictionary—a shared repository that defines the meaning, type, data size and format, units of measurement, precision, and allowed range or list of values for all data elements and structures used in the application.

The data dictionary is the glue that holds the various requirements documents and analysis models together. Integration problems are reduced if all developers comply with the contents of a data dictionary. To avoid redundancy and inconsistencies, establish a separate data dictionary for your project, rather than defining each data item in every requirement in which it appears. Maintain the data dictionary separately from the SRS, in a location that any stakeholder can access at any stage of the product's development or maintenance.

Items in the data dictionary are represented using a simple notation (Robertson and Robertson 1994). The item is shown on the left side of an equal sign, with its definition on the right. This notation defines primitive data elements, the composition of multiple data elements into structures, iteration (repeats) of a data item, enumerated values for a data item, and optional data items. The examples shown below are from (of course!) the Chemical Tracking System.

Primitive data elements A primitive data element is one for which no decomposition or subdivision is possible or sensible. It can be assigned a scalar value. The primitive's definition should identify its data type, size, range of allowed values, and so on. Primitives typically are defined with a comment, which is any text delimited by asterisks:

Request ID = ** 6digit system generated sequential integer, beginning with 1, that uniquely identifies each request **

Composition A data structure or record contains multiple data items. If an element in the data structure is optional, enclose it in parentheses:

Requested Chemical = *Chemical ID*
\+ *Quantity*
\+ *Quantity Units*
\+ *(Vendor Name)*

This structure identifies all the information associated with a request for a specific chemical. The Vendor Name is optional because the person placing the request might not care from which vendor the chemical is ordered. Each data item that appears in a structure must itself appear in the data dictionary. Structures can incorporate other structures.

Iteration If multiple instances of an item can appear in a data structure, enclose that item in curly braces. If you know the allowed number of possible repeats, show that number in the form *minimum:maximum* in front of the opening brace:

Request = *Request ID*
\+ *Charge Number*
\+ *1:10{Requested Chemical}*

This example shows that a chemical request must contain at least one chemical, but not more than ten chemicals. Each request also has a single request ID and one charge number whose format would be defined elsewhere in the data dictionary.

Selection If a primitive data element can take on a limited number of discrete values, enumerate those values in a list:

Quantity Units = *["grams" | "kilograms" | "each"]*
** 9-character text string indicating the units associated with the quantity of chemical requested **

This entry shows that there are just three allowed values for the text string Quantity Units. The comment provides the informal definition of the data element.

The time you invest in creating a data dictionary and a glossary will be more than repaid by the time gained in avoiding the mistakes that happen when project participants do not share the same understanding of critical pieces of information. If you keep the glossary and data dictionary current, they will remain valuable tools throughout the system's maintenance life and during the development of related or follow-on systems.

Next Steps

- Take a page of functional requirements from your project's SRS. Examine each statement to see how well it complies with the characteristics of excellent requirements. Rewrite any requirements that don't measure up.
- If your organization doesn't already have a standard format for documenting requirements, convene a small working group to adopt a standard SRS template. Begin with the template in Figure 9-1 and adapt it to best meet the needs of your organization's projects and products. Agree on a convention for labeling individual requirements.
- Convene a group of three to seven project stakeholders to formally review the SRS for your project. Make sure each requirement statement is clear, feasible, verifiable, unambiguous, and so forth. Look for any conflicts between different requirements in the specification, for missing requirements, and for missing sections of the SRS. Follow through to make sure the defects you find are corrected in the SRS and in any downstream work products based on those requirements.

---

[1]Function points are a measure of the quantity of user-visible functionality of an application, independent of how it is constructed. You can estimate the function points from an understanding of the user requirements, based on the counts of internal logical files, external interface files, and external inputs, outputs, and queries (IFPUG 1999).